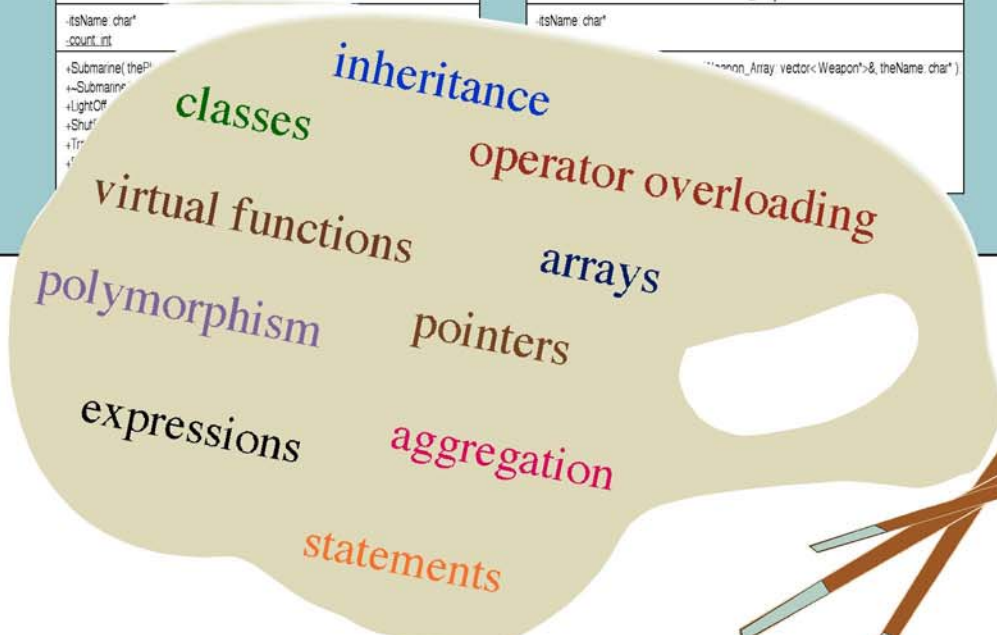
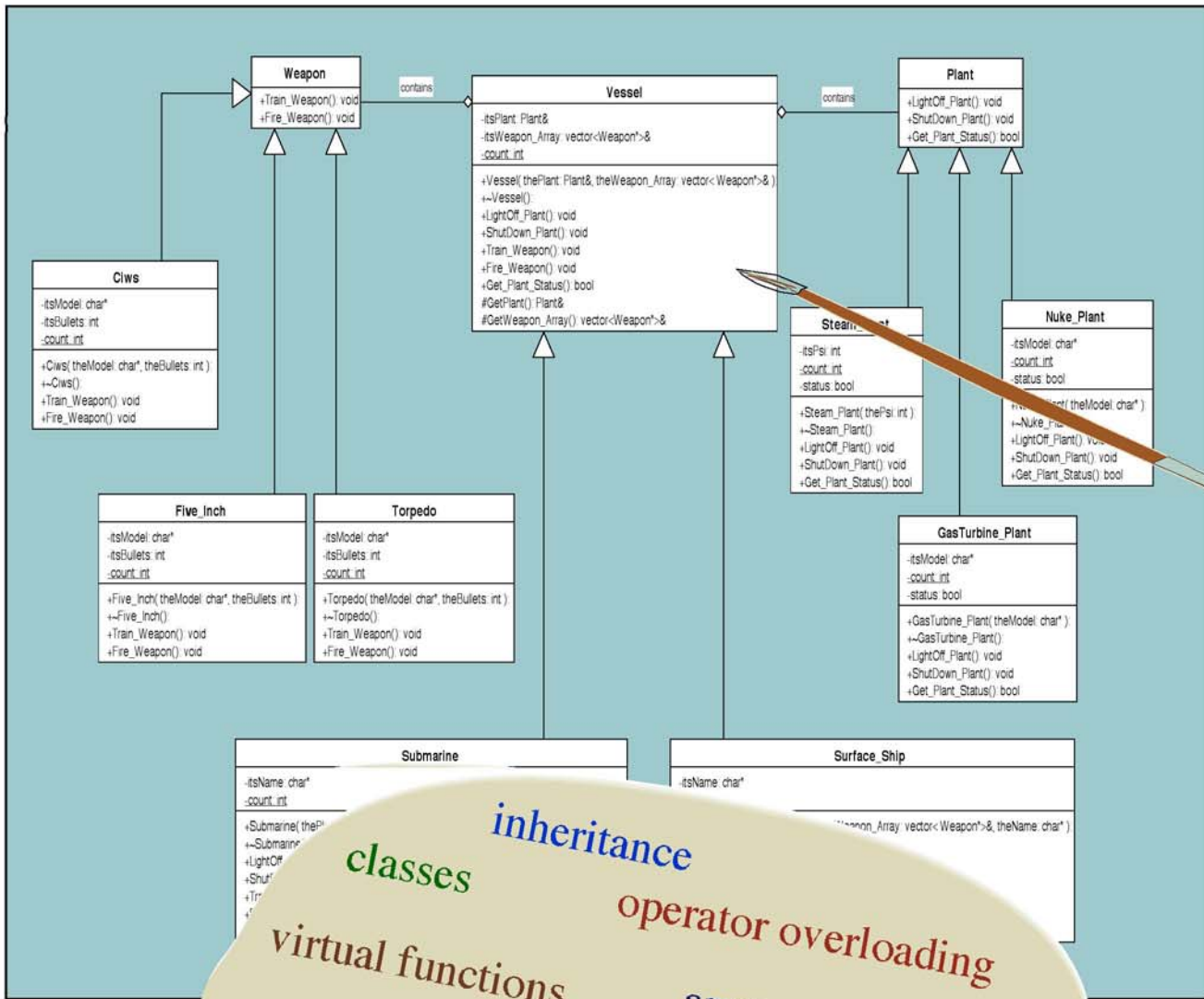


C++ FOR ARTISTS

THE ART, PHILOSOPHY, AND SCIENCE OF OBJECT-ORIENTED PROGRAMMING



CD Edition Enclosed!



Rick Miller
A Pulp FREE PRESS Book

UNLEASH THE CREATIVE GENIUS IN YOU!

C++ FOR ARTISTS

C++ FOR ARTISTS

THE ART, PHILOSOPHY, AND SCIENCE OF OBJECT-ORIENTED PROGRAMMING

RICK MILLER

Pulp FREE PRESS
FALLS CHURCH, VIRGINIA

Pulp FREE PRESS, Falls Church, Virginia 22042

www.pulpfreepress.com

info@pulpfreepress.com

©2003 Richard Warren Miller & Pulp Free Press— All Rights Reserved

No part of this book may be reproduced in any form without prior written permission from the publisher.

First edition published 2003

16 14 12 10 08 06 04

10 9 8 7 6 5 4 3 2

Pulp Free Press offers discounts on this book when ordered in bulk quantities. For more information regarding sales contact sales@pulpfreepress.com.

The eBook/PDF edition of this book may be licensed for bulk distribution. For whole or partial content licensing information contact licensing@pulpfreepress.com.

Publisher Cataloging-in-Publication Data: *Prepared by Pulp Free Press*

Miller, Rick, 1961 -

C++ For Artists: The Art, Philosophy, and Science of Object-Oriented Programming/Rick Miller.

p. cm.

Includes bibliographical references and index.

ISBN: 1-932504-02-8 (pbk)

1. C++ (Computer program language) I. Title.

2. Object-Oriented Programming (Computer Science)

QA76.73.C153M555 2003

005.13'3--dc21

2003093826

CIP

The source code provided in this book is intended for instructional purposes only. Although every possible measure was made by the author to ensure the programming examples contain source code of only the highest caliber, no warranty is offered, expressed, or implied regarding the quality of the code.

All product names mentioned in this book are trademark names of their respective owners.

C++ For Artists was meticulously crafted on a Macintosh PowerMac G4 using Adobe FrameMaker, Adobe Illustrator, Macromedia Freehand, Adobe Photoshop, Adobe Acrobat, Embarcadero Technologies Describe, ObjectPlant, Microsoft Word, Maple, and VirtualPC. Photos were taken with a Nikon F3HP, a Nikon FM, a Nikon CoolPix 800, and a Contax T3. C++ source code examples were prepared using Metrowerks CodeWarrior for Macintosh, Apple OSX developer tools, and Tenon Intersystems CodeBuilder. Java code examples prepared using Sun's Java 2 Standard Edition (J2SE) command line tool suite. Assembly language examples prepared using Microsoft Macro Assembler (MASM).

ISBN 1-932504-00-1 First eBook/PDF Edition

ISBN 1-932504-01-X First CD ROM Edition

ISBN 1-932504-02-8 First Paperback Edition

To CORALIE AND Kyle FOR THEIR ETERNAL PATIENCE

DETAILED CONTENTS

PREFACE

WELCOME – AND THANK YOU!	xlvi
TARGET AUDIENCE	xlvi
APPROACH	xlvi
ARRANGEMENT	xlvi
PART I: THE C++ STUDENT SURVIVAL GUIDE	xlvi
CHAPTER 1: AN APPROACH TO THE ART OF PROGRAMMING	xlvi
CHAPTER 2: SMALL VICTORIES: CREATING PROJECTS WITH IDES	xlvi
CHAPTER 3: PROJECT WALKTHROUGH: AN EXTENDED EXAMPLE	xlvi
CHAPTER 4: COMPUTERS, PROGRAMS, AND ALGORITHMS	xlvi
PART II: C++ LANGUAGE FUNDAMENTALS	xlvi
CHAPTER 5: SIMPLE PROGRAMS	xlvi
CHAPTER 6: CONTROLLING THE FLOW OF PROGRAM EXECUTION	xlvi
CHAPTER 7: POINTERS AND REFERENCES	xlvi
CHAPTER 8: ARRAYS	xlvi
CHAPTER 9: FUNCTIONS	xlvi
CHAPTER 10: TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES	xlvi
CHAPTER 11: DISSECTING CLASSES	xlvi
CHAPTER 12: COMPOSITIONAL DESIGN	xlvi
CHAPTER 13: EXTENDING CLASS FUNCTIONALITY THROUGH INHERITANCE	xlvi
PART III: IMPLEMENTING POLYMORPHIC BEHAVIOR	xlvi
CHAPTER 14: AD HOC POLYMORPHISM: OPERATOR OVERLOADING	xlvi
CHAPTER 15: STATIC POLYMORPHISM: TEMPLATES	xlvi
CHAPTER 16: DYNAMIC POLYMORPHISM: OBJECT-ORIENTED PROGRAMMING	xlvi
PART IV: INTERMEDIATE CONCEPTS	xlvi
CHAPTER 17: WELL-BEHAVED OBJECTS: THE ORTHODOX CANONICAL CLASS FORM	xlvi
CHAPTER 18: MIXED LANGUAGE PROGRAMMING	xlvi
CHAPTER 19: THREE DESIGN PRINCIPLES	xlvi
CHAPTER 20: USING A UML MODELING TOOL	xlvi
HOW TO READ C++ FOR ARTISTS	li
PEDAOGY	lii
CHAPTER LAYOUT	lii
LEARNING OBJECTIVES	lii
INTRODUCTION	lii
CONTENT	lii
QUICK REVIEWS	lii
SUMMARY	lii
SKILL BUILDING EXERCISES	lii
SUGGESTED PROJECTS	lii
SELF TEST QUESTIONS	lii
REFERENCES	lii
NOTES	lii
CD-ROM	liv
SUPPORTSITE™ WEBSITE	liv
PROBLEM REPORTING	liv
ACKNOWLEDGEMENTS	lv

PART I: THE C++ STUDENT SURVIVAL GUIDE

I AN APPROACH TO THE ART OF PROGRAMMING

INTRODUCTION	4
---------------------------	----------

<i>The Difficulties You Will Encounter Learning C++</i>	4
Required Skills	4
<i>The Planets Will Come Into Alignment</i>	4
How This Chapter Will Help You	5
PROJECT MANAGEMENT	5
<i>THREE SOFTWARE DEVELOPMENT ROLES</i>	5
Analyst	5
Architect	5
Programmer	6
<i>A PROJECT APPROACH STRATEGY</i>	6
You Have Been Handed A Project – Now What?	6
Strategy Areas of Concern	6
Think Abstractly	7
<i>THE STRATEGY IN A NUTSHELL</i>	7
<i>Applicability To The Real World</i>	7
THE ART OF PROGRAMMING	8
<i>DON'T START AT THE COMPUTER</i>	8
<i>INSPIRATION STRIKES AT THE WEIRDEST TIME</i>	8
<i>OWN YOUR OWN COMPUTER</i>	8
You Either Have Time and No Money, or Money and No Time	8
The Family Computer Is Not Going To Cut It!	9
<i>SET THE MOOD</i>	9
Location, Location, Location	9
<i>CONCEPT OF THE FLOW</i>	9
The Stages of Flow	9
<i>BE EXTREME</i>	10
The Programming Cycle	10
The Programming Cycle Summarized	11
<i>A Helpful Trick: Stubbing</i>	11
<i>Fix The First Compiler Error First</i>	11
MANAGING PROJECT COMPLEXITY	11
<i>SPLIT EVEN SIMPLE PROJECTS INTO MULTIPLE SOURCE CODE FILES</i>	12
Separating A Class's Interface from its Implementation	12
Benefits of Separating Interface from Implementation	12
Helpful Preprocessor Directives	13
The Final Word on Preprocessor Directive Behavior	14
<i>PROJECT FILE FORMAT</i>	14
Header File	14
Implementation File	15
Main File	15
<i>COMMENTING</i>	16
C-Style Comments	16
C++-style Comments	17
Write Self-Commenting Code: Give Identifiers Meaningful Names	17
Adopt A Convention And Stick With It	19
Restrict The Number of Global Variables	19
Minimize Coupling, Maximize Cohesion	19
TEXTBOOKS, REFERENCE BOOKS, AND QUICK REFERENCE GUIDES	19
SUMMARY	20
SKILL BUILDING EXERCISES	20
SUGGESTED PROJECTS	21
SELF TEST QUESTIONS	22
REFERENCES	22
NOTES	23

2 SMALL VICTORIES: CREATING PROJECTS WITH IDE'S

INTRODUCTION	26
THE PROGRAM CREATION PROCESS	26
INTEGRATED DEVELOPMENT ENVIRONMENTS	27
<i>Metrowerks CodeWarrior</i>	28
<i>Microsoft Visual C++</i>	32
<i>Intermission</i>	35
<i>Tenon Intersystems MachTen CodeBuilder™</i>	36

ATTENTION LINUX USERS	36
ORGANIZING PROJECT FILES	37
CREATING SOURCE FILES	37
CREATING MAKEFILE	37
SUMMARY	39
SKILL BUILDING EXERCISES	39
SUGGESTED PROJECTS	40
SELF TEST QUESTIONS	40
REFERENCES	40
NOTES	41

3 PROJECT WALKTHROUGH: AN EXTENDED EXAMPLE

INTRODUCTION	44
THE PROJECT APPROACH STRATEGY	44
THE DEVELOPMENT CYCLE	45
THE PROJECT SPECIFICATION	46
<i>ANALYZING THE PROJECT SPECIFICATION</i>	<i>47</i>
REQUIREMENTS.....	47
PROBLEM DOMAIN	48
LANGUAGE FEATURES.....	50
<i>DESIGN (FIRST ITERATION)</i>	<i>51</i>
<i>IMPLEMENTATION (FIRST ITERATION)</i>	<i>53</i>
<i>TESTING (FIRST ITERATION)</i>	<i>55</i>
<i>INTEGRATION (FIRST ITERATION)</i>	<i>55</i>
<i>DESIGN (SECOND ITERATION)</i>	<i>56</i>
FUNCTION STUBBING	56
OTHER CONSIDERATIONS	56
<i>IMPLEMENTATION (SECOND ITERATION)</i>	<i>57</i>
<i>TESTING (SECOND ITERATION)</i>	<i>59</i>
<i>INTEGRATION (SECOND ITERATION)</i>	<i>60</i>
<i>DESIGN (THIRD ITERATION)</i>	<i>60</i>
<i>IMPLEMENTATION (THIRD ITERATION)</i>	<i>60</i>
<i>TESTING (THIRD ITERATION)</i>	<i>62</i>
<i>INTEGRATION (THIRD ITERATION)</i>	<i>64</i>
<i>DESIGN (FOURTH ITERATION)</i>	<i>64</i>
<i>IMPLEMENTATION (FOURTH ITERATION)</i>	<i>66</i>
<i>TESTING (FOURTH ITERATION)</i>	<i>68</i>
<i>INTEGRATION (FOURTH ITERATION)</i>	<i>68</i>
<i>DESIGN (FIFTH ITERATION)</i>	<i>68</i>
<i>IMPLEMENTATION (FIFTH ITERATION)</i>	<i>70</i>
<i>TESTING (FIFTH ITERATION)</i>	<i>70</i>
<i>INTEGRATION (FIFTH ITERATION)</i>	<i>71</i>
WRAPPING UP THE PROJECT	72
COMPLETE ROBOT RAT SOURCE CODE LISTING	72
SUMMARY	76
SKILL BUILDING EXERCISES	76
SUGGESTED PROJECTS	76
SELF TEST QUESTIONS	76
REFERENCES	77
NOTES	77

4 COMPUTERS, PROGRAMS, & ALGORITHMS

INTRODUCTION	80
WHAT IS A COMPUTER?	80
COMPUTER VS. COMPUTER SYSTEM	80

COMPUTER SYSTEM	80
PROCESSOR	82
THREE ASPECTS of COMPUTER ARCHITECTURE	83
FEATURE SET	83
FEATURE SET IMPLEMENTATION	83
FEATURE SET ACCESSIBILITY	83
WHAT IS A PROGRAM?	83
TWO VIEWS of A PROGRAM	84
THE HUMAN PERSPECTIVE	84
THE COMPUTER PERSPECTIVE	84
CONCEPT of OBSERVABLE BEHAVIOR	84
THE C++ PROGRAM TRANSFORMATION PROCESS	85
PHASE 1	85
PHASE 2	85
PHASE 3	85
PHASE 4	86
PHASE 5	86
PHASE 6	86
PHASE 7	86
PHASE 8	87
PHASE 9	87
THE PROCESSING Cycle	87
FETCH	88
DECODE	88
EXECUTE	88
STORE	88
Why A Program Crashes	88
MEMORY ORGANIZATION	88
MEMORY BASICS	88
MEMORY Hierarchy	89
BITS, BYTES, WORDS	89
ALIGNMENT AND Addressability	90
ALGORITHMS	91
Good vs. Bad Algorithms	92
DON'T REINVENT THE WHEEL!	94
SUMMARY	94
SKILL BUILDING EXERCISES	94
SUGGESTED PROJECTS	94
SELF TEST QUESTIONS	95
REFERENCES	95
NOTES	95

PART II: C++ LANGUAGE FUNDAMENTALS

5 Simple Programs

INTRODUCTION	100
A MINIMAL C++ PROGRAM	100
Disassembly is a GREAT LEARNING TOOL	101
ANOTHER C++ PROGRAM	102
PARTS of the PROGRAM	103
COMMENTS	103
PREPROCESSOR DIRECTIVE	103
LIBRARIES	103
Using Directive	103
main() FUNCTION	103
CONSTANTS	103
VARIABLES	103

<i>STATEMENTS AND EXPRESSIONS</i>	103
Keywords	104
FUNDAMENTAL TYPES	104
<i>DETERMINING YOUR DATA TYPE RANGES</i>	105
<i>DETERMINING DATA TYPE SIZE WITH THE sizeof OPERATOR</i>	106
LITERALS	106
<i>INTEGER LITERALS</i>	107
<i>Decimal</i>	107
<i>Octal</i>	107
<i>Hexadecimal</i>	107
<i>A WORD OF CAUTION</i>	107
<i>CHARACTER LITERALS</i>	108
<i>Single Character Literals</i>	108
<i>Multiple Character Literals</i>	108
<i>Escape Sequences</i>	109
<i>FLOATING POINT LITERALS</i>	110
<i>STRING LITERALS</i>	111
<i>BOOLEAN LITERALS</i>	111
EXPRESSIONS	111
OPERATORS	113
<i>OPERATOR PRECEDENCE</i>	114
<i>USE PARENTHESES</i>	115
<i>Multiplicative Operators</i>	116
<i>Multiplication Operator</i>	116
<i>Division Operator</i>	116
<i>Modulus Operator</i>	117
<i>Additive Operators</i>	117
<i>Addition Operator</i>	117
<i>Subtraction Operator</i>	118
<i>Shift Operators</i>	118
<i>Left Shift Operator</i>	118
<i>Right Shift Operator</i>	119
<i>RELATIONAL OPERATORS</i>	120
<i>Less Than Operator</i>	120
<i>Greater Than Operator</i>	120
<i>Less Than or Equal To Operator</i>	121
<i>Greater Than or Equal To Operator</i>	121
<i>Equality Operators</i>	121
<i>Equal To Operator</i>	121
<i>Not Equal To Operator</i>	121
<i>Bitwise AND Operator - &</i>	121
<i>Bitwise Exclusive OR Operator - ^</i>	122
<i>Bitwise Inclusive OR Operator - </i>	122
<i>Logical AND Operator - &&</i>	123
<i>Logical OR Operator - </i>	123
<i>Conditional Operator - ? :</i>	123
<i>ASSIGNMENT OPERATORS</i>	124
<i>lvalue vs. rvalue</i>	124
<i>Compound Assignment Operators</i>	125
<i>Comma Operator - ,</i>	125
<i>INCREMENT AND DECREMENT OPERATORS (++, --)</i>	125
Identifiers	126
<i>Identifier Naming Conventions</i>	126
<i>Hungarian Notation</i>	126
CONSTANTS	128
VARIABLES	128
<i>Declaring</i>	128
<i>Scope</i>	128
<i>Local Scope</i>	129
<i>Function Scope</i>	130
<i>File Scope</i>	130
Multifile Variable Usage	131
<i>Sharing File Scope Variables Across Multiple Files</i>	131
<i>Limiting File Scope Variable Visibility to One File</i>	131
THE main() FUNCTION	132

<i>THE PURPOSE OF THE main() FUNCTION</i>	132
<i>TWO FORMS OF main()</i>	132
<i>EXITING main()</i>	133
<i>CALLING FUNCTIONS UPON EXITING main()</i>	133
Simple Input and Output	133
<i>cin</i>	134
<i>Trapping Bad Input</i>	134
<i>cout</i>	135
<i>LEARNING MORE ABOUT cout AND cin</i>	135
SUMMARY	135
Skill Building Exercises	136
SUGGESTED PROJECTS	137
Self Test Questions	138
REFERENCES	138
NOTES	139

6 CONTROLLING THE FLOW OF PROGRAM EXECUTION

INTRODUCTION	142
STATEMENTS, Null STATEMENTS, AND Compound STATEMENTS	142
<i>STATEMENTS</i>	142
<i>Null STATEMENTS</i>	142
<i>Compound STATEMENTS</i>	143
SELECTION STATEMENTS	143
<i>if STATEMENT</i>	143
<i>if STATEMENTS AND Compound STATEMENTS</i>	144
<i>if-else STATEMENT</i>	145
<i>Nesting if-else STATEMENTS</i>	146
<i>switch STATEMENT</i>	147
<i>Break AND THE Default CASE</i>	150
ITERATION STATEMENTS	150
<i>while STATEMENT</i>	150
<i>Controlling while STATEMENTS with Sentinel Values</i>	151
<i>Nesting while STATEMENTS</i>	152
<i>Doing SOMETHING FOREVER</i>	152
<i>Exiting While Loops with the break STATEMENT</i>	152
<i>do STATEMENT</i>	155
<i>Nesting do STATEMENTS</i>	155
<i>for STATEMENT</i>	155
<i>Nesting for STATEMENTS</i>	157
<i>break</i>	157
<i>Doing SOMETHING FOREVER with a for STATEMENT</i>	157
<i>CONTINUE</i>	158
<i>Avoiding break AND CONTINUE</i>	158
Writing ELEGANT Code	158
Labeled STATEMENTS	159
<i>goto STATEMENT</i>	159
<i>Advice ON Using goto</i>	159
CONTROL STATEMENT Usage Guide	160
SUMMARY	160
Skill Building Exercises	161
SUGGESTED PROJECTS	162
Self Test Questions	163
REFERENCES	164
NOTES	164

7 POINTERS AND REFERENCES

INTRODUCTION	166
BUT FIRST, A SHORT STORY	166
<i>WHAT IS AN OBJECT?</i>	167
<i>WHAT IS A MEMORY ADDRESS?</i>	168
<i>HOW DO YOU DETERMINE AN OBJECT'S MEMORY ADDRESS?</i>	169
<i>WHAT IS A POINTER?</i>	171
<i>HOW DO YOU DECLARE A POINTER?</i>	172
<i>HOW DO YOU ACCESS THE OBJECT A POINTER POINTS TO?</i>	173
<i>HOW DO YOU DYNAMICALLY CREATE AND DELETE OBJECTS?</i>	174
<i>THE NEW OPERATOR</i>	175
<i>A NEAT TRICK: CALLING OBJECT CONSTRUCTORS</i>	177
<i>WHAT'S THE DIFFERENCE BETWEEN A POINTER AND A REFERENCE?</i>	177
<i>HOW DO YOU DECLARE AND USE REFERENCES?</i>	178
SUMMARY	178
Skill Building Exercises	179
SUGGESTED PROJECTS	179
Self Test Questions	180
REFERENCES	180
NOTES	181

8 ARRAYS

INTRODUCTION	184
WHAT IS AN ARRAY?	184
<i>LOCATING ARRAY ELEMENTS</i>	185
DECLARING & DEFINING STATICALLY ALLOCATED ARRAYS	185
<i>SINGLE-DIMENSIONAL ARRAYS</i>	185
<i>ACCESSING ARRAY ELEMENTS</i>	186
<i>SUBSCRIPT METHOD</i>	186
<i>POINTER ARITHMETIC METHOD</i>	187
<i>BEWARE THE UNINITIALIZED ARRAY!</i>	187
<i>COMBINING ARRAY DEFINITION WITH ARRAY DECLARATION</i>	187
<i>ARRAYS OF POINTERS</i>	188
<i>MULTI-DIMENSIONAL ARRAYS</i>	189
<i>ARRAYS OF TWO DIMENSIONS</i>	189
<i>ARRAYS OF THREE OR MORE DIMENSIONS</i>	191
<i>AUTOMATIC INITIALIZATION OF MULTI-DIMENSIONAL ARRAYS</i>	193
DECLARING AND DEFINING DYNAMIC ARRAYS	196
<i>DYNAMICALLY ALLOCATED SINGLE DIMENSIONAL ARRAYS</i>	196
<i>DYNAMICALLY ALLOCATED MULTI-DIMENSIONAL ARRAYS</i>	197
STRINGS	200
SUMMARY	200
Skill Building Exercises	201
SUGGESTED PROJECTS	202
Self Test Questions	203
REFERENCES	203
NOTES	204

9 FUNCTIONS

INTRODUCTION	206
WHAT IS A FUNCTION?	206
<i>INTERFACE VS. IMPLEMENTATION</i>	207
<i>PUT FUNCTION INTERFACE DECLARATIONS IN HEADER FILES</i>	207
<i>#ifndef...#define...#endif</i>	207

<i>Put Function Definitions in Implementation Files</i>	207
<i>CHARACTERISTICS OF A WELL-WRITTEN FUNCTION</i>	208
DECLARING AND DEFINING FUNCTIONS	208
<i>NAMING FUNCTIONS</i>	208
<i>FUNCTION DECLARATION</i>	209
<i>FUNCTION DEFINITION</i>	209
<i>FUNCTION CALLING</i>	209
<i>A COMPLETE EXAMPLE</i>	210
<i>Quick Review</i>	211
LOCAL FUNCTION VARIABLE SCOPING	212
<i>DECLARING LOCAL VARIABLES</i>	212
<i>HIDING GLOBAL VARIABLES WITH LOCAL VARIABLES</i>	212
<i>USING SCOPING BLOCKS IN FUNCTIONS</i>	212
<i>STATIC FUNCTION VARIABLES</i>	213
<i>SCOPE OF FUNCTION PARAMETERS</i>	214
<i>Quick Review</i>	215
PASSING ARGUMENTS TO FUNCTIONS	215
<i>FUNCTION CALLING</i>	215
<i>Responsibilities of the Calling Function</i>	216
<i>Responsibilities of the Called Function</i>	216
<i>PASSING ARGUMENTS BY VALUE</i>	216
<i>ANOTHER EXAMPLE</i>	218
<i>PASSING ARGUMENTS BY REFERENCE</i>	219
<i>CONTINUING THE STORY</i>	220
<i>PASSING POINTERS</i>	220
<i>PASSING REFERENCES</i>	221
<i>PASSING ARRAYS TO FUNCTIONS</i>	222
<i>PASSING MULTI-DIMENSIONAL ARRAYS TO FUNCTIONS</i>	223
<i>ANOTHER EXAMPLE</i>	224
USING FUNCTION RETURN VALUES	225
<i>RETURNING OBJECTS</i>	226
<i>THE RETURN KEYWORD: MANTRA ON PROPER USAGE</i>	226
<i>ANOTHER EXAMPLE</i>	227
<i>RETURNING POINTERS</i>	227
<i>HOW NOT TO RETURN A POINTER FROM A FUNCTION: AVOIDING THE DANGLING REFERENCE</i>	229
<i>RETURNING REFERENCES</i>	229
<i>Quick Review</i>	230
FUNCTION OVERLOADING	231
CALLING FUNCTIONS RECURSIVELY	232
<i>ANOTHER EXAMPLE</i>	233
FUNCTION POINTERS	234
<i>DECLARING FUNCTION POINTERS</i>	235
<i>ASSIGNING THE ADDRESS OF A FUNCTION TO A FUNCTION POINTER</i>	235
<i>CALLING THE FUNCTION VIA THE FUNCTION POINTER</i>	236
<i>ARRAYS OF FUNCTION POINTERS</i>	236
<i>IMPLEMENTING CALLBACK FUNCTIONS WITH FUNCTION POINTERS</i>	237
CREATING A FUNCTION LIBRARY	239
<i>STEPS TO CREATING A LIBRARY</i>	239
<i>CREATE EMPTY PROJECT</i>	239
<i>ADD IMPLEMENTATION FILE</i>	240
<i>SET LIBRARY TARGET SETTINGS</i>	240
<i>NAME LIBRARY AND SET PROJECT TYPE</i>	241
<i>BUILD THE PROJECT</i>	241
<i>USE THE LIBRARY</i>	241
SUMMARY	242
SKILL BUILDING EXERCISES	242
SUGGESTED PROJECTS	243
<i>EISCS MKI LANGUAGE SET</i>	244
<i>SAMPLE PROGRAM</i>	245
<i>BASIC OPERATION OF THE EISCS MKI</i>	245
<i>MEMORY</i>	245
<i>INSTRUCTION DECODING</i>	245
<i>ADDITIONAL EISCS SPECIFICATIONS</i>	246

SELF TEST QUESTIONS	247
REFERENCES	247
NOTES	248

10 TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES

INTRODUCTION	250
TOWARD DATA ABSTRACTION: typedef	250
<i>CREATING TYPE SYNONYMS</i>	250
CREATING ENUMERATED DATA TYPES WITH ENUM	252
<i>ENUMS AND SWITCH STATEMENTS</i>	252
<i>CHANGING AN ENUM'S DEFAULT STATE VALUES</i>	252
<i>ENUM STATE NAME CONFLICTS</i>	253
<i>The Utility of NAME SPACES</i>	253
<i>Quick Summary</i>	254
STRUCTURES: C-Style	254
<i>ACCESSING STRUCTURAL ELEMENTS</i>	255
<i>ACCESSING STRUCTURAL DATA MEMBERS VIA THE DOT "." OPERATOR</i>	255
<i>ACCESSING STRUCTURAL ELEMENTS VIA THE SHORTHAND MEMBER ACCESS ">" OPERATOR</i>	256
<i>Quick Summary</i>	258
STRUCTURES: C++-Style	259
<i>PERSON STRUCTURE REDESIGN</i>	259
<i>PUBLIC INTERFACE FUNCTIONS AND THE PUBLIC ACCESS SPECIFIER</i>	259
<i>PRIVATE DATA MEMBERS AND THE PRIVATE ACCESS SPECIFIER</i>	260
<i>THE DEEP SECRET: THE THIS POINTER</i>	262
<i>Quick Summary</i>	262
CLASSES: A GENTLE INTRODUCTION	263
<i>Quick Summary</i>	265
THE DIFFERENCES BETWEEN STRUCTURES & CLASSES	265
<i>Quick Summary</i>	265
OBJECT-ORIENTED THINKING	266
<i>Object Speak: A New Vocabulary</i>	266
SUMMARY	267
SKILL BUILDING EXERCISES	268
SUGGESTED PROJECTS	269
SELF TEST QUESTIONS	270
REFERENCES	270
NOTES	271

11 DISSECTING CLASSES

INTRODUCTION	274
THE CLASS CONSTRUCT	274
<i>PARTS OF A CLASS DECLARATION</i>	274
<i>A MINIMUM CLASS DECLARATION</i>	275
<i>PLACE CLASS DECLARATIONS IN SEPARATE HEADER FILES</i>	276
<i>THE UML CLASS DIAGRAM</i>	276
<i>THE CONCEPTS OF STATE AND BEHAVIOR</i>	276
<i>OBJECT STATE</i>	276
<i>OBJECT BEHAVIOR</i>	276
CLASS MEMBER FUNCTIONS	276
<i>CLASS MEMBER FUNCTION ACCESS TO CLASS ATTRIBUTES</i>	278
<i>OBTAINING ACCESS TO CLASS ATTRIBUTES FROM A MEMBER FUNCTION</i>	278
<i>OBTAINING ACCESS TO INSTANCE ATTRIBUTES FROM A MEMBER FUNCTION</i>	278
<i>SPECIAL MEMBER FUNCTIONS</i>	278
<i>CONSTRUCTOR</i>	278
<i>TESTCLASS EXAMPLE</i>	279
<i>COPY CONSTRUCTOR</i>	281

<i>TestClass Example Extended</i>	281
<i>Copy Assignment Operator</i>	282
<i>TestClass Example Extended</i>	283
<i>Destructor</i>	283
<i>TestClass Example Extended</i>	285
Behavior of Default Special Functions	286
<i>Quick Summary</i>	287
Accessor and Mutator Functions	287
<i>Accessor Functions</i>	288
<i>Mutator Functions</i>	288
<i>Quick Summary</i>	288
Using Access Specifiers To Control Horizontal Member Access	288
<i>The Concept of Horizontal Access</i>	289
<i>Data Encapsulation</i>	289
<i>Access Specifiers</i>	289
<i>The Public Access Specifier</i>	289
<i>The Protected Access Specifier</i>	289
<i>The Private Access Specifier</i>	289
Overloading Class Member Functions	290
<i>Function Signatures</i>	290
<i>Why would you want to overload member functions?</i>	290
Separating A Class's Interface From Its Implementation	293
<i>Manage Physical Complexity</i>	293
<i>Allow the Creation of Code Libraries</i>	293
A Complete Example: Class Person	293
<i>Summary</i>	296
<i>Skill Building Exercises</i>	297
<i>Suggested Projects</i>	298
<i>Self Test Questions</i>	298
<i>References</i>	299
<i>Notes</i>	299

12 Compositional Design

Introduction	302
Managing Physical Complexity	302
Aggregation	302
<i>Simple vs. Composite Aggregation</i>	302
<i>The Relationship Between Aggregation and Object Lifetime</i>	302
<i>Aggregation Example Code</i>	303
<i>Composite Aggregation Example</i>	303
<i>Another Composite Aggregation Example</i>	304
<i>Simple Aggregation Example</i>	305
Extending the Class Diagram	307
Sequence Diagrams	308
<i>Quick Summary</i>	308
The Aircraft Engine Simulation: An Extended Aggregation Example	309
<i>The Purpose of the Engine Class</i>	309
<i>An Engine and its Parts</i>	309
<i>The Engine Class</i>	311
The Entire Aircraft Engine Simulation Project	314
<i>aircraftutils.h</i>	314
<i>fuelpump.h</i>	314
<i>oilpump.h</i>	315
<i>TEMPERATURESENSOR.H</i>	315
<i>OXYGENSENSOR.H</i>	315
<i>COMPRESSOR.H</i>	316
<i>ENGINE.H</i>	316
<i>fuelpump.cpp</i>	317
<i>oilpump.cpp</i>	317

TEMPERATURESENSOR.CPP	318
OXYGENSENSOR.CPP	318
COMPRESSOR.CPP	319
ENGINE.CPP	320
MAIN.CPP	321
SUMMARY	322
Skill Building Exercises	322
SUGGESTED PROJECTS	323
SELF TEST QUESTIONS	323
REFERENCES	324
NOTES	324

13 EXTENDING CLASS FUNCTIONALITY THROUGH INHERITANCE

INTRODUCTION	328
PURPOSE AND USE OF INHERITANCE	328
EXPRESSING INHERITANCE WITH A UML CLASS DIAGRAM	328
IMPLEMENTING BASECLASS AND DERIVEDCLASSONE	329
<i>Quick Review</i>	<i>331</i>
ACCESS SPECIFIERS AND VERTICAL ACCESS	332
<i>PUBLIC INHERITANCE</i>	<i>333</i>
<i>PROTECTED INHERITANCE</i>	<i>333</i>
<i>PRIVATE INHERITANCE</i>	<i>333</i>
<i>Quick Review</i>	<i>334</i>
CALLING BASE CLASS CONSTRUCTORS	335
<i>Quick Review</i>	<i>337</i>
FUNCTION NAME HIDING: THIS IS NOT FUNCTION OVERRIDING!	337
<i>FUNCTION HIDING VS. FUNCTION OVERLOADING</i>	<i>337</i>
<i>Quick Review</i>	<i>340</i>
WHAT THEN IS FUNCTION OVERRIDING?	340
CREATING VIRTUAL FUNCTIONS: THE VIRTUAL KEYWORD	340
<i>PURPOSE OF VIRTUAL FUNCTIONS</i>	<i>340</i>
<i>DECLARING AND USING VIRTUAL FUNCTIONS</i>	<i>340</i>
<i>VIRTUAL DESTRUCTORS</i>	<i>341</i>
<i>Quick Review</i>	<i>342</i>
PURE VIRTUAL FUNCTIONS	342
<i>DECLARING PURE VIRTUAL FUNCTIONS</i>	<i>342</i>
ABSTRACT CLASSES	343
FLEET SIMULATION SOURCE CODE	346
<i>ciws.h</i>	<i>346</i>
<i>five_inch.h</i>	<i>347</i>
<i>TORPEDO.h</i>	<i>347</i>
<i>GASTURBINE.h</i>	<i>347</i>
<i>NUKE_PLANT.h</i>	<i>348</i>
<i>STEAM_PLANT.h</i>	<i>348</i>
<i>SUBMARINE.h</i>	<i>348</i>
<i>SURFACE_SHIP.h</i>	<i>349</i>
<i>ciws.cpp</i>	<i>349</i>
<i>five_inch.cpp</i>	<i>349</i>
<i>GASTURBINE_PLANT.cpp</i>	<i>350</i>
<i>NUKE_PLANT.cpp</i>	<i>350</i>
<i>STEAM_PLANT.cpp</i>	<i>351</i>
<i>SUBMARINE.cpp</i>	<i>351</i>
<i>SURFACE_SHIP.cpp</i>	<i>352</i>
<i>TORPEDO.cpp</i>	<i>352</i>
<i>VESSEL.cpp</i>	<i>353</i>

Multiple Inheritance	353
Virtual Base Classes: Virtual Inheritance	357
Getting Inheritance Right: Some Points To Consider	360
<i>Two Different Uses of Inheritance</i>	361
<i>Reasoning About Object-Oriented Application Design</i>	361
<i>Incremental Code Evolution</i>	361
<i>Protect Yourself In Your Design</i>	362
Summary	362
Skill Building Exercises	362
Suggested Projects	364
Self Test Questions	366
References	366
Notes	367

PART III: IMPLEMENTING POLYMORPHIC BEHAVIOR

14 Ad Hoc Polymorphism: Operator Overloading

Introduction	372
Ad Hoc Polymorphism: Function Overloading	372
The Goal Of Operator Overloading	372
Overloadable Operators	372
Overloading Operators	373
Overloading IOSTREAM INSERTION AND EXTRACTION OPERATORS: <<, >>	374
Overloading THE ASSIGNMENT OPERATOR: =	378
<i>Shallow Copy vs. Deep Copy</i>	379
Overloading RELATIONAL OPERATORS: <, >, <=, >=	380
Overloading EQUALITY OPERATORS: ==, !=	381
Overloading ARITHMETIC OPERATORS: +, -, *, /, %	383
<i>A Few Words About Error Checking</i>	384
Overloading THE SUBSCRIPT OPERATOR: []	384
Overloading COMPOUND ASSIGNMENT OPERATORS: +=, -=, *=, ETC.	386
Overloading INCREMENT & DECREMENT OPERATORS: ++, --	387
Overloading VARIOUS OTHER OPERATORS: (), +, -, <<, >>, ETC.	389
<i>The FUNCTION OPERATOR: OPERATOR()</i>	392
<i>The MEMBER OPERATOR: OPERATOR->()</i>	392
<i>The COMMA OPERATOR: OPERATOR,() - A.K.A. THE SEQUENCING OPERATOR</i>	392
VIRTUAL OVERLOADED OPERATORS	392
Summary	394
Skill Building Exercises	394
Suggested Projects	395
Self Test Questions	395
References	396
Notes	396

15 Static Polymorphism: Templates

Introduction	398
Definition of Template	398
<i>Function Templates</i>	398
<i>Class Templates</i>	398

<i>STRUCTURE TEMPLATES</i>	398
How Templates Work: An Analogy	398
Declaring And Using Function Templates	399
<i>SEPARATING DECLARATION FROM IMPLEMENTATION: SOME BACKGROUND</i>	399
<i>WHEN IN DOUBT REFER TO YOUR COMPILER DOCUMENTATION</i>	400
<i>EXAMPLE 15.1 CONTINUED</i>	400
<i>Using Multiple Placeholders</i>	400
<i>Quick Review</i>	402
Declaring And Using Class Templates	403
<i>A MORE COMPLEX CLASS TEMPLATE EXAMPLE</i>	404
<i>Quick Review</i>	405
OVERVIEW OF THE STANDARD TEMPLATE LIBRARY (STL)	405
<i>CONTAINERS AND CONTAINER ADAPTERS</i>	406
<i>ITERATORS</i>	407
<i>Algorithms</i>	408
<i>Quick Review</i>	408
Using Standard Template Library Components	408
<i>Using VECTOR</i>	408
<i>Using list</i>	411
<i>Quick Review</i>	411
SUMMARY	412
Skill Building Exercises	412
SUGGESTED PROJECTS	413
SELF TEST QUESTIONS	413
REFERENCES	414
NOTES	414

16 Dynamic Polymorphism: Object-Oriented Programming

INTRODUCTION	416
ABSTRACTION: Amplify The Essential—Eliminate The Irrelevant	416
OBJECT-ORIENTED PROGRAMMING DEFINED	417
DYNAMIC POLYMORPHISM DEFINED	417
LANGUAGE FEATURES THAT SUPPORT OBJECT-ORIENTED PROGRAMMING	417
AN EXAMPLE: CLASS INTERFACE	419
<i>Quick Review</i>	420
EXTENDED EXAMPLE: ENGINE COMPONENTS REVISITED	422
<i>A BASIS FOR COMPARISON</i>	422
<i>POLYMORPHIC ENGINE COMPONENT CODE</i>	424
<i>icomponent.h</i>	424
<i>component.h</i>	424
<i>component.cpp</i>	425
<i>pump.h</i>	425
<i>pump.cpp</i>	426
<i>sensor.h</i>	426
<i>sensor.cpp</i>	427
<i>waterpump.h</i>	427
<i>waterpump.cpp</i>	428
<i>oilpump.h</i>	428
<i>oilpump.cpp</i>	428
<i>fuelpump.h</i>	428
<i>fuelpump.cpp</i>	429
<i>airflowsensor.h</i>	429
<i>airflowsensor.cpp</i>	429
<i>oxysensor.h</i>	429
<i>oxysensor.cpp</i>	430
<i>temperaturesensor.h</i>	430
<i>temperaturesensor.cpp</i>	430
<i>engine.h</i>	431
<i>engine.cpp</i>	431
<i>smallengine.h</i>	432
<i>smallengine.cpp</i>	432
<i>engineutils.h</i>	433
<i>main.cpp</i>	433

<i>Discussion of the Polymorphic Engine Component Code</i>	434
<i>ICOMPONENT AND DERIVED CLASSES</i>	434
<i>WHAT IS MEANT BY A PURE VIRTUAL VS. A VIRTUAL MEMBER FUNCTION DECLARATION</i>	434
<i>ENGINE AND SMALLENGINE</i>	434
<i>RUNNING THE POLYMORPHIC ENGINE COMPONENT PROGRAM</i>	434
A SHORT STORY	435
<i>TAMING THE COMPLEXITY OF THE C++ LANGUAGE</i>	435
SUMMARY	436
SKILL BUILDING EXERCISES	436
SUGGESTED PROJECTS	439
SELF TEST QUESTIONS	440
REFERENCES	441
NOTES	441

PART IV: INTERMEDIATE CONCEPTS

17 WELL-BEHAVED OBJECTS: THE ORTHODOX CANONICAL CLASS

INTRODUCTION	446
WHAT IS A WELL-BEHAVED OBJECT?	446
<i>Object Usage Contexts</i>	446
<i>Object Creation</i>	446
<i>Object Copying</i>	447
<i>Object Assignment</i>	447
<i>Object Destruction</i>	447
<i>Other Contexts By Design</i>	447
THE ORTHODOX CANONICAL CLASS FORM (OCCF)	448
<i>Four Required Functions</i>	448
<i>Default Constructor</i>	449
<i>Destructor</i>	449
<i>Copy Constructor</i>	449
<i>Copy Assignment Operator</i>	449
<i>Implementing Foo Class OCCF Functions</i>	449
<i>Consider Future Desired Behavior</i>	449
<i>Extending Foo To Participate In Other Contexts: Overloading More Operators</i>	451
<i>Quick Review</i>	452
SUMMARY	453
SKILL BUILDING EXERCISES	453
SUGGESTED PROJECTS	454
SELF TEST QUESTIONS	455
REFERENCES	455
NOTES	455

18 MIXED LANGUAGE PROGRAMMING

INTRODUCTION	458
C++ AND C	458
<i>How C++ Allows Overloaded Functions: Name Mangling</i>	458
<i>EXTERN KEYWORD</i>	458
<i>Building a C Library: The SQUARE() Function</i>	458
<i>Deciphering C Standard Library Files</i>	464
<i>Quick Review</i>	464
C++ AND ASSEMBLY	465
<i>Some Things To Think About Before Using Assembly</i>	465
<i>Know Thy Implementation Dependencies</i>	465

<i>Inline Assembly Language in a C++ Function</i>	465
Linking An Object File Created From Assembly Language	467
Process Steps.....	467
Using Inline Assembly in the Macintosh Environment	469
Quick Review	470
C++ and Java: The Java Native Interface (JNI)	470
Steps To Create a JNI C++ Program	470
Win32 JNI Example	471
Step 1: Create Java Source File.....	471
Step 2: Compile Java Source File	472
Step 3: Create Header File	472
Step 4: Create C++ Source File.....	474
Step 5: Compile C++ Source File to Create Dynamic Link Library.....	474
Step 6: Run Java Program	474
Macintosh OSX JNI Example	475
Step 1: Create Java Source File.....	476
Step 2: Compile Java Source File	476
Step 3: Create Header File	476
Step 4: Create C++ Source File.....	476
Step 5: Compile C++ Source File to Create Dynamic Link Library.....	476
Step 6: Run Java Program	477
When To Use JNI	477
Quick Review	477
Summary	478
Skill Building Exercises	478
Suggested Projects	478
Self Test Questions	479
References	479
Notes	480

19 THREE DESIGN PRINCIPLES

Introduction	482
The Preferred Characteristics of an Object-Oriented Architecture	482
Easy to Understand – (How does this thing work?)	482
Easy to Reason About – (What are the effects of change?)	482
Easy to Extend – (Where do I add functionality?)	482
The Liskov Substitution Principle & Design by Contract	483
Reasoning About the Behavior of Supertypes and Subtypes	483
Relationship Between the LSP and DbC.....	483
The Common Goal of the LSP and DbC.....	483
C++ Support for the LSP and DbC.....	483
Designing with the LSP/DbC in Mind	483
The Power and Danger of C++.....	484
Class Declarations Viewed as Behavior Specifications	484
Preconditions, Postconditions, and Class Invariants	484
Class Invariant.....	484
Precondition.....	484
Postcondition.....	484
An Example	485
Using Incrementer as a Base Class	486
Changing the Preconditions of Derived Class Functions	488
Adopting the Same Preconditions	489
Weakening Preconditions	489
Strengthening Preconditions.....	491
Quick Review	493
Changing the Postconditions of Derived Class Functions	493
Special Cases of Preconditions and Postconditions	494
Function Argument Types	494
Function Return Types.....	496
Function Access Rights.....	496
Quick Review	497
Three Rules of the Substitution Principle	497
Signature Rule.....	497
Methods Rule.....	497

<i>Properties Rule</i>	497
The Open-Closed Principle	497
<i>Achieving The Open-Closed Principle</i>	498
<i>An OCP Example</i>	498
<i>Additional OCP Conventions</i>	498
<i>Relationship Between The OCP and The LSP/DbC</i>	498
<i>Quick Review</i>	498
The Dependency Inversion Principle	500
<i>Characteristics of Bad Software Architecture</i>	500
<i>Characteristics of Good Software Architecture</i>	501
<i>Selecting The Right Abstractions Takes Experience</i>	501
<i>Quick Review</i>	501
SUMMARY	501
TERMS AND DEFINITIONS	502
SKILL BUILDING EXERCISES	502
SUGGESTED PROJECTS	503
SELF TEST QUESTIONS	503
REFERENCES	503
NOTES	504

20 Using A UML Modeling Tool

INTRODUCTION	506
THE PURPOSE OF A UML Modeling Tool	506
INTRODUCING EMBARCADERO TECHNOLOGIES' DESCRIBE™	507
<i>Primary Features</i>	507
THE PROJECT SPECIFICATION: ROBOT RAT	508
CREATING USE CASE DIAGRAMS	509
<i>Adding Documentation to Diagram Elements</i>	511
<i>Programmer Perspective Use Cases</i>	512
PAUSING TO CONSIDER DESIGN ISSUES	513
CREATING CLASS DIAGRAMS	515
<i>Creating An Overall Package Architecture Diagram</i>	515
<i>Moving Beyond The Package Diagram</i>	516
<i>Adding Operations and Attributes to Classes</i>	517
ITERATING THROUGH THE DESIGN PROCESS	519
CREATING SEQUENCE DIAGRAMS	522
<i>Proper Use of Sequence Diagrams</i>	522
<i>Adding Objects to Sequence Diagrams</i>	522
<i>Adding Messages to Sequence Diagrams</i>	523
GENERATING SOURCE CODE	525
REVERSE ENGINEERING	527
<i>Merging Systems</i>	528
LINKING DIAGRAM OBJECTS TO DIAGRAMS	529
GENERATING WEB PROJECT REPORTS	530
SUMMARY	531
ROBOTRAT SOURCE CODE	532
<i>abstractposition.h</i>	532
<i>abstractmarker.h</i>	532
<i>abstractcontrolledobject.h</i>	532
<i>position.h</i>	533
<i>marker.h</i>	533
<i>remotecontrolledobject.h</i>	534
<i>abstractcontrolledrodent.h</i>	534
<i>robotrat.h</i>	534
<i>rodentworld.h</i>	535

<i>USERINTERFACE.H</i>	535
<i>CONTROLLER.H</i>	536
<i>POSITION.CPP</i>	536
<i>MARKER.CPP</i>	538
<i>REMOTECONTROLLEDOBJECT.CPP</i>	538
<i>ROBOTRAT.CPP</i>	539
<i>RODENTWORLD.CPP</i>	540
<i>USERINTERFACE.CPP</i>	542
<i>CONTROLLER.CPP</i>	543
<i>MAIN.CPP</i>	544
Skill Building Exercises	544
SUGGESTED PROJECTS	545
Self Test Questions	545
REFERENCES	546
NOTES	546

APPENDICES

Appendix A: Project Approach Strategy Checkoff List

PROJECT APPROACH STRATEGY CHECKOFF LIST	549
--	------------

Appendix B: ASCII Table

ASCII TABLE	551
--------------------------	------------

Appendix C: ANSWERS TO SELF TEST QUESTIONS

CHAPTER 1	555
CHAPTER 2	556
CHAPTER 3	557
CHAPTER 4	558
CHAPTER 5	560
CHAPTER 6	561
CHAPTER 7	563
CHAPTER 8	564
CHAPTER 9	565
CHAPTER 10	567
CHAPTER 11	568
CHAPTER 12	569
CHAPTER 13	569
CHAPTER 14	571
CHAPTER 15	571
CHAPTER 16	572
CHAPTER 17	574
CHAPTER 18	575
CHAPTER 19	576
CHAPTER 20	578

Tables

Table 1-1: Header File Contents	14
Table 1-2: What Not To Put In A Header File	15
Table 1-3: Good vs. Bad Variable Names	18
Table 1-4: Good vs. Bad Constant Naming	18
Table 1-5: Function Naming	18
Table 3-1: Project Approach Strategy	44
Table 3-2: Development Cycle	45
Table 3-3: Project Specification	46
Table 3-4: Robot Rat Nouns and Verbs	48
Table 3-5: Language Feature Study Checkoff List For Robot Rat Project	51
Table 3-6: First Iteration Feature Set	52
Table 3-7: Second Iteration Feature Set	56
Table 3-8: Third Iteration Feature Set	60
Table 3-9: Fourth Iteration Design Consideration and Design Decisions	64
Table 3-10: Design Considerations and Decisions: Fifth Iteration	69
Table 3-11: Things To Double-Check Before Handing In Project	72
Table 4-1: Trigraph Replacement	85
Table 4-2: Escape Sequences	86
Table 4-3: CodeWarrior Structure Alignment	91
Table 5-1: Fundamental Types and Their Value Ranges	104
Table 5-2: Simple Escape Sequences	109
Table 5-3: Expression Forms	112
Table 5-4: C++ Operators, Precedence, and Associativity	114
Table 5-5: Multiplicative Operators	116
Table 5-6: Additive Operators	117
Table 5-7: Shift Operators	118
Table 5-8: Relational Operators	120
Table 5-9: Equality Operators	121
Table 5-10: Assignment Operators	124
Table 5-11: Possible Hungarian Notation Prefixes	126
Table 6-1: Control Statement Usage Guide	160
Table 7-1: Pointers vs. References	178
Table 9-1: Characteristics of Well-Written Functions	208
Table 10-1: Differences Between Structures and Classes	265
Table 10-2: Object-Oriented Terminology	266
Table 14-1: Overloadable Operators	373
Table 15-1: STL Containers	406
Table 15-2: STL Container Adapters	407
Table 15-3: STL Algorithm Function Templates	408
Table 16-1: Language Features That Support Object-Oriented Programming	417
Table 16-2: Language Features vs. Inheritance Behavior	436
Table 17-1: Object Usage Contexts	447
Table 19-1: Terms and Definitions Related to the LSP	502
Table 20-1: Robot Rat Project Design Considerations	513
Table 20-2: Robot Rat Application Package Names and Their Purpose	516
Table 20-3: Robot Rat Project Classes	519
Table Appendix A-1: Project Approach Strategy Checkoff List	549
Table Appendix B-1: ASCII Table	551

FIGURES

FIGURE 2-1: THE PROGRAM CREATION PROCESS	26
FIGURE 2-2: CREATING A NEW PROJECT IN CODEWARRIOR	28
FIGURE 2-3: SELECTING STATIONERY AND NAMING PROJECT	28
FIGURE 2-4: SETTING A PROJECT'S LOCATION	29
FIGURE 2-5: SELECT PROJECT TYPE	29
FIGURE 2-6: FIRSTCLASS PROJECT WINDOW	30
FIGURE 2-7: SOURCES GROUP OPEN REVEALING HelloWorld.cp	30
FIGURE 2-8: CREATING NEW TEXT FILE	30
FIGURE 2-9: EDITING firstclass.h	31
FIGURE 2-10: ADDING FILES TO PROJECT	31
FIGURE 2-11: firstclass.cpp AND main.cpp ADDED, HelloWorld.cp REMOVED	31
FIGURE 2-12: FIRSTCLASS PROJECT OUTPUT	31
FIGURE 2-13: CREATING NEW VISUAL C++ PROJECT	32
FIGURE 2-14: NAMING THE PROJECT	32
FIGURE 2-15: SELECTING CONSOLE APPLICATION TYPE	33
FIGURE 2-16: NEW PROJECT INFORMATION WINDOW	33
FIGURE 2-17: WORKSPACE ENVIRONMENT WITH CLASSVIEW SELECTED	33
FIGURE 2-18: EDITED PROJECT1.cpp FILE	34
FIGURE 2-19: ADDING NEW C++ HEADER FILE TO PROJECT 1	34
FIGURE 2-20: FILE NAME ENTERED	34
FIGURE 2-21: EDITING firstclass.h	35
FIGURE 2-22: CREATING A NEW C++ SOURCE FILE	35
FIGURE 2-23: LINKING...MESSAGE AND RESULTS OF BUILDING PROJECT 1	36
FIGURE 2-24: RUNNING PROJECT1.EXE	36
FIGURE 2-25: CREATING firstclass.h WITH EMACS	37
FIGURE 2-26: CREATING MAKEFILE WITH EMACS	37
FIGURE 2-27: RUNNING THE MAKE UTILITY	38
FIGURE 2-28: RESULTS OF EXECUTING MAKE UTILITY AND firstprog	39
FIGURE 3-1: TIGHT SPIRAL DEVELOPMENT CYCLE DEPLOYMENT	45
FIGURE 3-2: ROBOT RAT VIEWED AS ATTRIBUTES	49
FIGURE 3-3: ROBOT RAT FLOOR SKETCH	49
FIGURE 3-4: COMPLETE ROBOT RAT ATTRIBUTES	50
FIGURE 3-5: FUNCTIONAL DECOMPOSITION OF ROBOT RAT PROGRAM	52
FIGURE 3-6: OVERVIEW OF PROJECT CREATION PROCESS	53
FIGURE 3-7: ROBOTRAT.H	54
FIGURE 3-8: ROBOTRAT.CPP	54
FIGURE 3-9: MAIN.CPP	55
FIGURE 3-10: ROBOT RAT MENU	55
FIGURE 3-11: ROBOTRAT.H	57
FIGURE 3-12: MAIN.CPP	59
FIGURE 3-13: TEST RESULTS	59
FIGURE 3-14: DEFAULT CASE TEST	59
FIGURE 3-15: ROBOTRAT.CPP WITH FLOOR ARRAY DECLARATION	61
FIGURE 3-16: ROBOTRAT.H WITH ROWS & COLS CONSTANTS DECLARED	61
FIGURE 3-17: THE printFloor() FUNCTION	62
FIGURE 3-18: ROBOT RAT printFloor() FUNCTION TEST	62
FIGURE 3-19: setTestPattern() FUNCTION	63
FIGURE 3-20: setTestPattern() FUNCTION BEING USED FOR TESTING IN THE printFloor() FUNCTION.	63
FIGURE 3-21: ROBOT RAT printFloor() TEST WITH TEST PATTERN	64

FIGURE 3-22: STATE TRANSITION DIAGRAM FOR rats_direction VARIABLE	65
FIGURE 3-23: STATE TRANSITION DIAGRAM FOR pen_position	66
FIGURE 3-24: DIRECTION AND PenPosition ENUM Types Added to robotrat.h	66
FIGURE 3-25: DECLARATION OF pen_position & rats_position	66
FIGURE 3-26: setPenUp() & setPenDown() FUNCTIONS	67
FIGURE 3-27: turnRight() FUNCTION	67
FIGURE 3-28: turnLeft() FUNCTION	67
FIGURE 3-29: turnLeft() FUNCTION WITH cout STATEMENTS	68
FIGURE 3-30: turnLeft() TEST	68
FIGURE 3-31: move() FUNCTION, Top Half	71
FIGURE 3-32: move() FUNCTION TEST	71
FIGURE 4-1: Typical Power Mac G4 System	80
FIGURE 4-2: SYSTEM UNIT	81
FIGURE 4-3: MAIN Logic BOARD Block Diagram	81
FIGURE 4-4: POWERPC G4 PROCESSOR	82
FIGURE 4-5: MOTOROLA POWERPC 7400 Block Diagram	82
FIGURE 4-6: C++ TRANSLATION PHASES	87
FIGURE 4-7: PROCESSING Cycle	88
FIGURE 4-8: MEMORY Hierarchy	89
FIGURE 4-9: Simplified MEMORY Subsystem Diagram	89
FIGURE 4-10: Simplified MAIN MEMORY Diagram	90
FIGURE 4-11: CodeWARRIOR Code GENERATION SETTINGS Window	91
FIGURE 4-12: Dumb SORT RESULTS 1	93
FIGURE 4-13: Dumb SORT RESULTS 2	93
FIGURE 4-14: Dumb SORT RESULTS 3	93
FIGURE 4-15: ALGORITHMIC Growth RATES	93
FIGURE 5-1. SELECTING Std C++ CONSOLE SETTINGS	101
FIGURE 5-2. PPC Std C++ CONSOLE SETTINGS Dialog	101
FIGURE 5-3. Minimal_PROGRAM Project Window	102
FIGURE 5-4. SELECTING Disassemble FROM THE PROJECT MENU	102
FIGURE 5-5: RESULTS OF RUNNING Example 5.3	106
FIGURE 5-6: INTEGER VALUE OF CHARACTER LITERAL 'Help'	109
FIGURE 5-7: PARTS OF A FLOATING POINT LITERAL	110
FIGURE 5-8: Left Shifting shift_val	119
FIGURE 5-9: Right Shifting shift_val	120
FIGURE 5-10: AND Truth Table	122
FIGURE 5-11: Exclusive OR Truth Table	122
FIGURE 5-12: Inclusive OR Truth Table	122
FIGURE 5-13: Conditional OPERATOR Map	123
FIGURE 5-14: Assignment OPERATOR OPERANDS	124
FIGURE 5-15: CREATING LOCAL SCOPE Blocks with BRACES	129
FIGURE 6-1: if STATEMENT Diagram	143
FIGURE 6-2: ifElse STATEMENT Diagram	145
FIGURE 6-3: switch STATEMENT Diagram	148
FIGURE 6-4: while STATEMENT Diagram	150
FIGURE 6-5: do STATEMENT Diagram	155
FIGURE 6-6: for STATEMENT Diagram	156
FIGURE 6-7: goto STATEMENT Diagram	159
FIGURE 7-1: MEMORY	168
FIGURE 7-2: ANOTHER Way TO REPRESENT MEMORY	169
FIGURE 7-3: Hexadecimal Addressing	169
FIGURE 7-4: RUNNING Example 7.2 with CodeWARRIOR	170
FIGURE 7-5: RUNNING Example 7.2 ALONE	171
FIGURE 7-6: POINTER	171
FIGURE 7-7: CONTENTS of int_ptr	172
FIGURE 7-8: RUNNING Example 7.5	174
FIGURE 7-9: Application Stack and Heap Relationship	175
FIGURE 8-1: ARRAY of FOUR INTEGER OBJECTS	184
FIGURE 8-2: RESULTS of RUNNING Example 8.5	187

FIGURE 8-3: RESULTS OF RUNNING EXAMPLE 8.6188

FIGURE 8-4: RESULTS OF RUNNING EXAMPLE 8.7188

FIGURE 8-5: RESULTS OF RUNNING EXAMPLE 8.8188

FIGURE 8-6: ARRAY OF INTEGER POINTERS AND DYNAMICALLY CREATED INTEGER OBJECTS IN HEAP MEMORY188

FIGURE 8-7: SINGLE-DIMENSIONAL ARRAY REPRESENTATION AND DECLARATION189

FIGURE 8-8: TWO-DIMENSIONAL ARRAY AND DECLARATION190

FIGURE 8-9: TWO-DIMENSION ARRAY MEMORY REPRESENTATION191

FIGURE 8-10: VISUAL REPRESENTATION OF A THREE DIMENSIONAL ARRAY192

FIGURE 8-11: VISUAL REPRESENTATION OF A FOUR-DIMENSIONAL ARRAY192

FIGURE 8-12: THREE_D_INT_ARRAY INITIALIZED TO ZEROS193

FIGURE 8-13: ALL ROWS OF FIRST SHEET INITIALIZED194

FIGURE 8-15: RELATIONSHIP OF DECLARATION BRACES TO ARRAY ELEMENTS FOR THREE_D_INT_ARRAY194

FIGURE 8-14: FIRST ROW OF EACH SHEET INITIALIZED195

FIGURE 8-16: RESULTS OF INITIALIZATION SHOWN IN EXAMPLE 8.14195

FIGURE 8-17: DYNAMIC ARRAY OF THREE INTEGER POINTERS196

FIGURE 8-18: RESULTS OF RUNNING EXAMPLE 8.17 USING ROW VALUE 6198

FIGURE 8-19: RESULTS OF RUNNING EXAMPLE 8.19 USING ROWS = 10 & COLS = 6199

FIGURE 9-1: TESTFUNCTIONONE PROJECT SCREEN SHOT211

FIGURE 9-2: RESULTS OF CALLING TESTFUNCTIONONE()211

FIGURE 9-3: RESULTS OF CALLING TESTFUNCTIONTWO() FIVE TIMES WITH STATIC VARIABLE214

FIGURE 9-4: RESULTS OF CALLING TESTFUNCTIONTHREE() WITH AN ARGUMENT VALUE OF 5215

FIGURE 9-5: FUNCTION ACTIVATION RECORD SEQUENCE215

FIGURE 9-6: PARTIAL DISASSEMBLY OF main.cpp216

FIGURE 9-7: PARTIAL DISASSEMBLY OF TESTFUNCTIONTHREE.cpp217

FIGURE 9-8: RESULTS OF RUNNING TESTFUNCTIONFOUR PROGRAM219

FIGURE 9-9: RESULTS OF RUNNING addressCopyTEST PROGRAM220

FIGURE 9-10: RESULTS OF RUNNING TESTFUNCTIONFIVE PROGRAM221

FIGURE 9-11 RESULTS OF RUNNING TESTFUNCTIONSIX PROGRAM222

FIGURE 9-12: RESULTS OF RUNNING EXAMPLE 9.28225

FIGURE 9-13: RESULTS OF CALLING OVERLOADED FUNCTION FUNCTIONA()232

FIGURE 9-14: RESULTS OF RUNNING THE SIMPLE RECURSE PROGRAM233

FIGURE 9-15: RESULTS OF RUNNING THE QUICKSORT PROGRAM235

FIGURE 9-16: RESULTS OF CALLING DUMBSORT() USING COMPAREASCENDING() AND COMPAREDESCENDING() CALLBACK FUNCTIONS239

FIGURE 9-17: CREATING AN EMPTY PROJECT IN CODEWARRIOR240

FIGURE 9-18: DUMBSORT.cpp ADDED TO THE EMPTY PROJECT240

FIGURE 9-19: SETTING LIBRARY TARGET SETTINGS240

FIGURE 9-20: SELECTING PROJECT TYPE AND LIBRARY NAME241

FIGURE 9-21: USING THE DUMBSORT LIBRARY241

FIGURE 10-1: EXAMPLE 10.9 OUTPUT255

FIGURE 10-2: C++ LANGUAGE SETTINGS: SET ENUMS ALWAYS INT256

FIGURE 10-3: FORMAT OF STRUCTURE FUNCTION DEFINITION262

FIGURE 11-1: UML REPRESENTATION FOR THE CLASS CLASSNAME276

FIGURE 11-2: UML CLASS DIAGRAM OF A SIMPLE NAVY FLEET SIMULATION APPLICATION277

FIGURE 11-3: RESULTS OF RUNNING EXAMPLE 11.4281

FIGURE 11-4: RESULTS OF RUNNING EXAMPLE 11.7282

FIGURE 11-5: RESULTS OF RUNNING EXAMPLE 11.10284

FIGURE 11-6: RESULTS OF RUNNING EXAMPLE 10.10 AGAIN286

FIGURE 11-7: RESULTS OF RUNNING EXAMPLE 11.15287

FIGURE 11-8: HORIZONTAL ACCESS289

FIGURE 11-9: RESULTS OF RUNNING EXAMPLE 11.18293

FIGURE 11-10: PERSON CLASS DIAGRAM294

FIGURE 12-1: RESULTS OF RUNNING EXAMPLE 12.5304

FIGURE 12-2: RESULTS OF RUNNING EXAMPLE 12.5 AGAIN305

FIGURE 12-3: UML DIAGRAM ILLUSTRATING SIMPLE AGGREGATION308

FIGURE 12-4: UML SEQUENCE DIAGRAM ILLUSTRATING MESSAGE PASSING BETWEEN OBJECTS308

FIGURE 12-6: FUELPUMP CLASS309

FIGURE 12-5: ENGINE COMPOSITE AGGREGATION CLASS DIAGRAM310

FIGURE 12-7: ENGINE CLASS DIAGRAM311

FIGURE 12-8: RESULTS OF RUNNING EXAMPLE 12-16313

FIGURE 13-1: UML CLASS DIAGRAM SHOWING GENERALIZATION	329
FIGURE 13-2: RESULTS OF RUNNING EXAMPLE 13.5	332
FIGURE 13-3: EFFECTS OF USING DIFFERENT INHERITANCE SPECIFIERS	332
FIGURE 13-4: PUBLIC, PROTECTED, & PRIVATE INHERITANCE	334
FIGURE 13-5: PUBLIC INHERITANCE FROM A HORIZONTAL ACCESS PERSPECTIVE	334
FIGURE 13-6: PERSON/STUDENT CLASS DIAGRAM	335
FIGURE 13-7: RESULTS OF RUNNING EXAMPLE 13.9	337
FIGURE 13-8: FOO AND DERIVEDFOO CLASS DIAGRAM	338
FIGURE 13-9: RESULTS OF RUNNING EXAMPLE 13.14	339
FIGURE 13-10: RESULTS OF RUNNING EXAMPLE 13.14 AFTER MODIFYING foo.cpp	341
FIGURE 13-11: RESULTS OF RUNNING EXAMPLE 13.14 AFTER REMOVING THE VIRTUAL KEYWORD FROM FOO CLASS DESTRUCTOR DECLARATION	342
FIGURE 13-12: FLEET SIMULATION CLASS DIAGRAM	344
FIGURE 13-13: RESULTS OF RUNNING EXAMPLE 13.20	345
FIGURE 13-14: PAYROLL APPLICATION CLASS DIAGRAM	354
FIGURE 13-15: RESULTS OF RUNNING EXAMPLE 13.28	356
FIGURE 13-16: CLASS DIAGRAM SHOWING COMMON BASE CLASS INHERITANCE	358
FIGURE 13-17: NON-VIRTUAL INHERITANCE WILL RESULT IN MULTIPLE INSTANCES OF BASE CLASSES	358
FIGURE 13-18: RESULTS OF RUNNING EXAMPLE 13.33.	360
FIGURE 13-19: RESULTS OF RUNNING 13.33 SHOWING EFFECTS OF VIRTUAL INHERITANCE	361
FIGURE 13-20: VIRTUAL INHERITANCE RESULTS IN ONE INSTANCE OF A	361
FIGURE 14-1: I/O STREAM CLASS HIERARCHY	375
FIGURE 14-2: RESULTS OF RUNNING EXAMPLE 14.3	376
FIGURE 14-3: RESULTS OF RUNNING EXAMPLE 14.5	378
FIGURE 14-4a: BEFORE SHALLOW COPY OF COMPLEX OBJECTS	379
FIGURE 14-4b: AFTER SHALLOW COPY OF COMPLEX OBJECTS	379
FIGURE 14-5: RESULTS OF RUNNING EXAMPLE 14.9	381
FIGURE 14-6: RESULTS OF RUNNING EXAMPLE 14.12	383
FIGURE 14-7: RESULTS OF RUNNING EXAMPLE 14.15	384
FIGURE 14-8: RESULTS OF RUNNING EXAMPLE 14.18	386
FIGURE 14-9: RESULTS OF RUNNING EXAMPLE 14.21	387
FIGURE 14-10: RESULTS OF RUNNING EXAMPLE 14.24	389
FIGURE 14-11: RESULTS OF RUNNING EXAMPLE 14.27	392
FIGURE 14-12: RESULTS OF RUNNING EXAMPLE 14.32	393
FIGURE 15-1: PLACEHOLDER USE IN MAIL MERGE	399
FIGURE 15-2: RESULTS OF RUNNING EXAMPLE 15.2	400
FIGURE 15-3: ERROR RESULTING FROM CALLING SUM() WITH TWO DIFFERENT TYPE ARGUMENTS	400
FIGURE 15-4: RESULTS OF RUNNING EXAMPLE 15.4	401
FIGURE 15-5: RESULTS OF RUNNING EXAMPLE 15.5	402
FIGURE 15-6: RESULTS OF RUNNING EXAMPLE 15.9	404
FIGURE 15-7: RESULTS OF RUNNING EXAMPLE 15.11	405
FIGURE 15-8: RESULTS OF RUNNING EXAMPLE 15.13	409
FIGURE 15-9: RESULTS OF RUNNING EXAMPLE 15.14	410
FIGURE 15-10: RESULTS OF RUNNING EXAMPLE 15.15	410
FIGURE 15-11: RESULTS OF RUNNING EXAMPLE 15.16	411
FIGURE 16-1: BASE CLASS DECLARES BEHAVIOR SHARED BY ALL DERIVED CLASS OBJECTS	416
FIGURE 16-2: CLASS DIAGRAM SHOWING THREE-LEVEL INHERITANCE HIERARCHY	419
FIGURE 16-3: RESULTS OF RUNNING EXAMPLE 16.4	421
FIGURE 16-4: ORIGINAL AIRCRAFT ENGINE COMPONENTS MODEL	422
FIGURE 16-5: UML CLASS DIAGRAM SHOWING POLYMORPHIC ENGINE COMPONENTS	423
FIGURE 16-6: RESULTS OF RUNNING POLYMORPHIC ENGINE PROGRAM	435
FIGURE 17-1: RESULTS OF RUNNING EXAMPLE 17.3	451
FIGURE 17-2: RESULTS OF RUNNING EXAMPLE 17.9	452
FIGURE 18-1: CREATING A NEW EMPTY PROJECT NAMED SQUARE_LIB IN CODEWARRIOR	459
FIGURE 18-2: EMPTY PROJECT WINDOW	460
FIGURE 18-3: SELECT ADD FILES... FROM THE PROJECT MENU	460
FIGURE 18-4: SELECT SQUARE.C TO ADD IT TO THE PROJECT	460
FIGURE 18-5: PROJECT WINDOW AFTER ADDING SQUARE.C	461
FIGURE 18-6: SELECT SQUARE_LIB SETTINGS...FROM THE EDIT MENU	461
FIGURE 18-7: SETTINGS WINDOW WITH TARGET SETTINGS SELECTED	461

FIGURE 18-8: SETTING PROJECT TYPE AND LIBRARY FILE NAME462

FIGURE 18-9: ENSURE THE ACTIVATE C++ COMPILER CHECK BOX IS NOT CHECKED462

FIGURE 18-10: SELECT MAKE FROM THE PROJECT MENU TO CREATE THE SQUARE.lib FILE463

FIGURE 18-11: C++ PROJECT WINDOW WITH SQUARE.lib LIBRARY FILE ADDED.463

FIGURE 18-12: LINK ERROR RESULTING FROM FIRST ATTEMPT TO BUILD THE C++ PROJECT THAT USES A C FUNCTION464

FIGURE 18-13: RESULTS OF RUNNING THE C++ PROJECT USING THE C SQUARE() FUNCTION464

FIGURE 18-14: WIN32 PROJECT USING INLINE ASSEMBLY LANGUAGE466

FIGURE 18-15: RESULTS OF RUNNING THE INLINE ASSEMBLY PROJECT466

FIGURE 18-16: ADDING ASSEMBLY OBJECT FILE TO C++ PROJECT467

FIGURE 18-17: ASSEMBLING double.ASM WITH MASM VER. 6.14468

FIGURE 18-18: WIN32 PROJECT USING dv.obj469

FIGURE 18-19: RESULTS OF RUNNING MACINTOSH VERSION OF doubleVal()470

FIGURE 18-20: STEPS TO CREATE A JAVA NATIVE INTERFACE (JNI) PROGRAM471

FIGURE 18-21: COMPILING SayHi.java472

FIGURE 18-22: COMPILING SayHi.java RESULTS IN SayHi.class472

FIGURE 18-23: USING javah TO CREATE THE SayHi.h HEADER FILE473

FIGURE 18-24: RESULTS OF CREATING SayHi.h USING javah COMMAND LINE TOOL473

FIGURE 18-25: BLANK CODEWARRIOR PROJECT474

FIGURE 18-26: sayhi.cpp ADDED TO BLANK PROJECT474

FIGURE 18-27: BLANK PROJECT WINDOW SHOWING ADDED LIBRARY FILES475

FIGURE 18-28: TARGET SETTINGS WINDOW475

FIGURE 18-29: DIRECTORY LISTING SHOWING SayHi.dll475

FIGURE 18-30: RESULTS OF RUNNING THE SayHi JAVA APPLICATION476

FIGURE 18-31: COMPILING sayhi.cpp USING g++ TO GENERATE AN OSX DYNAMIC LINK LIBRARY477

FIGURE 18-32: DIRECTORY LISTING SHOWING libSayHi.jnilib FILE477

FIGURE 18-33: RESULTS OF RUNNING SayHi JAVA PROGRAM IN AN OSX TERMINAL WINDOW477

FIGURE 19-1: RESULTS OF RUNNING EXAMPLE 19.3486

FIGURE 19-2: RESULTS OF RUNNING EXAMPLE 19.4487

FIGURE 19-3: RESULTS OF RUNNING EXAMPLE 19.7489

FIGURE 19-4: RESULTS OF RUNNING EXAMPLE 19.13493

FIGURE 19-5: INHERITANCE HIERARCHY SHOWING WEAKER AND STRONGER TYPES494

FIGURE 19-6: RESULTS OF RUNNING EXAMPLE 19.14495

FIGURE 19-7: RESULTS OF RUNNING EXAMPLE 19.14 WITH MODIFIED C CLASS FUNCTION495

FIGURE 19-8: RESULTS OF RUNNING EXAMPLE 19.14 USING PRIVATE C::f() OVERRIDING FUNCTION496

FIGURE 19-9: FLEET SIMULATION MODEL CLASS DIAGRAM499

FIGURE 19-10: PROCEDURE-ORIENTED SOFTWARE MODULE HIERARCHY500

FIGURE 20-1: DESCRIBE USER MODES507

FIGURE 20-2: ROBOT RAT PROJECT SPECIFICATION508

FIGURE 20-3: CREATING THE NEW ROBOTRAT SYSTEM509

FIGURE 20-4: ADDING A NEW DIAGRAM509

FIGURE 20-5: CREATING USE CASE DIAGRAM510

FIGURE 20-6: USER PERSPECTIVE USE CASES510

FIGURE 20-7: ADDING DOCUMENTATION VIA THE PROPERTIES EDITOR WINDOW511

FIGURE 20-8: LINKING TO EXTERNAL DOCUMENTATION VIA THE PROPERTIES EDITOR WINDOW511

FIGURE 20-9: COMPLETED ROBOT RAT USER'S PERSPECTIVE USE CASE DIAGRAM512

FIGURE 20-10: PROGRAMMER PERSPECTIVE USE CASES513

FIGURE 20-11: PARTIAL APPLICATION ARCHITECTURE USE CASE DIAGRAM515

FIGURE 20-12: OVERALL ROBOT RAT APPLICATION PACKAGE ARCHITECTURE516

FIGURE 20-13: CLASS DIAGRAM SHOWING REMOTE CONTROLLED OBJECT PACKAGE CLASSES517

FIGURE 20-14: PROPERTIES EDITOR FOR ABSTRACTPOSITION CLASS517

FIGURE 20-15: PROPERTIES EDITOR WINDOW FOR THE ABSTRACTPOSITION OPERATION518

FIGURE 20-16: PROPERTIES EDITOR WINDOW FOR THE setRow() FUNCTION518

FIGURE 20-17: ADDING OPERATION PARAMETERS USING THE PROPERTIES EDITOR WINDOW519

FIGURE 20-18: COMPLETED AND ANNOTATED OVERALL CLASS DIAGRAM522

FIGURE 20-19: START OF SEQUENCE DIAGRAM FOR ROBOT RAT APPLICATION LAUNCH523

FIGURE 20-20: EDITING CONTROLLER() MESSAGE PROPERTIES524

FIGURE 20-21: COMPLETED ROBOT RAT APPLICATION LAUNCH SEQUENCE524

FIGURE 20-22: CREATE NEW ROBOTRAT SEQUENCE DIAGRAM525

FIGURE 20-23: FIRST STEP TO GENERATING CODE: SELECT CLASS DIAGRAMS526

Figures

FIGURE 20-24: GENERATE CODE MENU ITEM	526
FIGURE 20-25: CODE GENERATION DIALOG	527
FIGURE 20-26: REVERSE ENGINEERING DIALOG	527
FIGURE 20-27: STEP 2 IN THE REVERSE ENGINEERING PROCESS: NAMING THE NEW SYSTEM AND SETTING VARIOUS SYSTEM PROPERTIES	528
FIGURE 20-28: REPROGRESS WINDOW	528
FIGURE 20-29: MERGE SYSTEM DIALOG	529
FIGURE 20-30: ASSOCIATING DIAGRAM OBJECT WITH SYSTEM DIAGRAMS	529
FIGURE 20-31: NAVIGATING TO LINKED DIAGRAM	530
FIGURE 20-32: Web VIEWER WIZARD	530
FIGURE 20-33: SELECTING SYSTEM FOR Web REPORT GENERATION	531
FIGURE 20-34: MAIN SCREEN - ROBOTRAT PROJECT Web VIEW	531

Code Examples

1.1 test.h	13
1.2 firstclass.h	14
1.3 firstclass.cpp	16
1.4 main.cpp	16
1.5 C-style COMMENTS	16
1.6 C-style COMMENTS	17
1.7 C++-style COMMENT	17
1.8 C++ COMMENT CLUTTER	17
2.1 firstclass.h	27
2.2 firstclass.cpp	27
2.3 main.cpp	28
2.4 makefile	38
3.1 Robot RAT Pseudocode	51
3.2 Pseudocode FOR PROCESSING USER MENU CHOICES	57
3.3 robotrat.cpp	57
3.3 robotrat.cpp CONTINUED	58
3.4 move() FUNCTION pseudocode	69
3.5 NORTH move pseudocode PEN IN THE UP position	70
3.6 NORTH move pseudocode PEN IN DOWN position	70
3.7 COMPLETE Robot RAT SOURCE CODE LISTING	72
4.1 Dumb SORT TEST PROGRAM	92
5.1 DISASSEMBLED Minimal main() FUNCTION	100
5.2 ANOTHER C++ PROGRAM	102
5.3 Using NUMERIC_limits TEMPLATE CLASS TO CALCULATE Type RANGES	105
5.4 Using THE sizeof OPERATOR	106
5.5 SOURCE CODE SHOWING LOCAL, FUNCTION, AND FILE SCOPING	130
file1.cpp	131
file2.cpp	131
5.6 file scope linkage	131
file1.cpp	132
file2.cpp	132
5.7 static linkage	132
5.8 REGISTERING FUNCTIONS WITH atexit()	133
5.9 Using cin OBJECT TO READ INTEGER VALUES FROM KEYBOARD	134
5.10 TESTING FOR Valid Input	134
6.1 if STATEMENT	144
6.2 ASSIGNMENT	144
6.3 EQUALITY	144
6.4 DECLARATION IN CONDITION	144
6.5 COMPOUND STATEMENTS	145
6.6 if-else	145
6.7 ! OPERATOR	145
6.8 COMPOUND STATEMENTS WITH if-else	146
6.9 NESTING if-else	146
6.10 USE OF SELECTION STATEMENTS	147
6.11 SWITCH STATEMENT	148
6.12 WHILE STATEMENT	151
6.13 WHILE STATEMENT	151
6.14 USE OF SENTINEL VALUE	151

6.15 NESTED WHILE STATEMENTS	152
6.16 LOOPING FOREVER	152
6.17 LOOPING FOREVER	152
6.18 NESTED WHILE LOOP	153
6.19 SWITCH INSIDE OF WHILE LOOP	153
6.21 WHILE LOOP BEHAVING LIKE FOR LOOP	156
6.22 FOR STATEMENT	156
6.23 FOR LOOP SCOPE	156
6.24 IMPLEMENTING SUMMATION	157
6.25 NESTING FOR STATEMENTS	157
6.26 BREAK STATEMENT	157
6.27 LOOPING FOREVER	158
6.28 CONTINUE STATEMENT	158
6.29 LABELED STATEMENT	159
6.30 GOTOLESS CODE	159
7.1 INTEGER OBJECTS	168
7.2 USING & OPERATOR	170
7.3 USING * OPERATOR	172
7.4 DEREFERENCING POINTERS	173
7.5 DEREFERENCING POINTERS	173
7.6 DYNAMIC MEMORY ALLOCATION	176
7.7 CALLING OBJECT CONSTRUCTOR	177
7.8 CALLING OBJECT CONSTRUCTOR	177
7.9 USING A REFERENCE	178
8.1 DECLARING & USING INTEGER ARRAY	186
8.2 USING INTEGER ARRAY	186
8.3 MANIPULATING ARRAY WITH FOR STATEMENT	186
8.4 POINTER ARITHMETIC	187
8.5 GARBAGE OUT	187
8.6 GOOD OUTPUT	188
8.7 UNINITIALIZED POINTERS	188
8.8 GOOD OUTPUT	188
8.9 POINTER NULL INITIALIZATION	189
8.10 USING DELETE OPERATOR ON POINTER ARRAY ELEMENTS	189
8.11 USING 3-DIMENSIONAL ARRAY	193
8.12 INITIALIZING 3-DIMENSIONAL ARRAY	194
8.13 BRACE USAGE	194
8.14 BRACE USAGE	195
8.15 DYNAMIC ARRAY ALLOCATION	197
8.16 DYNAMIC ARRAY ALLOCATION	197
8.17 DYNAMIC MULTI-DIMENSIONAL ARRAY ALLOCATION	198
8.19 DYNAMICALLY ALLOCATING 2-DIMENSIONAL ARRAY	199
9.1 TESTFUNCTIONONE.H	210
9.2 TESTFUNCTIONONE.CPP	210
9.3 MAIN.CPP	211
9.4 LOCAL FUNCTION VARIABLES	212
9.5 MASKING GLOBAL VARIABLES	212
9.6 BLOCK SCOPE	213
9.7 SCOPE OF VARIABLES IN LOOPING STATEMENTS	213
9.8 STATIC FUNCTION VARIABLES	214
9.9 MASKING FUNCTION PARAMETERS	214
9.10 FUNCTION CALL WITH ARGUMENT	214
9.11 TESTFUNCTIONFOUR.H	218
9.12 TESTFUNCTIONFOUR.CPP	218
9.13 MAIN.CPP	218
9.14 PASSING ADDRESSES BY COPY	219
9.15 TESTFUNCTIONFIVE.CPP	220
9.16 MAIN.CPP	221
9.17 MAIN.CPP	221

9.18	testfunctionsix.cpp	222
9.19	main.cpp	222
9.20	printIntArray()	222
9.21	printIntArray.cpp	223
9.22	main.cpp	223
9.23	print_2d_int_array.h	223
9.24	print_2d_int_array.cpp	224
9.25	main.cpp	224
9.26	sort_int_array.h	224
9.27	sort_int_array.cpp	225
9.28	main.cpp	225
9.29	returnInt.cpp	226
9.30	main.cpp	226
9.31	square.h	227
9.32	square.cpp	227
9.34	getNewIntAddress.h	227
9.33	main.cpp	228
9.35	getNewIntAddress.cpp	228
9.36	main.cpp	229
9.37	getLargestInteger.h	230
9.38	getLargestInteger.cpp	230
9.39	main.cpp	230
9.40	functionA.h	231
9.41	functionA.cpp	231
9.42	main.cpp	232
9.43	countInput.h	232
9.44	countInput.cpp	233
9.45	main.cpp	233
9.46	quicksort.h	234
9.47	quicksort.cpp	234
9.48	main.cpp	235
9.49	arithFunctions.h	236
9.50	arithFunctions.cpp	236
9.51	main.cpp	237
9.52	dumbSort.h	237
9.53	dumbSort.cpp	238
9.54	main.cpp	238
10.1	mydefs.h	251
10.2	calculatePay.h	251
10.3	calculatePay.cpp	251
10.4	main.cpp	251
10.5	switch STATEMENT	252
10.6	NAMESPACES	253
10.7	switch STATEMENT	254
10.8	PERSONSTRUCT.H	254
10.9	ACCESSING STRUCT ELEMENTS	255
10.10	ACCESSING STRUCT ELEMENTS VIA POINTERS	256
10.11	PERSONFUNCTIONS.H	257
10.12	PERSONFUNCTIONS.CPP	257
10.12	CONTINUED	258
10.13	main.cpp	259
10.14	PERSONSTRUCT.H	260
10.15	PERSONSTRUCT.CPP	261
10.16	main.cpp TESTING PERSON STRUCT	263
10.17	PERSONCLASS.H	263
10.18	PERSONCLASS.CPP	264
10.19	main.cpp TESTING PERSON CLASS	265
11.1	PARTS OF A TYPICAL CLASS DECLARATION	275
11.2	TESTCLASS.H	279

11.3 TESTCLASS.CPP	280
11.4 MAIN.CPP	280
11.5 TESTCLASS.H	281
11.6 TESTCLASS.CPP	282
11.7 MAIN.CPP	282
11.8 TESTCLASS.H	283
11.9 TESTCLASS.CPP	284
11.10 MAIN.CPP	284
11.11 TESTCLASS.H	285
11.12 TESTCLASS.CPP	285
11.13 SIMPLECLASS.H	286
11.14 SIMPLECLASS.CPP	286
11.15 MAIN.CPP	287
11.16 FOO.H	291
11.18 MAIN.CPP	291
11.17 FOO.CPP	292
11.19 PERSON.H	294
11.21 PERSON.CPP	295
11.21 PERSON.CPP CONTINUED	296
11.20 MAIN.CPP	297
12.1 A.H	303
12.2 A.CPP	303
12.3 B.H	303
12.4 B.CPP	304
12.5 MAIN.CPP	304
12.6 B.H	305
12.7 B.CPP	305
12.8 A.H	306
12.9 A.CPP	306
12.10 B.H	306
12.11 B.CPP	307
12.12 MAIN.CPP	307
12.13 AIRCRAFTUTILS.H	309
12.14 FUELPUMP.H	311
12.15 ENGINE.H	312
12.16 MAIN.CPP	313
AIRCRAFTUTILS.H	314
FUELPUMP.H	314
OILPUMP.H	315
TEMPERATURESENSOR.H	315
OXYGENSENSOR.H	315
COMPRESSOR.H	316
ENGINE.H	316
FUELPUMP.CPP	317
OILPUMP.CPP	317
TEMPERATURESENSOR.CPP	318
OXYGENSENSOR.CPP	318
COMPRESSOR.CPP	319
ENGINE.CPP	320
MAIN.CPP	321
13.1 BASECLASS.H	329
13.2 DERIVEDCLASSONE.H	330
13.3 BASECLASS.CPP	330
13.4 DERIVEDCLASSONE.CPP	331
13.5 MAIN.CPP	333
13.6 PERSON.H	336
13.7 STUDENT.H	336
13.8 STUDENT.CPP	336
13.9 MAIN.CPP	337

13.10 foo.h	338
13.11 derivedfoo.h	338
13.12 foo.cpp	338
13.13 derivedfoo.cpp	339
13.14 main.cpp	339
13.15 foo.h	341
13.16 foo.h	341
13.17 vessel.h	343
13.18 plant.h	345
13.19 weapon.h	345
13.20 main.cpp	346
ciws.h	346
five_inch.h	347
torpedo.h	347
gasturbine_plant.h	347
nuke_plant.h	348
steam_plant.h	348
submarine.h	348
surface_ship.h	349
ciws.cpp	349
five_inch.cpp	349
gasturbine_plant.cpp	350
nuke_plant.cpp	350
steam_plant.cpp	351
submarine.cpp	351
surface_ship.cpp	352
torpedo.cpp	352
vessel.cpp	353
13.21 payable.h	354
13.22 employee.h	355
13.23 employee.cpp	355
13.24 hourlyemployee.h	355
13.25 salariedemployee.h	356
13.26 hourlyemployee.cpp	356
13.27 salariedemployee.cpp	357
13.28 main.cpp	357
13.29 a.h	359
13.30 b.h	359
13.31 c.h	359
13.32 d.h	359
13.33 main.cpp	359
13.34 b.h	360
13.35 c.h	360
14.1 modified person.h	375
14.2 overloaded stream operator implementation	376
14.3 main.cpp	377
14.4 foo.h	378
14.5 main.cpp	378
14.6 overloaded assignment operator implementation	380
14.8 overloaded relational operator implementation	380
14.9 main.cpp	380
14.7 person.h	381
14.10 extended person class	382
14.11 equality operator implementation	382
14.12 main.cpp	382
14.13 foo.h	383
14.14 foo.cpp	383
14.15 main.cpp	384
14.16 dynamicarray.h	385

14.17 dynamicarray.cpp	385
14.18 main.cpp	386
14.19 foo.h	386
14.20 foo.cpp	387
14.21 main.cpp	387
14.22 foo.h	388
14.23 foo.cpp	388
14.24 main.cpp	389
14.25 foo.h	390
14.27 main.cpp	390
14.26 foo.cpp	391
14.28 foo.h	392
14.29 bar.h	393
14.30 foo.cpp	393
14.31 bar.cpp	393
14.32 main.cpp	394
15.1 sumtemplate.h	399
15.2 main.cpp	400
15.3 sumtemplate.h	401
15.4 main.cpp	401
15.5 main.cpp	401
15.6 sumtemplate.h	402
15.7 main.cpp	402
15.8 foodef.h	403
15.9 main.cpp	403
15.10 dynamicarraydef.h	404
15.11 main.cpp	405
15.12 main.cpp	407
15.13 main.cpp	409
15.14 main.cpp	409
15.15 main.cpp	410
15.16 main.cpp	411
16.1 interface.h	419
16.2 derived_class_one.h	420
16.3 derived_class_two.h	420
16.4 main.cpp	421
icomponent.h	424
component.h	424
component.cpp	425
pump.h	425
pump.cpp	426
sensor.h	426
sensor.cpp	427
waterpump.h	427
waterpump.cpp	428
oilpump.h	428
oilpump.cpp	428
fuelpump.h	428
fuelpump.cpp	429
airflowsensor.h	429
airflowsensor.cpp	429
oxygensensor.h	429
oxygensensor.cpp	430
temperaturesensor.h	430
temperaturesensor.cpp	430
engine.h	431
engine.cpp	431
smallengine.h	432
smallengine.cpp	432

ENGINEUTILS.H	433
MAIN.CPP	433
17.1 foo.h	448
17.2 foo.cpp	449
17.3 main.cpp	450
17.4 f.h	450
17.5 f.cpp	450
17.6 dumsort.h (TEMPLATE VERSION)	451
17.7 foo.h (modified)	452
17.8 foo.cpp (modified)	452
17.9 main.cpp	453
18.1 SQUARE.H	459
18.2 SQUARE.C	459
18.3 main.cpp	463
18.4 modified SQUARE.H	464
18.5 double.h	465
18.6 double.cpp	466
18.7 double.ASM	468
18.8 main.cpp	468
18.9 double.h	469
18.10 double.cpp POWERPC VERSION	470
18.11 SayHi.java	472
18.12 SayHi.h	473
18.13 sayhi.cpp	474
19.1 INCREMENTER.H	485
19.3 main.cpp	485
19.2 INCREMENTER.CPP	486
19.4 main.cpp	487
19.5 derived.h	488
19.6 derived.cpp	488
19.7 main.cpp	489
19.8 derived.h (WEAKENED PRECONDITION)	490
19.10 main.cpp	490
19.9 derived.cpp (WEAKENED PRECONDITION)	491
19.11 derived.h (STRENGTHENED PRECONDITION)	492
19.12 derived.cpp (STRENGTHENED PRECONDITION)	492
19.13 main.cpp	493
19.14 main.cpp	495
19.15 c.h	496
ABSTRACTPOSITION.H	532
ABSTRACTMARKER.H	532
ABSTRACTCONTROLLEDOBJECT.H	532
POSITION.H	533
MARKER.H	533
REMOTECONTROLLEDOBJECT.H	534
ABSTRACTCONTROLLEDRODENT.H	534
ROBOTRAT.H	534
RODENTWORLD.H	535
USERINTERFACE.H	535
CONTROLLER.H	536
POSITION.CPP	536
MARKER.CPP	538
REMOTECONTROLLEDOBJECT.CPP	538
ROBOTRAT.CPP	539
RODENTWORLD.CPP	540
USERINTERFACE.CPP	542
CONTROLLER.CPP	543
MAIN.CPP	544

LEARNING OBJECTIVES

Identify and overcome the difficulties encountered by students when learning how to program

List and explain the software development roles played by students

List and explain the phases of the tight spiral software development methodology

Employ the concept of the flow to tap creative energy

List and explain the primary areas of the Project Approach Strategy

State the purpose of a header file

State the purpose of an implementation file

Explain the importance of separating interface from implementation

Employ multi-file programming techniques to tame project complexity

Explain the use of #ifndef, #define, and #endif preprocessor directives

Apply preprocessor directives to implement multi-file programming projects

State the importance of adopting consistent variable and constant naming conventions

List and describe the two types of C++ comments

List and describe the steps of the program creation process to include creating source code files, preprocessing, compiling, and linking

List the input and output to each stage of the program creation process

List and describe the primary functions of an Integrated Development Environment (IDE)

Describe the concept of a project

List and describe the steps required to create projects using Macintosh, Windows, and Unix development environments

Demonstrate your ability to create projects in the IDE of your choice

State the purpose of the Unix make utility

State the purpose of a Unix makefile

Demonstrate your ability to create and use Unix makefiles

Utilize Metrowerks CodeWarrior to create projects on Macintosh™ and PC platforms

Utilize Tenon Intersystems' CodeBuilder™ to create projects on the Macintosh™ platform

List and describe the similarities between different Unix development environments

Apply the Project Approach Strategy to help you systematically implement a program that satisfies the requirements of a given project specification

Iteratively apply the development cycle to help you implement your programming projects

List and describe the phases of the Project Approach Strategy

List and describe the steps of the software development cycle

List and describe the different development roles performed during the development cycle

Translate a project specification into a software design that can be implemented in C++

Implement a software design in C++ using a functional decomposition approach

List and describe the steps involved with functional decomposition

Describe how the development cycle can be employed in a tight spiral fashion

State the importance of compiling and testing early during the development process

Define the concept of a computer

Explain why the computer is a remarkable device

Explain how a computer differs from other machines

Explain how a computer stores and retrieves programs for execution

State the difference between a computer and a computer system

State the purpose of a microprocessor and the role it plays in a computer system

Define the concept of a program from the human perspective and the computer perspective

Describe how programs are represented in a computer's memory

List and describe the nine stages of the C++ program transformation process

List and describe the four steps of the processing cycle

State the purpose and objective of a computer's memory system

Define the concept of an algorithm

List the characteristics of a good algorithm

Describe what constitutes a minimum well-formed C++ program

List the keywords reserved for use by the C++ language

State the purpose of variables, constants, expressions, and statements

Demonstrate your ability to declare, define, and use variables

Demonstrate your ability to declare, define, and use constants

List and describe the purpose of the C++ fundamental data types

Determine data type sizes with the sizeof operator

Utilize variables and constants in simple C++ programs

List the native C++ operators and state their precedence

Write C++ programs using simple and compound statements

Describe variable scoping and state how the block structure of C++ can affect variable visibility

Utilize simple input and output techniques using the cin and cout objects

Describe the required parts of a minimal C++ program

Utilize an IDE's disassembly tool to gain deeper understanding of C++ program structure

List and describe the parts of a typical C++ program to include source files, main() function, library files, and preprocessor directives

Control the flow of program execution with C++ control flow statements

State the purpose and use of the if statement

Explain the purpose of a null statement

Utilize blocks to create local variable scopes in control statements

State the purpose and use of nested if statements

State the purpose and use of the for statement

State the purpose and use of nested for statements

State the purpose and use of the keywords break and continue

State the purpose and use of the while statement

State the purpose and use of the do statement

State the purpose and use of the switch statement

Explain the importance of using break to exit case statements properly

Explain the importance of a default case

Demonstrate your ability write effective, self-commenting expressions utilizing sound identifier naming techniques

State the purpose and use of pointers and references in C++

State the definition of an object

Explain how to determine an object's address using the & operator

Explain how to declare pointers using the pointer declarator *

Explain how to dereference a pointer using the * operator

Describe the concept of dynamic memory allocation

Explain how to dynamically create objects using the new operator

Explain how to destroy objects using the delete operator

Explain how to declare references using the reference declarator &

Explain why references must be defined at the point of declaration

Describe the benefits of using references vs. pointers

Utilize pointers and references to create powerful C++ programs

Describe the concept of an array

State the purpose and use of single- and multi-dimensional arrays

Describe how to declare and initialize single- and multi-dimensional arrays

Explain how the compiler uses the array's declared type to calculate offset addresses into an array

Explain how to access array elements using array subscript notation and pointer notation

List and describe the similarities between an array name and a pointer

Explain how to use pointers to dynamically allocate memory for an array with the new[] operator

Explain how to release dynamically allocated array memory with the delete[] operator
 Explain how to idiomatically process an array using a for loop
 Explain how to process multi-dimensional arrays using nested for loops
 Explain how strings are implemented in C++
 Utilize single- and multi-dimensional arrays in your C++ programming projects
 State the purpose and use of functions in C++
 Explain how to declare and define functions
 State the purpose and use of function return types
 State the purpose and use of function parameters
 Describe the concept of function calling
 Explain the use of local function variables and their scoping rules
 Describe how to pass arguments to a function by value and by reference
 Describe how to maximize function cohesion and minimize coupling
 Describe the concept of function signatures
 Describe how to overload functions
 Explain the concept of recursion
 Explain the concept and use of function pointers
 Explain how to create function libraries
 Utilize functions in your C++ programming projects
 Describe how functions are used to modularize C++ program functionality
 Demonstrate your ability to minimize function coupling and maximize function cohesion in C++ programming projects
 Create new data types to improve problem abstraction
 Use the typedef keyword to create type synonyms for existing data types better suited to the problem domain
 Explain how type synonyms can be used to improve program maintainability and readability
 Create and use enumerated data types in your programming projects
 Describe the default enum state values and explain how they can be changed
 Explain how to resolve enum state name conflicts
 Create and use structures in your programming projects
 Explain how to use the dot operator to access structure and class elements
 Create and use simple classes in your programming projects
 State the difference between structures and classes
 Describe when you would want to use structures vs. classes in a programming project
 List the key differences between structures and classes
 State the purpose and use of the this pointer
 List and define the following terms: class, base class, derived class, superclass, subclass, abstract base class, virtual function, object, message passing, OOA&D, inheritance, data encapsulation, interface, & implementation
 State the purpose and use of the class construct in C++
 List and describe the parts of a class declaration
 State the importance of the terminating semicolon of a class declaration
 Explain how to use access specifiers to control horizontal member access
 State the function and purpose of constructors
 State the purpose and use of overloaded constructors
 Explain how to overload constructors
 Explain how to use the initializer list to initialize class attributes
 State the purpose and use of destructors
 Explain how to overload class member functions
 Explain the importance of separating the class interface from its implementation
 Explain how to call class member functions from within class member functions
 Utilize complex class constructs in your C++ programming projects
 Utilize initializer lists to initialize class attributes
 List and define the following terms: constructor, destructor, default constructor, overloaded constructor, and overloaded functions
 Explain how to design complex classes using user-defined abstract data types
 Describe the concept of aggregation

STATE THE RELATIONSHIP BETWEEN AGGREGATION AND OBJECT LIFETIME
EXPLAIN THE DIFFERENCE BETWEEN CONTAINS BY VALUE AND CONTAINS BY REFERENCE
DESCRIBE THE CONCEPT OF SIMPLE AGGREGATION
DESCRIBE THE CONCEPT OF COMPOSITE AGGREGATION,
EXPLAIN HOW TO IMPLEMENT MESSAGE PASSING BETWEEN OBJECTS
EXPLAIN HOW TO UTILIZE POINTERS AND REFERENCES IN THE DESIGN OF COMPLEX CLASSES
EXPLAIN HOW TO EXPRESS AGGREGATION IN UML NOTATION
STATE THE PURPOSE AND USE OF A UML SEQUENCE DIAGRAM
DEMONSTRATE YOUR ABILITY TO USE SIMPLE AND COMPOSITE AGGREGATION TO IMPLEMENT C++ PROGRAMMING PROJECTS
STATE THE PURPOSE AND USE OF INHERITANCE IN C++ CLASS DESIGN
EXPLAIN HOW TO APPLY THE THREE ACCESS SPECIFIERS, PUBLIC, PROTECTED, AND PRIVATE
EXPLAIN HOW TO HIDE BASE CLASS FUNCTIONS WITH DERIVED CLASS FUNCTIONS
EXPLAIN HOW TO CALL A BASE CLASS CONSTRUCTOR FROM A DERIVED CLASS INITIALIZER LIST
EXPLAIN THE USE OF THE VIRTUAL KEY WORD AS IT RELATES TO DESTRUCTORS AND CLASS MEMBER FUNCTIONS
EXPLAIN HOW TO OVERRIDE VIRTUAL BASE CLASS FUNCTIONS
EXPLAIN HOW TO IMPLEMENT PURE VIRTUAL FUNCTIONS
EXPLAIN HOW TO DECLARE AND USE ABSTRACT BASE CLASSES
EXPLAIN HOW TO SUBSTITUTE DERIVED CLASS OBJECTS WHERE BASE CLASS OBJECTS ARE SPECIFIED
EXPLAIN HOW TO IMPLEMENT MULTIPLE INHERITANCE
STATE THE PURPOSE AND USE OF A VIRTUAL BASE CLASS
EXPLAIN HOW TO SAFELY USE INHERITANCE IN YOUR APPLICATION DESIGN
EXPLAIN HOW TO EXTEND THE UML CLASS DIAGRAM TO ILLUSTRATE CLASS INHERITANCE HIERARCHIES
DEMONSTRATE YOUR ABILITY TO EXPRESS INHERITANCE WITH A UML CLASS DIAGRAM
DEMONSTRATE YOUR ABILITY TO UTILIZE INHERITANCE IN THE DESIGN OF COMPLEX C++ PROGRAMMING PROJECTS
DEFINE THE TERM AD HOC POLYMORPHISM
EXPLAIN HOW TO ACHIEVE AD HOC POLYMORPHIC BEHAVIOR THROUGH OPERATOR OVERLOADING
IDENTIFY WHICH C++ OPERATORS CAN BE OVERLOADED
DEMONSTRATE YOUR ABILITY TO OVERLOAD THE FOLLOWING ARITHMETIC OPERATORS: +, -, *, /
DEMONSTRATE YOUR ABILITY TO OVERLOAD THE FOLLOWING RELATIONAL OPERATORS: <, >, <=, >=
DEMONSTRATE YOUR ABILITY TO OVERLOAD THE FOLLOWING EQUALITY OPERATORS: ==, !=
DEMONSTRATE YOUR ABILITY TO OVERLOAD THE FOLLOWING UNARY OPERATORS: PREFIX ++, POSTFIX ++, PREFIX -, POSTFIX -
DEMONSTRATE YOUR ABILITY TO OVERLOAD THE SUBSCRIPT OPERATOR: []
DEMONSTRATE YOUR ABILITY TO OVERLOAD IOSTREAM OPERATORS
EXPLAIN WHEN AND HOW TO USE FRIEND FUNCTIONS TO IMPLEMENT OPERATOR OVERLOADING
EXPLAIN WHEN OVERLOADED OPERATOR FUNCTIONS SHOULD BE CLASS MEMBERS
EXPLAIN WHY AND WHEN OPERATOR OVERLOADING IS RIGHT FOR YOUR DESIGN
EXPLAIN HOW TO ACHIEVE STATIC POLYMORPHIC BEHAVIOR THROUGH THE USE OF TEMPLATES
EXPLAIN HOW TO WRITE GENERIC CODE USING TEMPLATES
DESCRIBE THE CONCEPT OF A TEMPLATE CLASS
EXPLAIN HOW TO DECLARE AND IMPLEMENT FUNCTION TEMPLATES
EXPLAIN HOW TO DECLARE AND IMPLEMENT CLASS TEMPLATES
EXPLAIN HOW TO DECLARE AND IMPLEMENT CLASS MEMBER FUNCTION TEMPLATES
DEMONSTRATE YOUR ABILITY TO DECLARE AND IMPLEMENT SINGLE PARAMETER TEMPLATE CLASSES
DEMONSTRATE YOUR ABILITY TO DECLARE AND IMPLEMENT MULTIPLE PARAMETER TEMPLATE CLASSES
EXPLAIN HOW TO USE COMPONENTS OF THE C++ STANDARD TEMPLATE LIBRARY IN YOUR C++ PROGRAMMING PROJECTS
STATE THE PURPOSE AND USE OF STL ITERATORS, ALGORITHMS, AND CONTAINERS
DEMONSTRATE YOUR ABILITY TO UTILIZE CLASS AND FUNCTION TEMPLATES TO CREATE GENERIC CODE IN SUPPORT OF YOUR C++ PROGRAMMING PROJECTS
STATE THE DEFINITION OF DYNAMIC POLYMORPHISM
EXPLAIN HOW TO ACHIEVE DYNAMIC POLYMORPHIC BEHAVIOR THROUGH THE USE OF BASE CLASS POINTERS AND DERIVED CLASS OBJECTS
STATE THE IMPORTANCE OF ABSTRACT BASE CLASSES IN OBJECT-ORIENTED DESIGN
DESCRIBE THE ROLE VIRTUAL FUNCTIONS PLAY IN IMPLEMENTING DYNAMIC POLYMORPHIC BEHAVIOR
STATE THE PURPOSE AND USE OF VIRTUAL DESTRUCTORS
DESCRIBE THE CONCEPT OF PURE VIRTUAL FUNCTIONS

STATE THE PURPOSE AND USE OF ABSTRACT BASE CLASSES

STATE THE IMPORTANCE OF A CONSISTENT DERIVED CLASS INTERFACE AND THE ROLE IT PLAYS IN ACHIEVING ROBUST POLYMORPHIC BEHAVIOR

EXPLAIN WHY POLYMORPHIC BEHAVIOR IS A CRITICAL COMPONENT OF GOOD OBJECT-ORIENTED DESIGN

DEMONSTRATE YOUR ABILITY TO UTILIZE DYNAMIC POLYMORPHISM IN YOUR C++ PROGRAMMING PROJECTS

LIST AND DEFINE THE FOLLOWING TERMS: BASE CLASS, ABSTRACT BASE CLASS, VIRTUAL FUNCTION, PURE VIRTUAL FUNCTION, VIRTUAL DESTRUCTOR, INHERITANCE HIERARCHY, BASE CLASS POINTER, DERIVED CLASS OBJECT, AND DYNAMIC POLYMORPHIC BEHAVIOR

STATE THE IMPORTANT ROLE WELL-BEHAVED OBJECTS PLAY IN GOOD OBJECT-ORIENTED DESIGN

LIST AND DESCRIBE THE FUNCTIONS REQUIRED TO GET USER DEFINED OBJECTS TO BEHAVE LIKE NATIVE TYPES

LIST AND DESCRIBE THE FOUR MINIMUM FUNCTIONS REQUIRED TO IMPLEMENT THE ORTHODOX CANONICAL CLASS FORM

DEMONSTRATE YOUR ABILITY TO UTILIZE THE ORTHODOX CANONICAL CLASS FORM IN YOUR C++ PROGRAMMING PROJECTS

DEMONSTRATE YOUR ABILITY TO EXTEND THE ORTHODOX CANONICAL CLASS FORM TO SUIT THE NEEDS OF A PARTICULAR CLASS

EXPLAIN WHY COMPILER-SUPPLIED CONSTRUCTORS AND DESTRUCTORS MAY NOT PROVIDE APPROPRIATE OBJECT BEHAVIOR FOR COMPLEX CLASS TYPES

LIST AND DEFINE THE FOLLOWING TERMS: ORTHODOX CANONICAL CLASS FORM, DEFAULT CONSTRUCTOR, DESTRUCTOR, COPY ASSIGNMENT OPERATOR, COPY CONSTRUCTOR

EXPLAIN HOW TO CREATE AND INTEGRATE ASSEMBLY LANGUAGE OBJECT MODULES

EXPLAIN HOW TO INTEGRATE LEGACY C CODE

STATE THE PURPOSE AND USE OF THE EXTERN KEYWORD

EXPLAIN WHY THE EXTERN KEYWORD IS NECESSARY TO LINK TO LEGACY C CODE MODULES

DESCRIBE THE CONCEPT OF NAME MANGLING

EXPLAIN HOW TO CALL C AND C++ ROUTINES FROM JAVA APPLICATIONS

LIST THE STEPS REQUIRED TO CREATE, COMPILE, AND LINK TO AN ASSEMBLY LANGUAGE MODULE

LIST THE STEPS REQUIRED TO CREATE A JAVA JNI PROJECT AND CALL A C++ NATIVE METHOD FROM A JAVA PROGRAM

STATE THE PURPOSE AND USE OF THE JAVAH COMMAND LINE TOOL

DEMONSTRATE YOUR ABILITY TO UTILIZE ASSEMBLY LANGUAGE ROUTINES IN YOUR C++ PROGRAMMING PROJECTS

DEMONSTRATE YOUR ABILITY TO CALL NATIVE C++ FUNCTIONS FROM JAVA PROGRAMS

DEMONSTRATE YOUR ABILITY TO USE INLINE ASSEMBLY IN A MACINTOSH ENVIRONMENT

DEMONSTRATE YOUR ABILITY TO USE INLINE ASSEMBLY IN A PC ENVIRONMENT

LIST THE PREFERRED CHARACTERISTICS OF AN OBJECT-ORIENTED APPLICATION ARCHITECTURE

STATE THE DEFINITION OF THE LISKOV SUBSTITUTION PRINCIPLE (LSP)

STATE THE DEFINITION OF BERTRAND MEYER'S DESIGN BY CONTRACT (DbC)

RECOGNIZE THE CLOSE RELATIONSHIP BETWEEN THE LISKOV SUBSTITUTION PRINCIPLE AND DESIGN BY CONTRACT

SPECIFY PRECONDITIONS AND POSTCONDITIONS FOR CLASS AND INSTANCE FUNCTIONS

SPECIFY CLASS INVARIANTS

STATE THE DEFINITION OF THE OPEN-CLOSED PRINCIPLE (OCP)

STATE THE DEFINITION OF THE DEPENDENCY INVERSION PRINCIPLE (DIP)

APPLY THE LISKOV SUBSTITUTION PRINCIPLE IN THE DESIGN AND IMPLEMENTATION OF A CLASS INHERITANCE HIERARCHY

APPLY DESIGN BY CONTRACT IN THE DESIGN AND IMPLEMENTATION OF A CLASS INHERITANCE HIERARCHY

APPLY THE OPEN-CLOSED PRINCIPLE IN THE DESIGN AND IMPLEMENTATION OF A CLASS INHERITANCE HIERARCHY

APPLY THE DEPENDENCY INVERSION PRINCIPLE IN THE DESIGN AND IMPLEMENTATION OF A CLASS INHERITANCE HIERARCHY

STATE THE PURPOSE OF A UML MODELING TOOL

LIST KEY UML MODELING FEATURES SUPPORTED BY EMBARCADERO TECHNOLOGIES' DESCRIBE™

UTILIZE USE-CASE, SEQUENCE, AND CLASS DIAGRAMS TO ANALYZE AND DESIGN A SOLUTION TO A GIVEN PROGRAMMING PROBLEM

UTILIZE DESCRIBE™ TO DEVELOP A SOLUTION TO A GIVEN PROGRAMMING PROBLEM UP TO THE POINT OF C++ CODE GENERATION

SELECT THE APPROPRIATE UML DIAGRAM BASED ON THE CORRESPONDING PROBLEM ANALYSIS OR DESIGN PHASE

EMPLOY THE UML CONSTRUCTS OF AGGREGATION AND GENERALIZATION TO CREATE COMPLEX CLASS RELATIONSHIPS

UTILIZE DESCRIBE™ TO REVERSE ENGINEER EXISTING C++ SOURCE CODE

UTILIZE DESCRIBE™ TO GENERATE A WEB-BASED PROJECT REPORT

PREFACE

WELCOME – AND THANK YOU!

Thank you for choosing *C++ For Artists: The Art, Philosophy, and Science of Object-Oriented Programming*. You have selected an excellent book to augment your C++ and object-oriented programming learning experience.

If you purchased this book because it is required for a course you may feel like you had no say in the matter. Paying for expensive college books feels a lot like having your arm twisted behind your back at the checkout counter. If it will make you feel better I will let you in on a secret. You bought a keeper.

If you are standing in the computer section of your favorite book store reading these lines and trying hard to decide if you should buy this book or pay the rent I say to you this: If you knew the stuff inside this book you could easily own your own place. The landlord can wait.

TARGET AUDIENCE

C++ For Artists targets the student who demands more from a C++ programming textbook. What do I mean by student? A student is anyone who holds this book in their hands and by reading it expects to gain C++ and object-oriented programming knowledge. You may be a student enrolled in a high school, college, or university — or a practicing programmer seeking ways to expand your understanding of C++ and object-oriented programming. However you come to hold this book in your hands — you are my target audience.

APPROACH

C++ For Artists examines the topic of C++ and object-oriented programming from three unique perspectives.

First, programming is an art. It takes lots of skill (gained through study and training) and practice (gained from writing code) to succeed as a programmer. Talent separates the good programmers from the really great programmers. Just like some people have a knack for painting, some people have a knack for programming.

Second, object-oriented programmers can significantly benefit from a guiding philosophy. One that shows them how to tap their creativity, conquer challenges, and tame conceptual and physical complexity associated with large software systems.

Lastly, most programming students are not formally exposed to real-life, practical programming techniques and almost no object-oriented foundational theory during their tenure in the classroom.

These three perspectives: 1) programmer as artist, 2) creative approach philosophy, and 3) object-oriented programming theory, converge in *C++ For Artists* resulting in a truly unique programming text book.

ARRANGEMENT

The book is arranged into four parts: *Part I: The C++ Student Survival Guide*, *Part II: Language Fundamentals*, *Part III: Implementing Polymorphic Behavior*, and *Part IV: Intermediate Concepts*. Each part and its accompanying chapters are described in greater detail below.

PART I: THE C++ STUDENT SURVIVAL GUIDE

Part I: The C++ Student Survival Guide consists of four chapters designed to help you get a jump on your programming projects. The survival guide is meant to be referenced throughout your learning experience. The key features and discussion points of part I include:

- A discussion of the “flow”,
- A project approach strategy to be used to maintain a sense of progress when working on programming projects,
- A complete treatment on how to create C++ projects with two popular integrated development environments (IDEs) on Macintosh, Windows, and UNIX platforms,
- A step-by-step project walkthrough that applies the project approach strategy and development cycle to produce a complete working project.

CHAPTER 1: AN APPROACH TO THE ART OF PROGRAMMING

Chapter 1 begins with a discussion of the challenges you will face as you study C++ and object-oriented programming. It presents a project approach strategy specifically designed to help you maintain a sense of forward momentum when tackling your first programming projects. The chapter also presents a development methodology, a philosophical discussion of the concept of the “flow”, and practical advice on how to manage a programming project’s physical and conceptual complexity. I will show you how to use three important preprocessor directives: `#ifndef`, `#define`, and `#endif` to create separate header files. You may or may not be familiar with all the terms used in the chapter, especially those related to preprocessor directives and identifier naming, however, you are encouraged to return to the chapter as required. It serves to offer you a glimpse of things to come.

CHAPTER 2: SMALL VICTORIES: CREATING PROJECTS WITH IDEs

Chapter 2 shows you step-by-step how to create C++ projects using two popular integrated development environments: Metrowerks CodeWarrior on the Macintosh, and Microsoft Visual C++ for the Windows platform. The focus of the chapter is the concept of the project and the steps required to create projects regardless of the IDE employed. If you prefer to use UNIX development tools this chapter also shows you how to use the make utility and how to create a makefile that can be used to compile, link, and manage multi-file projects.

CHAPTER 3: PROJECT WALKTHROUGH: AN EXTENDED EXAMPLE

Chapter 3 takes you step-by-step through a complete programming project from specification to final implementation. Along the way you are shown how to apply the project approach strategy and the development cycle to arrive at an acceptable project solution. The `#ifndef`, `#define`, and `#endif` preprocessor directives are used to create safe header files that separate function interface declarations from function implementation code. If you are a novice student I do not expect you to fully comprehend all the material or programming techniques presented in this chapter, rather, the material serves as an excellent reference to which you will return periodically as you use bits and pieces of this knowledge in your programming projects.

CHAPTER 4: COMPUTERS, PROGRAMS, AND ALGORITHMS

Chapter 4 presents background information on computer hardware organization, memory systems, and algorithms. The emphasis is on understanding exactly what a program is from a computer and human perspective. I discuss the four phases of the program execution cycle, how program instructions are differentiated from ordinary data, and how memory is organized on common computer systems. I also talk about what makes a good and bad algorithm.

PART II: C++ LANGUAGE FUNDAMENTALS

Part II presents a treatment of the core C++ programming language features and comprises chapters 5 through 13. This is a critical part of the book because it prepares you for further study of intermediate and advanced C++ and object-oriented concepts. The key features and discussion points of part II include:

- The unique ordering of the material. For instance, pointers are covered early so you will understand their use in other language constructs,
- Pointers are presented as a dialog between a superhero named C++ Man and a confused student named Perplexed One,
- Emphasis on multi-file projects,
- Lots of targeted code examples to reinforce key lecture points,
- Successive chapters build upon knowledge gained from the previous chapter,
- In-depth coverage of tricky concepts normally glossed over or avoided in ordinary C++ texts.

CHAPTER 5: SIMPLE PROGRAMS

Chapter 5 shows you how to write simple C++ programs using fundamental data types and simple expressions. I give examples of how to use all the C++ operators, how to create local and multi-file variables and constants, and show you how you can limit a variable's scope to one file. You will learn how to write two versions of the main() function and how to call functions upon program exit.

CHAPTER 6: CONTROLLING THE FLOW OF PROGRAM EXECUTION

Chapter 6 moves beyond simple programs and shows you how to control the flow of program execution by using if, if-else, switch, for, while, and do-while statements. Many source code examples and diagrams are used to illustrate how control flow statements are written. The chapter includes a discussion of statements, null statements, and compound statements. I also show you how to write nested if, for, and while statements, and how to write loops that will repeat until explicitly exited.

CHAPTER 7: POINTERS AND REFERENCES

Chapter 7 uses a short story to simplify the complex topic of pointers and references. Perplexed One is a student who falls asleep in class and is awakened by the arrival of C++ Man. C++ Man then helps Perplexed One by answering questions and giving examples of how to declare and use pointers.

CHAPTER 8: ARRAYS

Chapter 8 builds upon chapter 7 and shows the relationship between pointers and arrays. The chapter continues by showing you how to build single and multi-dimensional static and dynamic arrays. Lots of code examples and diagrams help you visualize how arrays are declared, initialized, and used in programs.

CHAPTER 9: FUNCTIONS

Chapter 9 builds upon chapter 8 and shows you how to write complex functions that can pass arguments by value and by reference. The emphasis is on writing highly cohesive functions that are minimally coupled to other program elements. Header files are used to separate function declaration (interface) from definition (implementation). To support the creation of header files I review and discuss the three important preprocessor directives: #ifndef, #define, and #endif. Other topics covered include: function variable scoping, static function variables, passing arrays to functions, passing multi-dimensional arrays to functions, returning pointers from functions, how to avoid dangling references, function overloading, recursion, function pointers, and call back functions.

CHAPTER 10: TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES

Chapter 10 shows you how to create type synonyms and new data types using type definitions, enumerated types, structures, and classes. The emphasis is on problem abstraction and how it is used to map a real world problem to a set of supporting data structures that can be used in a program. Structures are compared to classes and the notion of object-oriented programming is introduced. The class declaration is discussed as it relates to the structure declaration

and how the notions of procedural and object-oriented programming differ from each other.

CHAPTER 11: DISSECTING CLASSES

Chapter 11 continues the discussion of classes and how they work. It introduces the UML class diagram and uses UML class diagrams to illustrate static class relationships. The special member functions are thoroughly discussed. These include the constructor, destructor, copy constructor, and copy assignment operator. A brief introduction to the orthodox canonical class form is given in preparation for a deeper treatment of the subject in chapter 17. Other topics include data encapsulation, member functions and attributes, access specifiers, member function overloading, and how to separate class interface from implementation.

CHAPTER 12: COMPOSITIONAL DESIGN

Chapter 12 builds upon chapter 11 and shows you how to build complex class types using simple and complex aggregation. The UML class diagram is extended to model simple and composite aggregate class relationships. The UML sequence diagram is also introduced to illustrate interobject message passing. Other topics discussed include: managing physical complexity, the use of pointers and references to build simple and complex aggregate classes, and how to properly use constructors and destructors in aggregate classes. The chapter concludes with a complex aggregation example.

CHAPTER 13: EXTENDING CLASS FUNCTIONALITY THROUGH INHERITANCE

Chapter 13 introduces the topic of inheritance and shows you how to extend class behavior through subclassing and subtyping. UML is used to illustrate simple and complex inheritance hierarchies. The compositional design techniques discussed in chapter 12 are combined with inheritance design concepts to provide you with a powerful arsenal of object-oriented design tools. The access specifiers public, protected, and private are discussed in the context of inheritance. Other topics covered include: virtual functions, function hiding, function overloading, pure virtual functions, abstract classes, abstract base classes, multiple inheritance, and virtual inheritance. The chapter includes a complex navy fleet simulation example that illustrates the use of inheritance and compositional design.

PART III: IMPLEMENTING POLYMORPHIC BEHAVIOR

Part III gives special coverage to the three types of polymorphic behavior: ad hoc (operator overloading), static (templates), and dynamic (base class pointers to derived class objects). Success as a C++ programmer demands a thorough understanding of these concepts. Key features and discussion points of part III include:

- In-depth treatment of ad-hoc, static, and dynamic polymorphism and how each type of polymorphic behavior is achieved using the C++ language,
- An example of how to overload almost every operator in the C++ language,
- How to overload the iostream operators to tailor them to your class needs,
- How to think about and apply the notion of polymorphic behavior in your application designs,
- How to write generic code using templates,
- How to use multiple place holders in template classes and functions,
- How to use the special template definition syntax to explicitly specify template parameter types,
- How to design with dynamic polymorphic behavior in mind.

CHAPTER 14: AD HOC POLYMORPHISM: OPERATOR OVERLOADING

Chapter 14 is devoted to operator overloading. It builds upon the concepts of function overloading and shows you how to overload nearly every operator in the C++ language complete with examples of their use. A complete table of overloadable operators is included along with a discussion of how to overload the iostream operators to tailor them to your class needs.

CHAPTER 15: STATIC POLYMORPHISM: TEMPLATES

Chapter 15 shows you how to write generic code using templates. It shows you how to replace overloaded functions with template functions and how to use template functions in your programs. The chapter also shows you how to

use the special template definition syntax to explicitly specify template parameter types. A brief overview of the C++ standard template library (STL) is offered along with a discussion of STL containers, iterators, and algorithms.

CHAPTER 16: DYNAMIC POLYMORPHISM: OBJECT-ORIENTED PROGRAMMING

Chapter 16 reinforces and builds upon concepts introduced in chapter 13. The focus is on the C++ language constructs required to write truly object-oriented programs. Topics discussed in depth include: employing pure virtual functions to create abstract base classes, how to use abstract base classes to specify the interface to derived classes, and what behavior to expect when using dynamic polymorphic programming techniques. The engine component aggregate class created in chapter 12 is revisited and redesigned to employ dynamic polymorphic behavior.

PART IV: INTERMEDIATE CONCEPTS

Part IV consists of four chapters and builds upon the concepts and material presented in the preceding three parts. Key features and discussion points of part IV include:

- How to write well-behaved, context-minded classes using the orthodox canonical class form,
- How to use legacy C code libraries in your C++ applications,
- How to use the Java Native Interface (JNI) to write C++ functions that can be called from Java applications,
- How to use assembly language in C++ programs,
- Coverage of three important object-oriented design concepts to include the Liskov substitution principle and Meyer design by contract programming, the open-closed principle, and the dependency inversion principle,
- How to use a UML design tool to assist in the design and implementation of complex applications,
- How to use a UML tool to reverse engineer existing C++ code.

CHAPTER 17: WELL-BEHAVED OBJECTS: THE ORTHODOX CANONICAL CLASS FORM

Chapter 17 presents an in-depth discussion of the orthodox canonical class form (OCCF) to write well-behaved, context-minded classes. Keeping the OCCF in mind when you design and write classes forces you to consider how those classes will be used in an application. The class's possible uses or usage contexts will guide you in your choice of which operators to overload to insure your class objects exhibit predictable and acceptable behavior.

CHAPTER 18: MIXED LANGUAGE PROGRAMMING

Chapter 18 shows you how to use C++ with C, assembly, and Java. Topics covered include: using the extern keyword to prevent C++ name mangling, the Java Native Interface (JNI) and how to write C++ functions that can be called from Java programs, how to use inline assembly code in C++ programs using the asm keyword, and how to link to object modules created in assembly language.

CHAPTER 19: THREE DESIGN PRINCIPLES

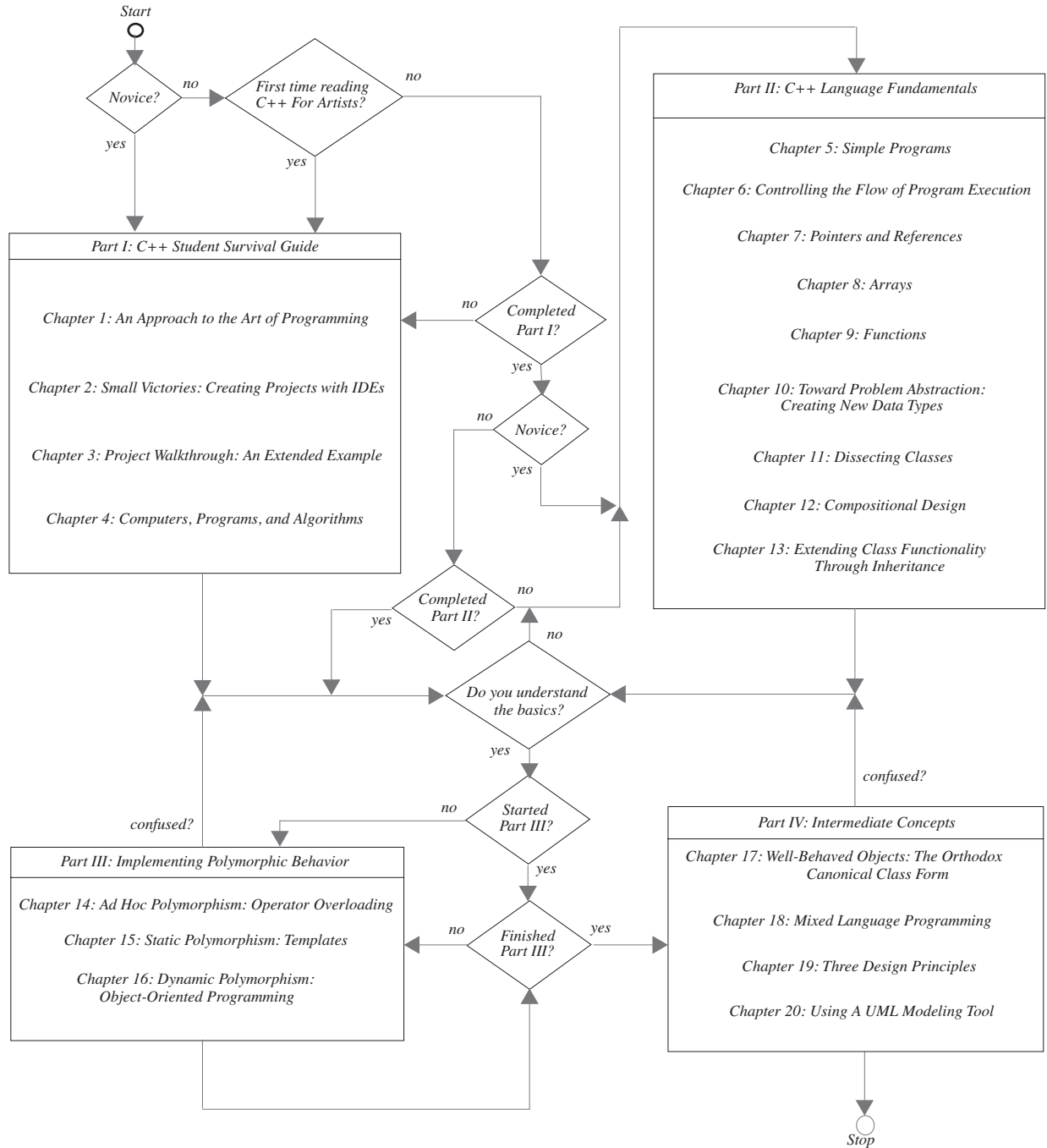
Chapter 19 presents and discusses three important object-oriented design principles: the Liskov substitution principle, the open-closed principle, and the dependency inversion principle. Bertrand Meyer's design by contract programming is discussed in relation to the Liskov substitution principle.

CHAPTER 20: USING A UML MODELING TOOL

Chapter 20 discusses the importance of using a UML design tool to assist in the application design process. The featured UML tool is Embarcadero Technologies's Describe™. The chapter focuses on only a few of Describe's many features: UML use-case, class, and sequence diagram creation, how to link diagram objects to other diagrams, how to generate code from class diagrams, how to reverse engineer existing C++ code, and how to generate comprehensive web-based project reports.

How To Read C++ FOR ARTISTS

The flow chart below is intended to give you an idea of how to read C++ For Artists. If you are a complete novice or first-time reader of C++ For Artists I recommend starting with part I. There you will find much to help you undertake significant programming projects. Read part II repeatedly to ensure you have a firm grasp of C++ fundamental language features before attempting parts III and IV.



Pedagogy

CHAPTER LAYOUT

Each chapter takes the following structure:

- Learning Objectives
- Introduction
- Content
- Quick Reviews
- Summary
- Skill Building Exercises
- Suggested Projects
- Self Test Questions
- References
- Notes

LEARNING OBJECTIVES

Each chapter begins with a set of learning objectives. The learning objectives represent the knowledge gained by reading the chapter material and completing the skill building exercises, suggested projects, and self test questions.

INTRODUCTION

The introduction motivates you to read the chapter content.

CONTENT

The chapter content represents the core material. Core material is presented in sections and sub-sections.

QUICK REVIEWS

The main points of each level 1 section are summarized in a quick review section.

SUMMARY

The summary section summarizes the chapter material

SKILL BUILDING EXERCISES

Skill building exercises are small programming or other activities intended to strengthen your C++ programming capabilities in a particular area. They could be considered focused micro-projects.

SUGGESTED PROJECTS

Suggested projects require the application of a combination of all knowledge and skills learned up to and including the current chapter to complete. Suggested projects offer varying degrees of difficulty.

SELF TEST QUESTIONS

Self-test questions test your comprehension on material presented in the current chapter. Self-test questions are directly related to the chapter learning objectives. Answers to all self-test questions appear in appendix C.

REFERENCES

All references used in preparing chapter material are listed in the references section.

NOTES

Note taking space.

CD-ROM

The CD-ROM contains the following goodies:

- PDF edition of C++ For Artists,
- Adobe Acrobat™ Reader version 6 for Windows and Macintosh
- Demo version of Embarcadero Technologies Describe™ UML modeling tool,
- Full working copy of ObjectPlant™ UML modeling tool for the Macintosh™,
- Open source C++ compiler tools,
- All source code example files used throughout the text organized by chapter,
- Metrowerks CodeWarrior projects,
- Links to commercial C++ development tool vendors.

SupportSite™ Website

The C++ For Artists SupportSite™ is located at [<http://pulpfreepress.com/SupportSites/C++ForArtists/>]. The support site includes source code, links to C++ compiler and UML tool vendors, and corrections and updates to the text.

Problem Reporting

Although every possible effort was made to produce a work of superior quality some mistakes will no doubt go undetected. All typos, misspellings, inconsistencies, or other problems found in C++ For Artists are mine and mine alone. To report a problem or issue with the text please contact me directly at rick@pulpfreepress.com or report the problem via the C++ For Artists SupportSite™. I will happily acknowledge your assistance in the improvement of this book both online and in subsequent editions.

Acknowledgements

C++ For Artists was made possible by the hard work and support of many talented people and companies. Some friends contributed unknowingly in unexpected ways.

I would first like to thank *Harish Ruchandani* and *Tracy Millman*, my former colleagues at Booz | Allen | Hamilton, for patiently listening to my ideas about writing this book and for providing critical comment on early versions of several chapters.

Many thanks to my good friend *Jose Pi* for many great mornings spent surfing California waves, and to *Michael Leahy*, a merchant mariner of the highest caliber, for letting me drive his Ferrari with no strings attached.

I would like to thank *Anke Braun*, *Thayne Conrad*, and *Petra Rector* of Prentice-Hall for entertaining my proposal and trying to fit C++ For Artists into the Prentice-Hall product line. Thanks also go to *Jim Leisy* of Franklin, Beedle & Associates, Inc., for seeing the merit in this work.

Special thanks go to the reviewers employed by Prentice-Hall who provided invaluable critical comment on chapters 1 through 13. They include: *John Godel*, *James Huddleston*, *Dr. Samuel Kohn*, and *Ms. Anne B. Horton*. C++ For Artists is significantly improved by their attention to detail.

Independent reviewers of different portions of the text include *Ken Stern* and *Brendan Richards* of SAIC. It is truly a pleasure working with such talented people.

I want to thank *Apple™ Computer Inc.*, for providing product images of the PowerMac™, *Motorola, Inc.* for providing images of the PowerPC 7400 and related architecture diagrams, *Embarcadero Technologies, Inc.*, for granting me a full-use license of Describe™, and *Michael Archtadeous* for working in the trenches to produce Object-Plant™.

Lastly, without the fathomless patience of Coralie Miller, an amazing woman, this book would simply not exist.

A handwritten signature in black ink, consisting of a large, stylized 'R' followed by a 'M' and a long horizontal flourish extending to the right.

Rick Miller
Falls Church, Virginia
7 July 2003

PART I: THE C++ STUDENT SURVIVAL GUIDE

CHAPTER 1



SNOW REFLECTIONS

AN APPROACH TO THE ART OF PROGRAMMING

LEARNING OBJECTIVES

- *Identify and overcome the difficulties encountered by students when learning how to program*
- *List and explain the software development roles played by students*
- *List and explain the phases of the tight spiral software development methodology*
- *Employ the concept of the flow to tap creative energy*
- *List and explain the primary areas of the Project Approach Strategy*
- *State the purpose of a header file*
- *State the purpose of an implementation file*
- *Explain the importance of separating interface from implementation*
- *Employ multi-file programming techniques to tame project complexity*
- *Explain the use of #ifndef, #define, and #endif preprocessor directives*
- *Apply preprocessor directives to implement multi-file programming projects*
- *State the importance of adopting consistent variable and constant naming conventions*
- *List and describe the two types of C++ comments*

INTRODUCTION

Programming is an art; there's no doubt about it. Good programmers are artists in every sense of the word. They are a creative bunch, although some would believe themselves otherwise out of modesty. Like any art you can learn the secrets of the craft. That is what this chapter is all about.

Perhaps the most prevalent personality trait I have noticed in good programmers is a knack for problem solving. Problem solving requires creativity, and lots of it. When you program a computer you are solving a problem with a machine. You transfer your knowledge of a particular problem into code, transform the code into a form understandable by a machine, and run the result on a machine. Doing this requires lots of creativity, especially when you find yourself stumped by a particular problem.

The material presented here is wrought from experience. Believe it or not, the hardest part about learning to program a computer, in any programming language, is not the learning of the language itself, rather, it is learning how to approach the art of problem solving with a computer. To this end the material in this chapter is aimed squarely at the beginner. However, I must issue a word of warning. If you are truly a novice, then some of what you read in this chapter will make less sense to you than to someone already familiar with C or C++. Do not worry, it is that way by design. If you feel like skipping parts of this chapter now, then go right ahead. The material will be here when you need it. In fact, you will grow to appreciate this chapter more as you gain experience as a programmer.

THE DIFFICULTIES YOU WILL ENCOUNTER LEARNING C++

During your studies of the C++ programming language you will face many challenges and frustrations. However, the biggest problem you will encounter is not the learning of the language itself, but the many other skills and tools you must learn before writing programs of any significance or gaining any measure of proficiency in solving problems with C++. If you are a seasoned student or practicing computer professional returning to the classroom to upgrade your skills, you have the advantage of experience. You can concentrate on learning the syntax and nuances of C++ and very quickly apply its powers to problems at hand. If you are an absolute beginner, however, you have much to learn.

Required Skills

In addition to the syntax and semantics of the C++ language you will need to master the following skills and tools:

- A development environment, which could be as simple as a text editor and compiler combination or a commercial product that integrates editing, compiling, and project management capabilities into one suite of tools,
- A computing platform of choice,
- Problem solving skills,
- How to approach a programming project,
- How to manage project complexity,
- How to put yourself in the mood to program,
- How to stimulate your creative abilities,
- Object-oriented analysis and design,
- Object-oriented programming principles.

The PLANETS Will COME INTO ALIGNMENT

I use a metaphor to describe what it takes before you can get even the simplest program to execute properly. It is as if the planets must come into alignment. You must learn a little of each skill and tool listed above, with the exception of object-oriented programming principles and object-oriented analysis and design, to write, compile, and run your first C++ program. But, when the planets do come into alignment, and you see your first program compile and execute, and you begin to make sense of all the class notes, documentation, and text books you have studied up to that point, you will spring up from your chair and do a victory dance. It is a great feeling!

How This CHAPTER Will Help You

This chapter will give you the information you need to bring the planets into alignment sooner rather than later. It presents an abbreviated software development methodology that formalizes the three primary roles you play as a programming student. It will discuss some philosophical topics related to tapping into your creative energies. It will offer several strategies to help you manage project complexity, something you will not need for very small projects but should get into the habit of doing as soon as possible.

I recommend you read this chapter at least once in its entirety and refer back as necessary as you progress through the text and attempt increasingly difficult programming assignments.

PROJECT MANAGEMENT

THREE SOFTWARE DEVELOPMENT ROLES

You will find yourself assuming the duties and responsibilities of three software development roles: Analyst, Architect, and Programmer.

Analyst

When you are handed a class programming project you may or may not understand what the instructor is actually asking you to program. Hey, it happens! Whatever the case may be, you, as the student, must read the assignment and design and implement a solution.

You can think of a project assignment as a requirements specification. They will come in several flavors. Some instructors go into painful detail about how they want the student to execute the project. Others prefer to generally describe the type of program they want thus leaving the details, and the creativity, up to you. There is no one correct method of writing a project assignment; each has its benefits and limitations.

A detailed assignment takes a lot of the guesswork out of what outcome the instructor expects. On the other hand, having every design decision made for you may prevent you from solving the problem in a unique, creative way.

A general project assignment delegates a lot of decision making to the student while also adding the responsibility of determining what project features will satisfy the assignment.

Both types of assignments model the real world to some extent. Sometimes requirements are well defined and there is little doubt what shape the final product will take and how it must perform. However, more often than not requirements are ill or vaguely defined. As an analyst you must clarify what is being asked of you. In an academic setting, do this by talking to the instructor and have them clarify the assignment. A clear understanding of the assignment will yield valuable insight into possible approaches to a solution.

Architect

Once you understand the assignment you must design a solution. If your project is extremely small you could perhaps skip this step with no problem. However, if your project contains several objects that interact with each other, then your design, and the foundation it sets, could make the difference between success and failure. A well-designed project reflects a subliminal quality that poorly designed projects do not.

Two objectives of good design are the ability to accommodate change and tame complexity. Change in this context means the ability to incrementally add features to your project as it grows without breaking the code you have already written. Several important object-oriented principles have been formulated to help tame complexity and will be discussed later in the book. For starters though, begin by imposing a good organization upon your source code files. You can use the source code file formats presented below to help in this endeavor.

PROGRAMMER

As programmer you will execute your design. The important thing to note here is that if you do a poor job as an architect your life as a programmer will be miserable. That doesn't mean the design has to be perfect. I will show you how to incrementally develop and make improvements to your design as you code.

Now that you know what roles you will play as a student let us discuss how you might approach a project.

A PROJECT APPROACH STRATEGY

Most students have difficulty implementing their first significant programming assignment, not because they lack brains or talent, but because they lack experience. If you are a novice and feel overwhelmed by your first programming project rest assured you are not alone. The good news is that with practice, and some small victories, you will quickly gain proficiency at formulating approach strategies to your programming projects.

Even experienced programmers may not immediately know how to solve a problem or write a particular piece of code when tasked to do so. What they do know, however, is how to formulate a strategy to solve the problem.

YOU HAVE BEEN HANDED A PROJECT – NOW WHAT?

Until you gain experience and confidence in your programming abilities the biggest problem you will face when given a large programming assignment is where to begin. What you need to help you in this situation is a project approach strategy. The strategy is presented below and discussed in detail. I have also summarized the strategy in a checklist located in appendix A. Feel free to reproduce the checklist and use as required.

The project approach strategy is a collection of areas of concern to take into consideration when you begin a programming project. It is not a hard, fast list of steps you must take. It is intended to put you in control, to point you in the right direction, and give you food for thought. It is flexible. You will not have to consider every area of concern for every project. After you have used it a few times to get you started you may not ever use it explicitly again. As your programming experience grows feel free to tailor the project approach strategy to suit your needs.

STRATEGY AREAS OF CONCERN

The project approach strategy is formulated around areas of concern. These include requirements, problem domain, language features, and design. When you use the strategy to help you solve a programming problem your efforts become focused and organized rather than ad hoc and confused. You will feel like you are making real progress rather than drowning in a sea of confusion.

REQUIREMENTS

A requirement is an assertion that specifies a particular aspect of expected behavior. A project's requirements are contained in a project specification or programming assignment. Ensure you completely understand the project specification. Seek clarification if you do not know, or if you are not sure, what problem the project specification is asking you to solve. In my academic career I have seen projects so badly written that I thought I had a comprehension problem. I'd read the thing over and over again until struck by a sudden flash of inspiration. But more often than not I would reinforce what I believed an instructor required by discussing the project with them.

PROBLEM DOMAIN

The problem domain is the specific problem you are tasked to solve. I would say that it is that body of knowledge necessary to implement a software solution apart and distinct from the knowledge of programming itself. For instance, "Write a program to simulate elevator usage in a skyscraper." You may understand what is being asked of you (requirements understanding) but not know anything about elevators, skyscrapers, or simulations (problem domain). You need to become enough of an expert in the problem domain you are solving so that you understand the issues involved.

PROGRAMMING LANGUAGE FEATURES

The source of greatest frustration to novice students at this stage of the project is knowing what to design but not knowing enough of the language features to begin the design. This is when panic sets in and students begin to buy extra books in hopes of discovering the Holy Grail of project wisdom.

To save yourself from panic make a list of the language features you need to understand and study each one, marking them off as you go. This provides focus and a sense of progress. As you read about each feature, keep notes on their usage so you can refer to them when you sit down to formulate your program design.

DESIGN

When you are ready to design a solution you will usually be forced to think along two completely different lines of thought: procedural vs. object-oriented.

PROCEDURAL DESIGN

A procedural design approach is one in which you identify and implement program data structures separate from the functions that manipulate those data structures. When taking a procedural approach to a solution you will break the problem into small, easily solvable pieces, implement the solution to each of the pieces, and combine the solved pieces into a complete problem solution. The solvable pieces I refer to here are functions. This methodology is also known as functional decomposition.

OBJECT-ORIENTED DESIGN

Object-oriented design refers to designing with objects and their interfaces. Whereas a procedural design treats data structures separately from the functions that manipulate them, object-oriented design uses encapsulation to hide an object's implementation data structures behind a public interface. Data structures and the functions that manipulate them combine to form classes from which objects can then be created.

A problem solved with an object-oriented approach is decomposed into a set of objects and their behavior. Design tools such as the Unified Modeling Language (UML) can be used to help with this task. Once the objects in a system are identified, a set of interface functions is then identified for each object. Classes are declared and defined to implement the interface functions. Once all the program classes have been designed and written, they are combined and used together to form the final program. Note that when using the object-oriented approach you are still breaking a problem into solvable pieces, only now the solvable pieces are objects that represent the interrelated parts of a system.

Once you get the hang of object-oriented design you will never return to functional decomposition again. However, after having identified the objects in your program and the interfaces they should have, you will have to implement your design. This means writing class member functions one line of code at a time.

Think Abstractly

One mistake students often make is to think too literally. It is very important to remember that the act of solving a real world problem with a computer requires abstraction.

THE STRATEGY IN A NUTSHELL

Identify the problem, understand the problem, make a list of language features you need to study and check them off as you go. Once you formulate a solution to the problem, break the problem into manageable pieces, solve each piece of the problem, and then combine the solved pieces to form a total solution.

Applicability To THE REAL WORLD

The programming problem solution strategy presented above is not intended to replace a formal course on software engineering, but it will help you when you enter the real world as a commercial programmer. In that world you will soon discover that all companies and projects are not created equal. Different companies have different design philosophies. Some companies have no software design philosophy. If you find yourself working for such a company you will probably be the software engineering expert!

THE ART OF PROGRAMMING

Programming is an art. Ask any programmer and they will agree — it takes a lot of creativity to solve problems with a computer. Creative people have an advantage in that they are not afraid to explore new avenues of design. Their open-mindedness and readiness to accept new ideas give them the ability to see problems differently from people who tend toward the cut and dry. This section offers a few suggestions on how you can stimulate your creativity.

DON'T START AT THE COMPUTER

Unless you have a good idea about what source code to write, sitting down at the computer first thing, without thinking through some design issues, is the worst mistake you can make. If you have ever suffered from writer's block when writing a paper for class then you can begin to understand what you will experience if you begin your project at the computer.

I recommend you forget the computer and go some place quiet and relaxing, with pen and paper, and draft a design document. It does not have to be big. Entire system designs can be sketched on the back of a napkin. The important thing is to have given some prior thought as to the design and structure of your program before you start coding.

The location you choose to relax in is important. It should be someplace where you feel really comfortable. If you like quiet spaces then seek quiet spaces; if you like to watch people walk by and think of the world, then an outdoor cafe may be the place for you. Inside, outside, at the beach, on the ski slope, wherever you prefer.

What you seek is the ability to let your mind grind away on the solution. Let your mind do the work. Writing code at the computer is a mechanical process. Formulating the solution is where real creativity is required, and is the part of the process that requires the most brainpower. Typing code is more like a drill on attention to detail.

INSPIRATION STRIKES AT THE WEIRDEST TIME

If you let your mind work on the problem it will offer its solution to you at the weirdest times. I solve most of my programming problems in my sleep. As a student I kept computers in the bedroom and would get up at all hours of the night to work on ideas that had popped into my head in a dream.

Try to have something to write on close at hand at all times. A pad of paper and pen next to the toilet comes in handy! You can also use a small tape recorder, or digital memo recorder, or your personal digital assistant. Whatever means suit your style. Just be prepared. There's nothing worse than the sinking feeling of having had the solution come to you in the middle of the night, or in the shower, or on the drive home from work or school, and say "I will remember that and write it down later," only to forget it and have no clue what you were thinking when you do finally get something with which to record your ideas.

OWN YOUR OWN COMPUTER

Do not rely on the computer lab! I repeat. Do not rely on the computer lab! They are the worst places for inspiration and cranking out code. Most schools use PC's running Windows or some flavor of Unix and/or Macintosh computers.

YOU EITHER HAVE TIME AND NO MONEY, OR MONEY AND NO TIME

The one good reason for not having your own personal computer to program your projects on is severe economic circumstance. Full-time students sometimes fall into this category. What they usually have gobs of is time. So much time that they spend their entire days at school and complain about not having a social life. But they can stay in the computer labs all day long and even be there when it is relatively quiet.

On the other hand, you may work full-time and be a part-time student. If this describes you then you don't have time to screw around driving to school to use the computer labs. You will gladly pay for any book or software package that makes your life easier and saves you time.

The Family Computer Is Not Going To Cut It!

If you are a family person working full-time and attending school part-time then time is a precious commodity. If you have a family computer that everyone shares, adults as well as children, then get another computer and put it off limits to everyone but yourself and password protect it. This will ensure your loving family does not accidentally wipe out your project the night before it is due through some unfortunate accident. It happens, don't kid yourself. Ensure your peace of mind by having your own computer in your own little space with a sign on it that reads "Touch This Computer And Die!"

SET THE MOOD

When you have a good idea on how to proceed with entering source code you will want to set the proper programming mood.

LOCATION, LOCATION, LOCATION

Locate your computer work area someplace that's free from distraction. If you are single this may be easier than if you are married with children. If you live in a dorm or frat house good luck! Perhaps the computer lab is an alternative after all.

Have your own room if possible, or at least your own corner of a larger room that is recognized as a quiet zone. Noise canceling headphones might help if you find yourself in this situation.

Set rules. Let your friends and family know that when you are programming not to bother you. I know it sounds rude but when you get into the flow, which is discussed below, that is, if you ever get into the flow, you will be really upset when someone interrupts your train of thought to ask you about school lunch tomorrow or the location of the car keys. Establish the ground rules up front that say when it is a good time to disturb you when you are programming. The rule is - never!

CONCEPT OF THE FLOW

Artists can really become absorbed in their work, not eating and ignoring personal hygiene for days, even weeks, at a time. Those who have experienced such periods of intense concentration and work describe it as a transcendental state where they have complete clarity of the finished product and tune out the world around them, living inside a cocoon of thought and energy.

Programmers can get into the flow. I have achieved the flow. You can achieve the flow and when you do you will crave the feeling of the flow again. It is a good feeling, one of complete and utter understanding of what you are doing and where you are going with your source code. You can do amazing amounts of programming while in the flow.

The Stages of Flow

Like sleep, there are stages to the flow.

GETTING SITUATED

The first stage. You sit down at the computer and adjust your keyboard and stuff around you. Take a few deep breaths to help you relax. By now you should have a good idea of how to proceed with your coding. If not you shouldn't be sitting at the computer.

RESTLESSNESS

Second stage. You may find it difficult to clear your mind from the everyday thoughts that block your creativity and energy. Maybe you had a bad day at work, or a great day. Perhaps your spouse or significant other is being a complete jerk! Perhaps they're treating you very good and you are wondering why?

Close your eyes and breathe deep and regular. Clear your mind and think of nothing. It is hard to do but you can do it with practice. When you can clear your mind and free yourself from distracting thoughts you will find yourself ready to begin coding.

Settling In

Now, your mind is clear. Non-productive thoughts are tucked neatly away. You begin to program. Line by line your program takes shape. You settle in and the clarity of your purpose takes hold and propels you forward.

Calm and Complete Focus

You don't notice it at first, but at some point between this and the previous stage you have slipped into a deeply relaxed state and are utterly focused on the task at hand. It is like reading a book and becoming completely absorbed. Someone can call your name but you will not notice, and not respond until they either shout at you or do something to break your concentration.

You know you were in the flow, if only to a small degree, when being interrupted brings you out of this focused state and you feel agitated and have to settle in once again. If you avoid doing things like getting up from your chair for fear of breaking your concentration or losing your thought process then you are in the flow!

BE EXTREME

Kent Beck, in his book "Extreme Programming Explained", describes the joy of doing really good programming. The following programming cycle is synthesized from his extreme programming philosophy.

THE PROGRAMMING Cycle

Plan

Plan a little. Your project design should serve as a guide in your programming efforts. Your design should also be flexible and accommodate change, which means that as you program, you may make changes to the design.

Essentially, you will want to design to the point where you have enough of the design to allow you to begin coding. The act of coding will soon reinforce your design decisions or detect fatal flaws that you must correct if you hope to have a polished, finished project.

Code

Code a little. Write code in small, cohesive modules. A class or function at a time is good granularity.

Test

Test a lot. Test each class, module or function separately or in whatever grouping makes sense. You will find yourself writing little programs on the side called test cases to test the code you have written. It is a good practice to get into. A test case is nothing more than a small little program you write and execute in order to test the functionality of some component or feature you have finished coding before integrating that component or feature into your project. The objective of testing is to break your code and correct its flaws before it has a chance to break your project in ways that are hard to detect.

Integrate

Integrate often. Once you have a tested module of code, be it either a function or complete set of related classes, integrate these tested components into your project regularly. The objective of regular integration is to see if the newly integrated components break any previously integrated components. If they do then remove them from the project and fix the problem. If a newly integrated component does break something you may have discovered a design flaw or a previously unnoticed dependency between components. If this is the case then the next step in the programming cycle should be performed.

Factor

Factor the design when possible. If you discover design flaws or ways to improve the design of your project you should factor the design to accommodate further development. An example of design factoring might be the migration of common elements from derived classes into the base class to take better advantage of code reuse.

REPEAT

Apply the programming cycle in a tight spiral fashion. You will quickly reach a point in your project where it all starts to come together, and very quickly so.

The Programming Cycle Summarized

Plan a little, code a little, test a lot, integrate often, factor the design when possible. **Don't Wait Until You Think You Are Finished Coding The Entire Project To Compile!** Trying to write the entire program before compiling a single line of code is the most frequent mistake new programmers tend to make. The best advice I can offer is don't do it! Use the programming cycle outlined above. Nothing will depress you more than seeing a million compiler errors scroll up the screen.

A Helpful Trick: Stubbing

Stubbing is a programmer's trick you can use to speed development and avoid having to write a ton of code just to get something useful to compile. Stubbing is best illustrated by example.

Say that your project requires you to display a text-based menu of program features on the screen. The user would then choose one of the menu items and press enter, thereby invoking that menu choice. What you would really like to do is write and test the menu display and choice functions without worrying about actually performing the indicated action. You can do exactly that with stubbing.

A stubbed function is a function that exists to display a simple message to the screen saying in effect "Yep, the program works great up to this point. If it were actually implemented you'd be using this feature right now!"

Stubbing is a great way to incrementally develop your project. Stubbing will change your life!

Fix The First Compiler Error First

OK. You compile some source code and it results in a slew of compiler errors. What should you do? I recommend you stay calm, take a deep breath, and fix the first compiler error first. Not the easiest compiler error, but the first compiler error. The reason is because the first error detected by the compiler, if fatal, will generate other compiler errors. Fix the first one first and you will generally find a lot of the other errors will also be resolved. If you pick an error from the middle of the pack and fix it, you may introduce more errors into your source code! Fix the first error first!

MANAGING PROJECT COMPLEXITY

Large projects differ from small projects in many ways. Large projects have more of everything: more variables, more user-defined types, more functions, more lines of code, and more complexity. There are two types of complexity: conceptual complexity and physical complexity.

Try to imagine a lot of something, like the number of dump truck loads required to move Mount Everest to North Carolina. Imagining large numbers poses a certain amount of conceptual complexity. Large software projects are very conceptually complex and many such projects end in failure because the conceptual complexity became impossible to manage. Object-oriented analysis and design (OOAD) techniques were developed to help tame conceptual complexity.

Conceptual complexity is accompanied by physical complexity. Large software development projects usually have many people working on many parts of the code at the same time. To ensure success, software developers adopt development standards. Development standards are rules developers must follow to ensure other developers can understand their work. Development standards may address issues like file naming, file location, configuration management, commenting requirements, and many, many other smart things to do to tame physical complexity.

Later in the book you will be taught how to tame conceptual complexity. This section presents you with a few smart things to do to help you manage the physical complexity of your projects. No project is too small to benefit from the techniques presented below. It is a good idea to develop good project management habits early in your programming career.

A word of warning: You could ignore the advice given here and manage to get small, simple projects to run, but if you try and structure large projects like small, simple projects, cramming all your code into one long file, you will doom yourself to failure. Formulate good programming habits now. Bad programming habits are hard to break and will end up breaking you in the long run.

Split Even Simple Projects Into Multiple Source Code Files

One of the first programming skills you must learn to help manage physical complexity is how to create multiple file projects. Your simplest programming project will have three files: a header file, an implementation file, and a main file. Larger projects will have more. As a rule of thumb you will have one header file and one implementation file for each class or abstract data type you declare. There will be only one main file which contains the main() function.

I will discuss these files and what goes into each one in more detail below, but first, I want to tell you why you want to learn the skill of developing multi-file projects. The following discussion about class interfaces may be somewhat advanced for novice readers. Fear not! Classes are discussed in great detail later in the book.

SEPARATING A CLASS'S INTERFACE FROM ITS IMPLEMENTATION

When you design a system using object-oriented techniques you model the system's functionality by identifying objects within the system and how they interact with each other. Each object will have a certain behavior associated with it, along with an interface that allows other objects to access that behavior.

An object will belong to a class of objects. A class of objects is modeled in C++ using the struct or class construct. When you declare a new user-defined type representing an object in the system you are modeling you will create a new class. In this class you will declare a set of public methods. It is this set of public methods that become the interface to objects of that class. Because the class declaration contains the prototypes for the public class interface functions, and therefore considered as the interface to class objects, you will put class declarations in header files. I will talk more about header files below.

After you have declared the interface to a class of objects you need to define class behavior. You define class behavior by implementing the class member functions you declared in the class declaration. All class member functions declared for a class will be defined in a separate implementation file. I will talk more about implementation files below too.

If all this talk of classes, objects, and interfaces makes little or no sense to you now, just hang in there. It is all covered in much greater detail later in the book.

BENEFITS OF SEPARATING INTERFACE FROM IMPLEMENTATION

You reap many benefits by declaring a class in one file and defining its behavior in another. I will talk about a few of those benefits now.

MAKES LARGE PROJECT FILE MANAGEMENT EASIER

The larger the project, the more source code files it will contain. Putting each class declaration into its own header file and its implementation in a separate implementation file allows you to adopt a simple file naming convention. Namely, name the file the same name as the class it contains suffixed by either an "h", meaning header, or "cpp", meaning C++ implementation file. Giving your files the same names as the classes they contain makes finding them among tens, hundreds, or even thousands of files a heck of a lot easier.

INCREASES PORTABILITY

Portability refers to the ability of source code to be ported to another computer system. Although seamless portability is difficult to achieve without serious prior planning, you can make it easier to achieve by keeping platform or operating system dependent code separate. Putting class declarations and implementations in separate files helps you do just that.

Allows YOU TO CREATE CLASS LIBRARIES

Putting class declarations and implementations in different files will let you create class libraries. With a class library you can share the interface to your class or classes along with the compiled implementation code. You keep the C++ source code to the implementation and thereby protect your rights to your hard work.

Helpful Preprocessor Directives

Before compiling your source code, a C++ compiler will preprocess your code. It does this by invoking a program called the preprocessor. The preprocessor performs macro substitution, conditional compilation and filename inclusion. You tell the preprocessor what to do by putting preprocessor directives in your source code.

While there are many different preprocessor directives available for your use, you need only learn four of them to help you create and manage multiple file projects and thus help you manage the physical complexity of your projects. These are `#ifndef`, `#define`, `#endif`, and `#include`. As your C++ expertise grows you will find many other uses for these directives, as well as uses for other preprocessor directives not covered in this section.

`#ifndef`, `#define`, `#endif`

You can use this combination of preprocessor directives together to help you perform conditional compilation of your header files or source code. The purpose of using these three directives in your header file is to prevent the header file and its contents from being included multiple times in a project. The reason multiple header file inclusion is not a good thing is because a header file will contain function and/or data type declarations. A function or data type declaration should be made only once in a program. Multiple declarations make compilers unhappy!

The best way to illustrate their usage is by example. The C++ source code shown in example 1.1 represents a small header file called `test.h` that declares one function prototype named `test()`.

```

#ifndef TEST_H                                     1.1 test.h
#define TEST_H

void test();

#endif

```

The `#ifndef` directive stands for “if not defined”. It is followed by an identifier, in this case `TEST_H`. The `#define` directive means exactly that, “define”. It is followed by the same identifier. The `#endif` directive stands for “end if”. It signals the end of the `#ifndef` preprocessor directive. The body of the header file appears between the `#ifndef` and `#endif` directives. This includes the `#define` directive and the function prototype `test()`.

Remember that the purpose of the preprocessor directives is to communicate with the C++ preprocessor. What will happen in this case is the preprocessor will encounter the `#ifndef` directive and its accompanying identifier. If the identifier `TEST_H` has not been previously defined then the `#define` directive will be executed next, defining `TEST_H`, followed by the declaration of `test()`.

On the other hand, if `TEST_H` has been previously defined, then everything between the `#ifndef` and `#endif` will be ignored by the preprocessor.

`#include`

Use the `#include` directive to perform file inclusion. There are essentially two ways to use the `#include` directive: `#include <filename>` and `#include "filename"`. Substitute the name of the header file you wish to include for the word *filename*.

The first usage, `#include <filename>`, will instruct the preprocessor to search in a number of directory locations as defined in your development environment. Most development environments let you customize this search sequence. If found, the entire `#include` line is replaced with the contents of *filename*.

The second usage, `#include "filename"`, acts much like the first with the usual difference of checking first for *filename* in a user default directory. If *filename* is not found in the user's default directory then the preprocessor searches a list of predefined search locations.

The Final Word on Preprocessor Directive Behavior

The behavior of many C++ language features is implementation dependent, meaning the exact behavior is left up to the compiler writer. The search paths of the `#include` directives will be different for each development environment. To learn where your compiler is searching for header files and more importantly, how to make it find your header files when you create them, consult your compiler documentation.

PROJECT FILE FORMAT

Your projects will be comprised of many header and implementation files and one main file. This section shows you the general format of each file and what goes into each one. I will use the declaration of a simple class as an example.

Header File

Example 1.2 represents the contents of a file named `firstclass.h`

```

#ifndef FIRSTCLASS_H                                1.2 firstclass.h
#define FIRSTCLASS_H

class FirstClass{
public:
    FirstClass();
    virtual ~FirstClass();

private:
    static int object_count;
};
#endif

```

Several conventions used here are worth noting. First, the name of the header file, `firstclass.h`, reflects the name of the class declaration it contains in lowercase letters with the suffix "h". Second, the identifier `FIRSTCLASS_H` is capitalized. The name of the identifier is the name of the file with the "." replaced with the underscore character "_". Doing these two simple little things makes your programming life easier by making it easy to locate your class header files and taking the guesswork out of generating identifier names for the `#ifndef` and `#define` statements.

Header files can contain other stuff besides class declarations. The following table will prove invaluable in helping you remember what you should and shouldn't put in header files.

Header Files Can Contain...	Examples
Comments	<code>// C++-style comments</code> <code>/* C-style comments */</code>
Include Directives	<code># include <helloworld.h></code> <code>#include "helloworld.h"</code>
Macro Definitions	<code>#define ARRAY_SIZE 100</code>

Table 1-1: Header File Contents

Header Files Can Contain...	Examples
Conditional Compilation Directives	<code>#ifndef FIRSTCLASS_H</code>
Name Declarations	<code>class FirstClass;</code>
Enumerations	<code>enum PenState {up, down};</code>
Constant Definitions	<code>const int ARRAY_SIZE = 100;</code>
Data Declarations	<code>extern int count;</code>
Inline Function Definitions	<code>inline static int getObjectCount(){ return object_count; }</code>
Function Declarations	<code>extern float getPay();</code>
Template Declarations	<code>template<class T> class MyClass;</code>
Template Definitions	<code>template<class T> class MyClass{ };</code>
Type Definitions	<code>class MyClass{ };</code>
Named Namespaces	<code>namespace MyNameSpace{ }</code>

Table 1-1: Header File Contents

It is just as helpful to know what you should not put in a header file. The following table offers some advice.

Header Should Not Contain...	Examples
Ordinary Function Definitions	<code>float getPay() {return itsPay; }</code>
Data Definition	<code>double d;</code>
Aggregate Definitions	<code>int my_array[] = { 3, 2, 1};</code>
Unnamed Namespaces	<code>namespace { }</code>
Exported Template Definitions	<code>export template<class T> setVal(T t) { }</code>

Table 1-2: What Not To Put In A Header File**IMPLEMENTATION FILE**

Now that `FirstClass` is declared in `firstclass.h` definitions must be given for each of the member functions. In this case there are two functions to define, the constructor, `FirstClass()` and the destructor `~FirstClass()`. C++ implementation files are suffixed with “`cpp`”. Name the implementation file the same name as the header file and add the “`cpp`” suffix to the filename. Thus, the implementation file for `FirstClass` is named `firstclass.cpp`. The code for `firstclass.cpp` is given in example 1.3.

MAIN FILE

The main file is a C++ implementation file but instead of defining class member functions it contains the `main()` function. It has the same suffix, “`cpp`”, as any other implementation file. I recommend naming this file `main.cpp`. This makes finding your main file an easy task.

```

#include "firstclass.h"
#include <iostreams>

/*****
 Initialize classwide static variables first
 *****/
*/
int object_count = 0;

/*****
 Define member functions
 *****/
*/
FirstClass::FirstClass() {
    object_count ++;
    cout<<"There is/are: " <<object_count
        <<" FirstClass object(s)!"<<endl;
}

FirstClass::~FirstClass() {
    if(--object_count) == 0)
        cout<<"Destroyed last FirstClass object!"<<endl;
    else
        cout<<"There are: " <<object_count
            <<" FirstClass objects left!"<<endl;
}

```

1.3 firstclass.cpp

```

#include "firstclass.h"

int main() {
    FirstClass f1, f2, f3;
}

```

1.4 main.cpp

That's it! Main files, and the main() function, should be kept short.

COMMENTING

A well commented program will be easier to understand by not only yourself but by others who read your code as well. There are two ways to comment source code. The first way involves adding additional, explicit comment lines to your source code by way of comment delimiters of which there are two styles: C and C++. The second way to comment your code is to write self-commenting code. This may sound complicated but it is easy to do. Besides making your code easier to read, writing self-commenting code reduces the need to rely on the first way of commenting. It also increases code reliability because you will find problems with your code easier if your code is easy to read and understand.

C-Style COMMENTS

Add C-style comments to your code by enclosing text between two sets of delimiters: “/*” and “*/”. For example:

```

/* *****/
   This is a C style comment
   *****/
*/

```

1.5 C-style comments

Everything between the `/*` and the `*/` is ignored by the compiler. Different programmers have different commenting styles. A word of advice: Programmers are often passionate about how they do business. Rise above the pettiness of arguing commenting issues with fellow programmers. Doing so is a complete waste of mental energy.

However, when using C-style comments keep a few things in mind. They are best used to insert blocks of comments. They can be used to insert one line of comments but C++-style comments are better suited for this purpose as you will see below.

I recommend aligning the `/*` and `*/` along the left margin as is shown in the example. You will be less likely to forget the `*/` and save yourself a lot of wondering why half your program doesn't compile!

Lastly, I also recommend you avoid the urge to make a cute little box out of whatever character you choose to use as a border. For example...

```

/* -----
-      This is also a C style comment      -
-                                           -
-----
*/

```

1.6 C-style comments

...only now, if you want to add a line to your comment you have to fiddle around with adding hyphens at the beginning and end of each line.

C++-STYLE COMMENTS

```
// This is a C++ style comment
```

1.7 C++-style comment

As you can see, a C++-style comment begins with two slash characters. They can appear anywhere in your program and tell the compiler to ignore everything that appears to the right up to the end of the line. Another example...

```

class TestClass{
    public: // public section
        TestClass(); // constructor
        virtual ~TestClass(); //destructor
}; //end of TestClass

```

1.8 C++ comment clutter

...shows how to use C++-style comments to really clutter up your code, which leads into a good piece of advice: use them sparingly!

To avoid the need to add comments to your source code in the first place I recommend strongly that you read the next section and take notes.

WRITE SELF-COMMENTING CODE: GIVE IDENTIFIERS MEANINGFUL NAMES

Self-commenting source code puts the joy back into programming. Self-commenting source code is easier to write, easier to read, easier to maintain, and, if you do happen to make a mistake, your mistake will be easier to find if your source code is self-commenting. How do you self-comment source code?

Essentially, you select names for identifiers that make sense in the context of your program. An identifier is a string of characters used to represent storage locations for variables, constants, functions, types, and other objects within your program.

How you form identifier names is as important as what you name them. Here's some guidance for naming variables, constants, and functions.

Variables

Use lower case letters when declaring variables. Separate each word of a multi-word identifier with an underscore character. Writing variables in lower case will make it easy to spot them in your program. Naming them something that makes sense will remind you of their purpose. The following table gives a few examples, both good and bad, of variable names.

Variable Declaration	Comment
<code>int a;</code>	Bad! What the @#%^ does “a” stand for?
<code>int mother_in_law_count;</code>	Good! Although you are counting mother-in-laws, at least you know what you are counting.
<code>Student *s[100] ;</code>	Bad! How will someone else know that s is an array of pointers to students if they don't see the declaration?
<code>Student *student_pointers[100] ;</code>	Good! Now they'll know what's supposed to be in each array element.

Table 1-3: Good vs. Bad Variable NamesCONSTANTS

Use upper case letters when defining constants. Separate each word of a multi-word constant with the underscore character. The following table offers a few examples, both good and bad, of constant names.

Constant Declaration	Comment
<code>const int a = 3;</code>	Bad! What does a stand for? Is a a variable or a constant?
<code>const int MAX_ARRAY_SIZE = 100;</code>	Good!
<code>#define object_count 25</code>	Bad! The word count sounds like it might change in the future. Because it is lower case it looks like a variable.
<code>#define MAX_OBJECT_COUNT 25</code>	Good! Now it is clear this is a constant and this is the maximum number of objects allowed.

Table 1-4: Good vs. Bad Constant NamingFUNCTIONS

Start function names with lower case letters. Join multi-word function names together and capitalize the first letter of each additional word. Functions do things. Verbs denote action. Choose function names that indicate the action the function performs. The following table gives a few examples of function names.

Function Declaration	Comment
<code>void printScreen();</code>	Good!
<code>int getObjectCount();</code>	Good!
<code>void print();</code>	Bad! Print what?
<code>void setPenPositionUp();</code>	Good! No mistaking what this function is supposed to do!

Table 1-5: Function Naming

Adopt A CONVENTION And Stick With It

The identifier naming recommendations presented here represent a convention. If you choose to adopt the styles suggested here, fine. If you don't, that's fine too. Whatever naming convention you choose to adopt I recommend you stick with it and be consistent. Don't start naming variables one way and then change the way you name them in the middle of your program. Nothing will confuse you faster than naming inconsistency.

RESTRICT THE NUMBER OF GLOBAL VARIABLES

Global variables tend to pollute the global name space and lead to the production of tightly coupled code. Tightly coupled code is bad juju, as you will learn below.

MINIMIZE COUPLING, MAXIMIZE COHESION

Repeat aloud several times; minimize coupling, maximize cohesion, minimize coupling, maximize cohesion. Good. Practice a few times on your own while I explain why you want to follow this mantra.

Coupling

Coupling refers to the degree to which each module in your source code is affected in any way by making a change to another module. Coupling can be loose, tight, or anywhere in between. You want to keep coupling as loose as possible. How does coupling occur?

One way to couple modules and not even realize you are doing it is through the reckless use of global variables. Modules can also be coupled to other modules, as is the case when one function depends on the services of another function.

It takes considerable knowledge and skill to eliminate all coupling from a group of code modules. For now, be aware that if your code is too tightly coupled, you will break it over there when you make a change here.

Cohesion

Cohesion refers to the degree to which the code in each module contributes to the purpose and function of that module. The rule of thumb is to maximize cohesion. All code belonging to a function should exist to implement that function. Don't do anything surprising or mysterious in a function because it happens to be a convenient place to do it at the time.

TEXTBOOKS, REFERENCE BOOKS, AND QUICK REFERENCE GUIDES

To be a successful C++ programmer you will need at least three books: A textbook, a language reference book, and a quick reference guide.

What you are reading is a textbook. I put a lot of thought and work into it and as a result I feel it will serve your needs as a textbook very well. However, it is not a language reference book or a quick reference guide to the C++ language. No textbook on the C++ language can be everything to everybody. The C++ standard is over 700 pages long. This book would be huge, and a huge waste of your time, if I tried to include in it everything contained in the standard. Also, when you are in the heat of programming and you just want to quickly see how to declare a class or write a for loop this book will not be the best place to turn.

If you are reading this book you have in your hands a great textbook. In the reference section below I have listed several reference books and quick reference guides I think you will find them very helpful. If it is listed in the reference section I have personally used it and wholeheartedly recommend it.

SUMMARY

The source of a student's difficulty with learning a programming language lies not with the language itself, but with the many other skills that must be mastered almost simultaneously along the way. Students will find it helpful to know the development roles they play and to have a project approach strategy.

The three development roles played by a student are those of analyst, architect, and programmer. As analyst students should strive to understand the project's requirements and what must be done to satisfy those requirements. As architect students are responsible for the design of their project. As programmer, students will implement their project's design in the C++ language.

The project approach strategy helps novice and experienced students systematically formulate solutions to programming projects. The project approach strategy deals with the following areas of concern: requirements, problem domain, language features, and design. By approaching projects in a systematic way, students put themselves in control and can maintain a sense of forward momentum during the execution of their projects. The project approach strategy can be tailored to suit individual needs.

Programming is an art. Formulating solutions to complex projects requires lots of creativity. There are certain steps students can take to stimulate their creative energy. Sketch the project design before sitting at the computer. Reserve quiet space in which to work and, if possible, have a computer dedicated to school and programming projects.

There are five steps to the programming cycle: plan, code, test, integrate, and factor. Use stubbing to test sections of source code without having to code the entire function.

There are two types of complexity: conceptual and physical. Object-oriented programming and design techniques help manage conceptual complexity. Physical complexity is managed with smart project file management techniques and by splitting projects into multiple files.

Use the `#ifndef`, `#define`, and `#endif` preprocessor directives to create header files. Use the `#include` preprocessor directive to include header files in implementation files.

Self-commenting source code is easy to read and debug. Adopt smart variable, constant, and function naming conventions and stick with them.

Minimize coupling, maximize cohesion!

This is a great textbook! Now, go get a good reference book and quick reference guide.

Skill Building Exercises

1. **Variable Naming Conventions:** Using the suggested naming conventions for variables derive a variable name for each of the concepts listed below:

Number of oranges

Required waivers

Day of week

Month

People in line

Next person

Average age

Student grades

Final grade

Key word

2. **Constant Naming Conventions:** Using the suggested naming convention for constants derive a constant name for each of the concepts listed below:

Maximum student count

Minimum employee pay

Voltage level

Required pressure

Maximum array size

Minimum course load

Carriage return

Line feed

Minimum lines

Home directory

3. **Function Naming Conventions:** Using the suggested naming convention for functions derive a function name for each of the concepts listed below:

Sort employees by pay

List student grades

Clear screen

Run monthly report

Engage clutch

Check coolant temperature

Find file

Display course listings

Display menu

Start simulation

SUGGESTED PROJECTS

1. **Feng Shui:** If you haven't already done so, stake your claim to your own quiet, private space where you will work on your programming projects. If you are planning on using the school's programming lab stop by and familiarize yourself with the surroundings.

2. **Procure and Install IDE:** If you are doing your programming on your own computer make sure you have procured and loaded an integrated development environment that will meet your programming requirements. If in doubt check with your instructor.
3. **Project Approach Strategy Checklist:** Familiarize yourself with the Project Approach Strategy Checklist in Appendix A.
4. **Obtain Reference Books:** Seek your instructor's or a friend's recommendation of any C++ reference books they think will be helpful to you during this course. There are also many good computer book review sites available on the Internet. Also, there are many excellent C++ reference books listed in the reference section of each chapter.
5. **Web Search:** Conduct a web search for C++ and object-oriented programming sites. Bookmark any site you feel might be helpful to you during this class.

SELF TEST QUESTIONS

1. List at least seven skills you must master in your studies of the C++ programming language.
2. What three development roles will you play as a student?
3. What is the purpose of the project approach strategy?
4. List and describe the four areas of concern addressed in the project approach strategy.
5. List and describe the five steps of the programming cycle.
6. What are the two types of complexity?
7. List several benefits to splitting even small projects into multiple files.
8. Discuss the concept of interface vs. implementation. How do you separate the interface of a class from its implementation?
9. What preprocessor directives can be used to allow multiple inclusion of header files?
10. List at least three things that can be contained in header files.
11. List three things that shouldn't be contained in header files.
12. Why do you think it would be helpful to write self-commenting source code?
13. What can you do in your source code to maximize cohesion?
14. What can you do in your source code to minimize coupling?

REFERENCES

International Standard. ISO/IEC 14882, *Programming Languages — C++*, First edition 1998-09-01. (This is the reference book to the C++ language. You can download it from the American National Standards Institute for a small cost and it is

worth every penny. If you are new to the language it is an extremely daunting document. I also recommend you have a fast Internet connection and an even faster printer!

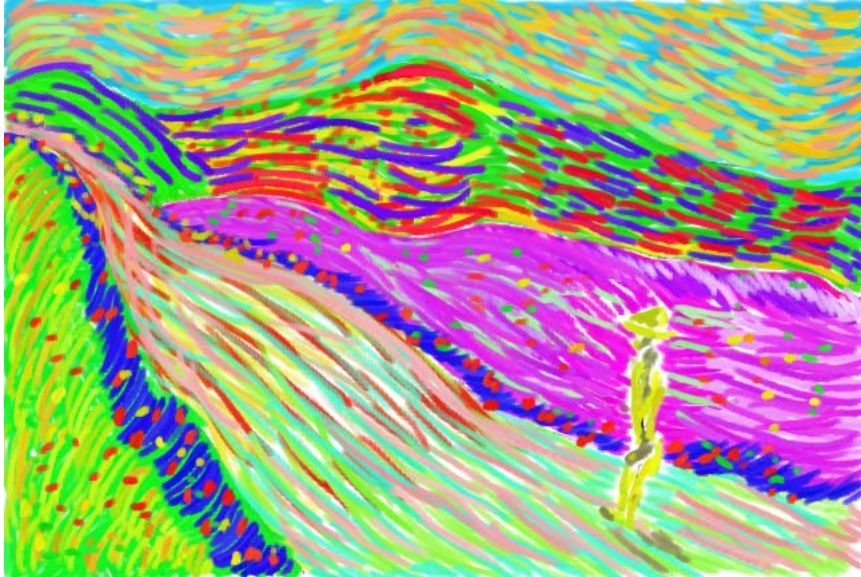
Beck, Kent. *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Reading, Massachusetts, 2000. ISBN 201-61641-6

Lucas, Paul J. *The C++ Programmer's Handbook*, Prentice Hall, Englewood Cliffs, New Jersey, 1992. ISBN 0-13-118233-1 (Great quick reference guide. I put this book in my backpack, take it to class, and show my students on the first day of every C++ class I teach. Age has done nothing to impair the usefulness of this work.)

Ellis, Margaret A., Stroustrup, Bjarne. *The Annotated C++ Reference Manual*, (a.k.a. The ARM), Addison-Wesley, Reading, Massachusetts, 1990. ISBN 0-201-51459-1 (This a great reference book. There is a second edition out now so look for it in the bookstores.)

NOTES

CHAPTER 2



Rolling Hills

SMALL VICTORIES: CREATING PROJECTS WITH IDEs

LEARNING OBJECTIVES:

- *List and describe the steps of the program creation process to include creating source code files, preprocessing, compiling, and linking*
- *List the input and output to each stage of the program creation process*
- *List and describe the primary functions of an Integrated Development Environment (IDE)*
- *Describe the concept of a project*
- *List and describe the steps required to create projects using Macintosh, Windows, and Unix development environments*
- *Demonstrate your ability to create projects in the IDE of your choice*
- *State the purpose of the UNIX make utility*
- *State the purpose of a UNIX makefile*
- *Demonstrate your ability to create and use UNIX makefiles*
- *Utilize Metrowerks CodeWarrior to create projects on Macintosh™ and PC platforms*
- *Utilize Tenon Intersystems' CodeBuilder™ to create projects on the Macintosh™ platform*
- *List and describe the similarities between different UNIX development environments*

INTRODUCTION

Before you can begin to create even simple programs you have to understand the program creation process and how to execute the process on your computer using a set of software development tools.

The kind of computer you use matters less than your choice of development tools. However, your choice of hardware platform dictates your choice of software development tools. For example, a mainframe programmer might be limited to the development tools that came with the machine but a programmer using an IBM PC™ or related computer can choose from a staggering array of development tool packages.

In this chapter I will teach you the steps of the program creation process and show you how these steps are combined in an integrated development environment (IDE). I will then show you how to use two popular IDE packages and a set of UNIX tools to write your programs.

THE PROGRAM CREATION PROCESS

Computers execute binary instructions. These binary instructions are known as machine instructions or machine code. It is difficult to program in machine code and early computer pioneers soon developed an easier way to write programs. The program creation process consists of the following steps:

- Step 1** - Write the program in a computer language humans can read and understand (like C++),
- Step 2** - Save the programs in text files. Programs can be a few lines long and reside in one file or can consist of many millions of lines of code and span thousands of files,
- Step 3** - Run the source code files through a program called a compiler to generate object code for the target computer,
- Step 4** - Run the object files through a program called a linker to produce an executable image.

These steps are illustrated in figure 2-1 below.

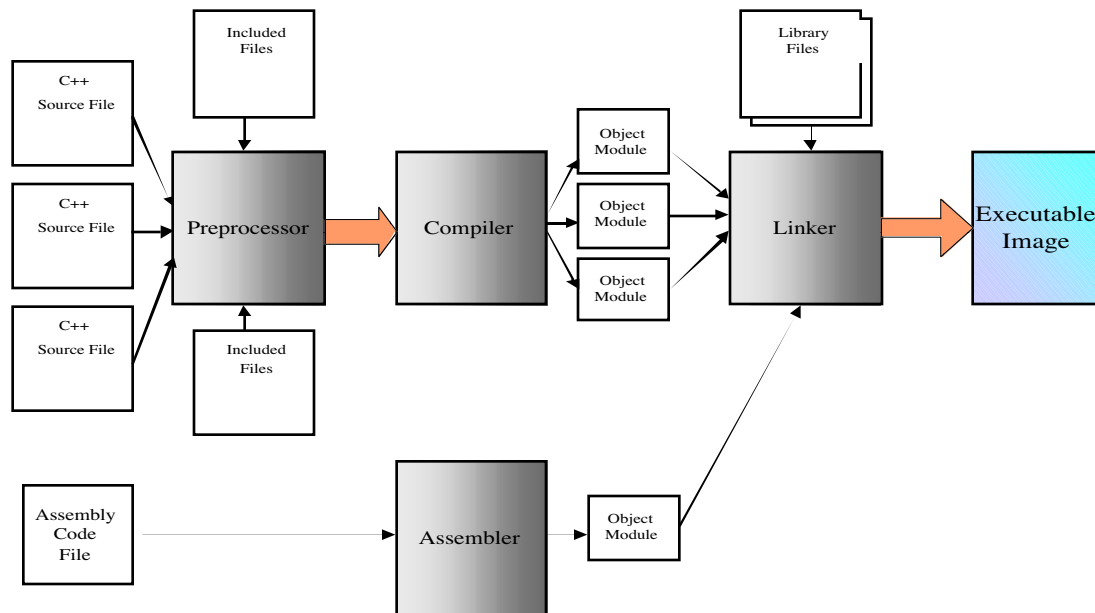


Figure 2-1: The Program Creation Process

Two other items of note are shown in figure 2-1. First, C++ adds a preprocessing step to the program creation process. The C++ preprocessor acts upon special instructions that can be contained in the C++ source code. These special preprocessor instructions are called preprocessor directives. You have already been introduced to several important and often used preprocessor directives in chapter 1. These were `#include`, `#ifndef`, `#define`, and `#endif`.

The second item of interest in figure 2-1 is the assembly code file and assembler step shown at the bottom. Program routines can be created in other languages and compiled into object modules and then later linked with object modules created with C++. This is often referred to as mixed-language programming. (see *chapter 18*) If you develop routines in other languages you will have to learn some special rules called calling conventions. Calling conventions establish responsibilities of the calling routine and the called routine. Preprocessor directives and mixed-language programming will be discussed in greater detail later in the book.

INTEGRATED DEVELOPMENT ENVIRONMENTS

To create even the smallest C++ project you will need some kind of development environment. A minimal development environment contains a text editor, compiler, and linker. Some nice-to-have components include a debugger to help you troubleshoot your source code, and a project management program to help you manage large projects. If you go the minimalist route using only a text editor, compiler, and linker you will run into difficulty when it comes time to do any serious programming. You may not need the services of the debugger but the project management software comes in really handy. An integrated development environment (IDE) contains everything you need to develop, troubleshoot, and manage software projects. In this section I will show you how to use two popular IDEs, Metrowerks CodeWarrior® and Microsoft Visual C++® to create, compile, and execute a sample project named FirstClass. Before getting started let us look at the three source code files that comprise the FirstClass project presented in chapter 1. They are `firstclass.h`, `firstclass.cpp`, and `main.cpp` and are given in examples 2.1 through 2.3.

```

#ifndef FIRSTCLASS_H
#define FIRSTCLASS_H

class FirstClass{

    public:
        FirstClass();
        virtual ~FirstClass();

    private:
        static int object_count;
};
#endif

```

2.1 *firstclass.h*

```

#include "firstclass.h"
#include <iostream.h>

int FirstClass::object_count = 0;

FirstClass::FirstClass(){
    cout<<"There are "<<++object_count<<" FirstClass objects!"<<endl;
}

FirstClass::~FirstClass(){
    if(--object_count)
        cout<<"There are "<<object_count<<" FirstClass objects!"<<endl;
    else cout<<"There are no FirstClass objects!"<<endl;
}

```

2.2 *firstclass.cpp*


```
#include "firstclass.h"

int main(){
    FirstClass f1, f2, f3, f4;

    return 0;
}
```

2.3 main.cpp

METROWERKS CodeWARRIOR

Metrowerks CodeWarrior™ is an integrated development environment available in both Macintosh™ and Windows™ versions. The screenshots below are taken from CodeWarrior™ version 5 for the Macintosh™.

The central concept in any IDE is that of the project. Start your programming projects by creating a new project in the IDE. Figure 2-2 shows CodeWarrior's New window with the Project tab selected.

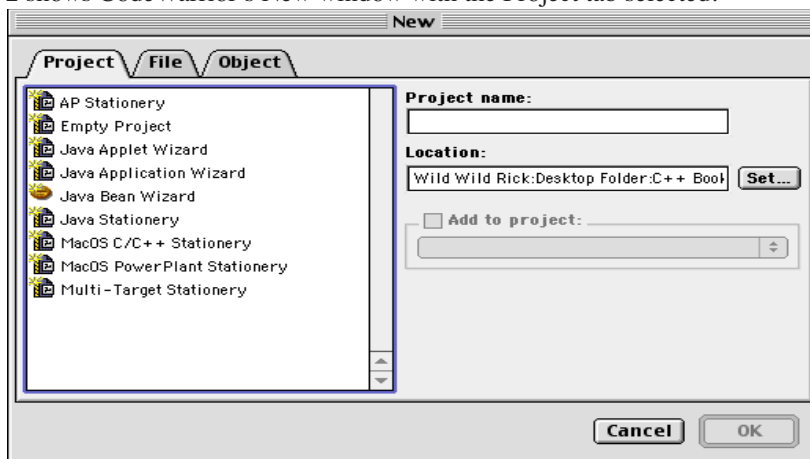


Figure 2-2: Creating a New Project in CodeWarrior

Select the project stationery you wish to use. In figure 2-3 MacOS C/C++ Stationery is selected. Type the name of the project in the Project name box.

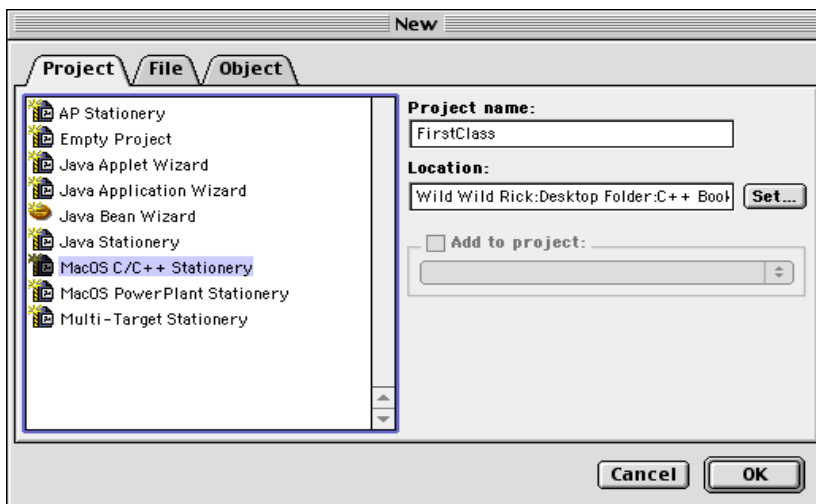


Figure 2-3: Selecting Stationery and Naming Project

Before clicking OK check to ensure you have set the proper location for your project. I recommend creating a new folder called CodeWarrior Projects and creating individual folders for each of your projects within that folder. Setting up your programming environment is a matter of personal taste. Figure 2-4 shows the window that pops up when you press the Set... button located to the right of the Location text box. (shown in figure 2-3)

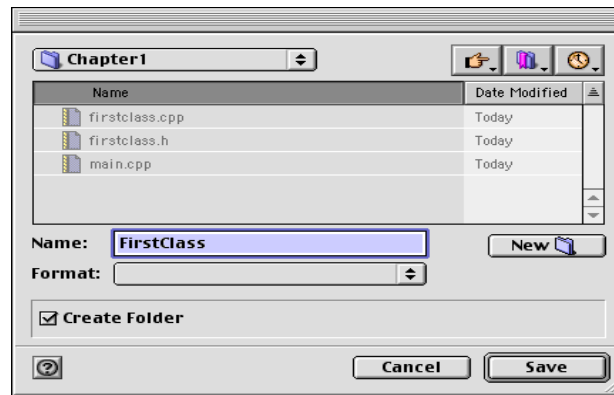


Figure 2-4: Setting a Project's Location

Figure 2-4 also shows the project named FirstClass being saved to a folder called Chapter1. If you wish to save your project to a location other than the CodeWarrior default location make the necessary changes and click Save. Otherwise, click Cancel to return to the previous window. After you have named your project, and set its location, click OK (see figure 2-3). You will now be presented with a window similar to figure 2-5 where you will select the type of project you want to create.

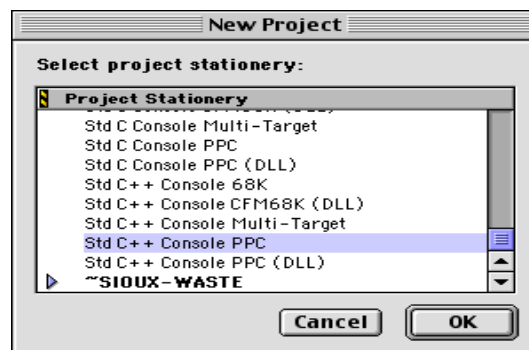


Figure 2-5: Select Project Type

There are a lot of choices but since we are creating a C++ project and are interested only in simple stream I/O to a console, select the Std C++ Console PPC. PPC stands for PowerPC. As you can see you can create projects for 68K Macs as well. CodeWarrior for Windows has lots of different project types for Windows programmers too. Once the project type is highlighted click OK. This will create the project and present you with the CodeWarrior project window that looks like figure 2-6.

Folders called Groups are created automatically by CodeWarrior. The first group contains an automatically generated C++ source file called HelloWorld.cp. See figure 2-7. You can compile the project now and see the output generated by HelloWorld.cp just to make sure everything is working correctly.

Now it is time to create the three source files, firstclass.h, firstclass.cpp, and main.cpp. Select New Text File from the File menu as shown in figure 2-8 and enter the source code for firstclass.h as shown in figure 2-9.

When you have finished with each file, save it to the project's folder. When you have finished with all three you then need to add firstclass.cpp and main.cpp to the Sources Group in the FirstClass project window.

It is important to note here that in CodeWarrior you only add implementation files to the Sources Group, that is, files with the .cpp or .cp extension. Do not add header files. Simply save them to the project folder and the compiler

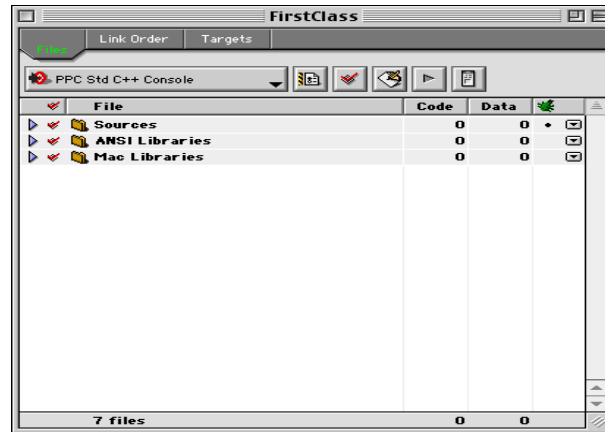


Figure 2-6: FirstClass Project Window

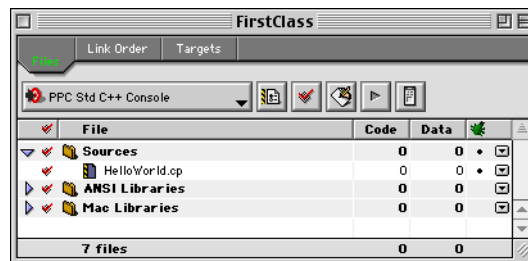


Figure 2-7: Sources Group Open Revealing HelloWorld.cp

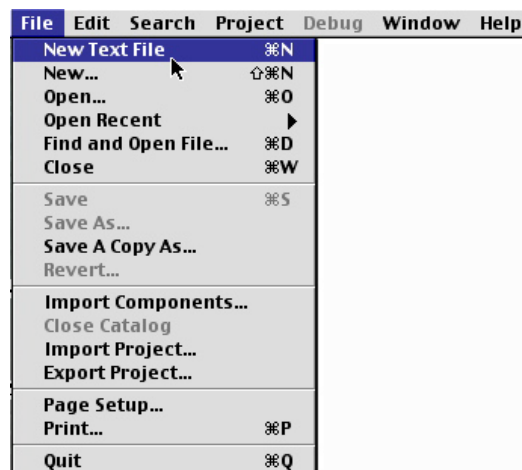


Figure 2-8: Creating New Text File

will search for the header files there. If you organize your projects differently, as would be the case for complex projects, you will have to explicitly set project search paths.

To add files to the FirstClass project select the Sources group in the project window and then select Add Files... from the Project menu. If you have saved your files in the correct folder you will see them listed in a window similar to figure 2-10. Select the firstclass.cpp and main.cpp files and click Open. Next, remove HelloWorld.cp from the Sources group. When you have finished adding the files to the project your FirstClass project window will look like figure 2-11. Now you are ready to compile and run the project.

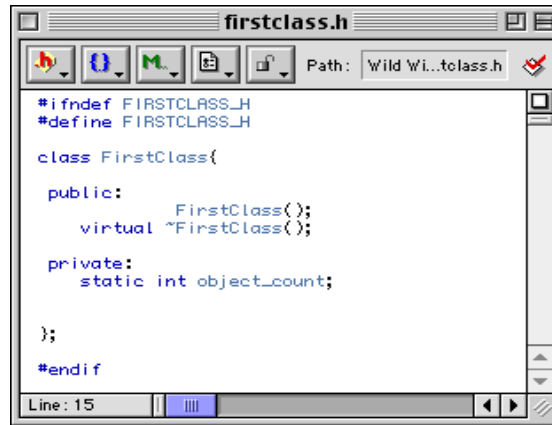


Figure 2-9: Editing firstclass.h

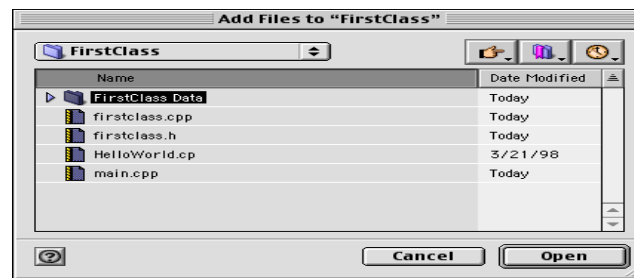


Figure 2-10: Adding Files to Project

Select Run from the Project menu. CodeWarrior will then compile and link each file in the project automatically.



Figure 2-11: firstclass.cpp and main.cpp Added, HelloWorld.cp Removed

When finished, it will launch the resulting executable file. The results of running the FirstClass project are shown in figure 2-12.

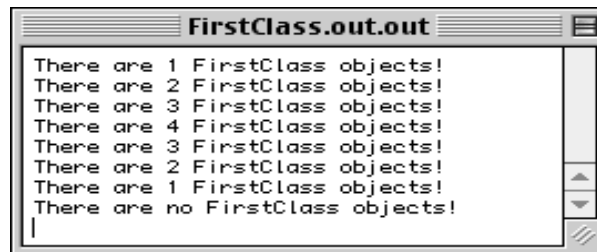


Figure 2-12: FirstClass Project Output

MICROSOFT VISUAL C++

Of all integrated development environments Microsoft Visual C++ is the most difficult for students to learn. I think the primary reason for this is due in part to its power and sheer number of features. However, if you are planning to do any serious Windows programming I recommend mastering this tool.

All IDEs are similar in that they are project oriented. Visual C++ uses the workspace metaphor. Begin your project by creating a new workspace and selecting the project type. Figure 2-13 shows the Win32 Console Applica-

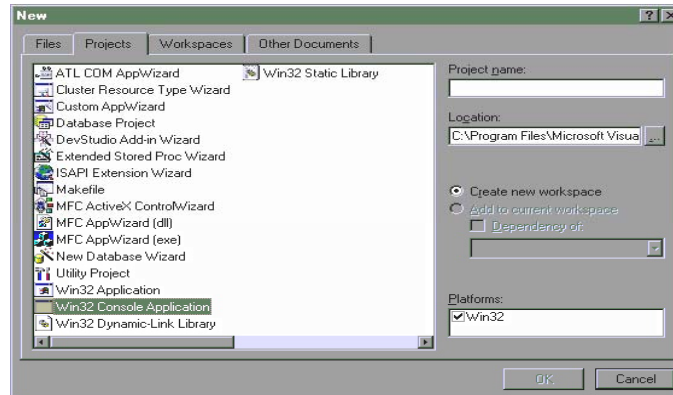


Figure 2-13: Creating New Visual C++ Project

tion project type highlighted. Enter a name for your project in the Project name text box as shown in figure 2-14. The Create new workspace radio button is automatically selected. I have named this project Project1.

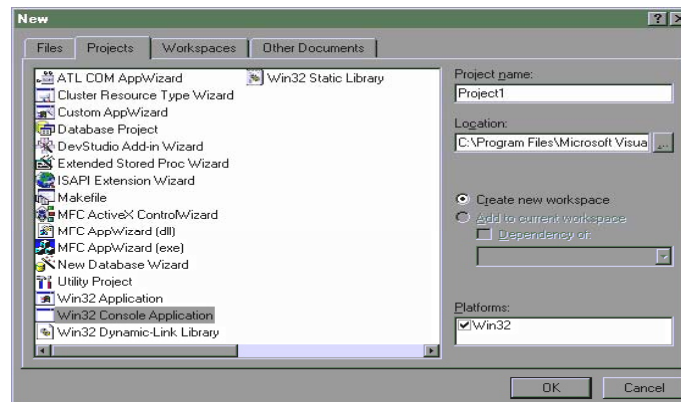


Figure 2-14: Naming the Project

Once you have named your project click OK. Next, choose what kind of Console Application you wish to create. (see figure 2-15) Click the “A simple application” radio button and then click Finish.

After setting the Console Application type the New Project Information window will appear looking similar to figure 2-16. If you agree with the information it contains click OK.

Now you are presented with the Workspace environment. Refer to figure 2-17.

Visual C++ automatically creates a file with the same name as the project name you entered into the Project name text box. It puts the main() function in this file. Edit this file so that it looks like example 2.3 but leave in the line containing `#include "StdAfx.h"`. Figure 2-18 shows the edited file.

Visual C++ differs from CodeWarrior in that header files are contained in the workspace in a group called Header Files. Let us add the firstclass.h file. Select New... from the File menu. A window similar to figure 2-19 will appear.

Enter the filename firstclass.h in the File name text box as shown in figure 2-20 and click OK. Visual C++ will now open an empty text file ready for editing. Enter the code from firstclass.h given in example 2.1. Figure 2-21 shows the completed firstclass.h file. Visual C++ will automatically put new header files in the Header Files group.

Now create the file firstclass.cpp. Select New... from the File menu and highlight C++ Source File. Refer to figure 2-22.

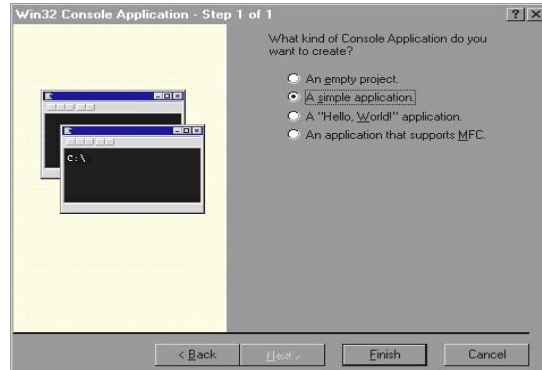


Figure 2-15: Selecting Console Application Type

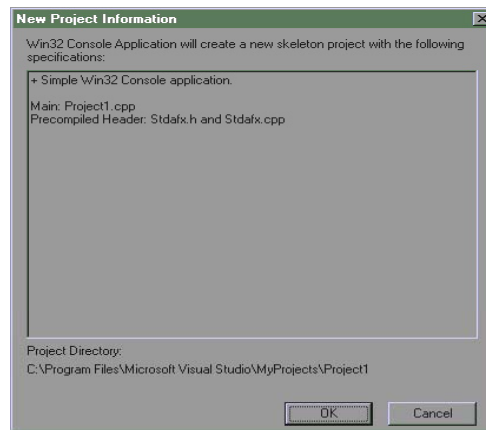


Figure 2-16: New Project Information Window

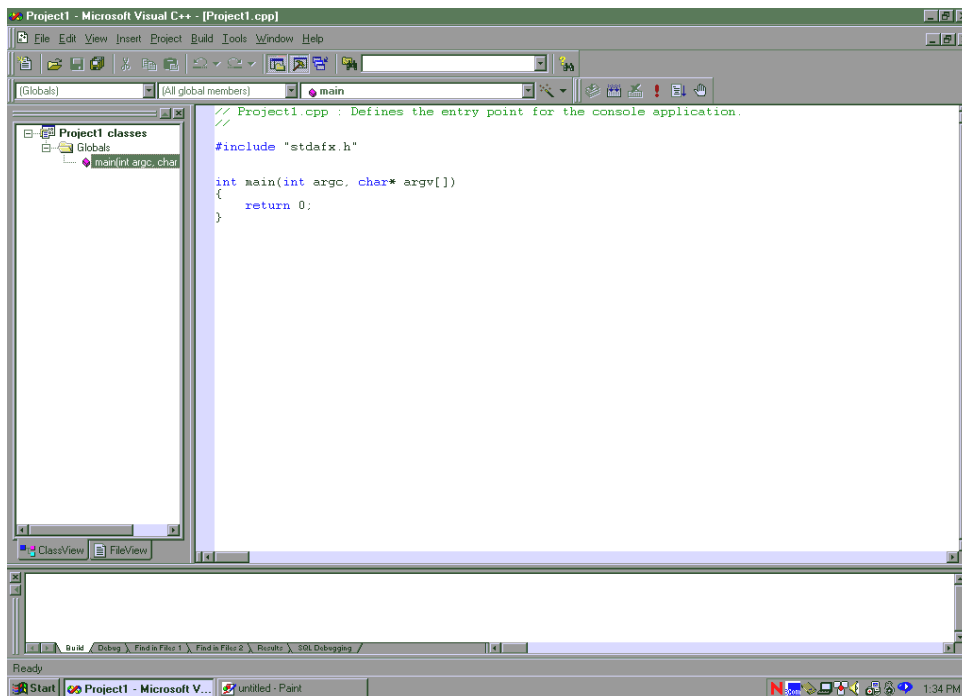


Figure 2-17: Workspace Environment with ClassView Selected

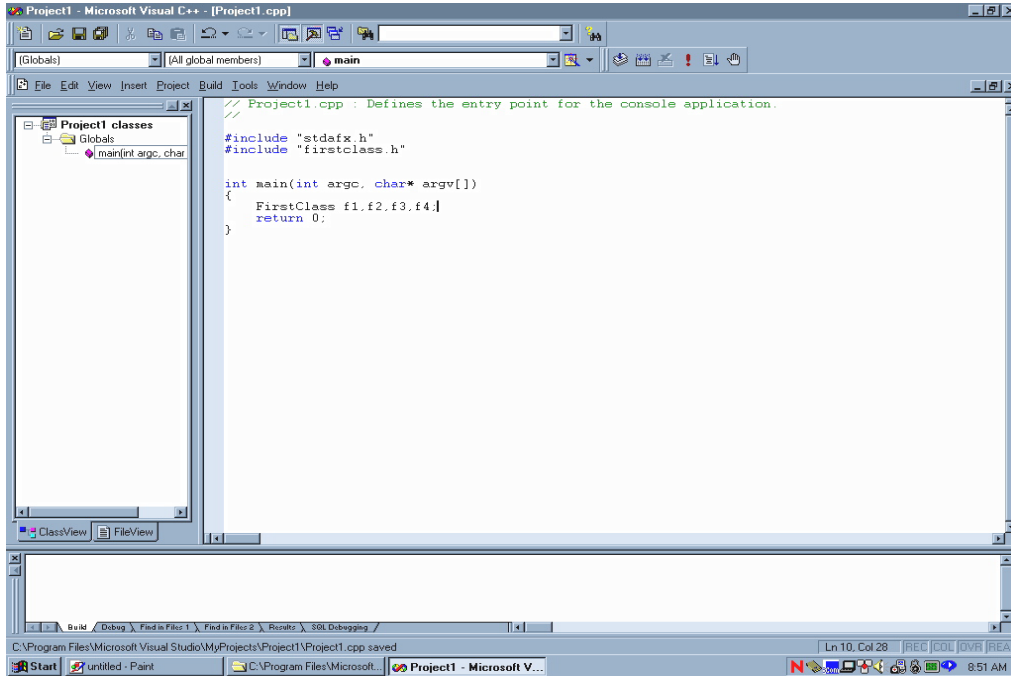


Figure 2-18: Edited Project1.cpp File

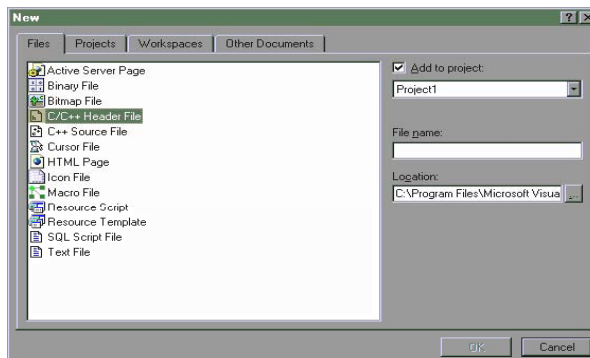


Figure 2-19: Adding New C++ Header File to Project 1

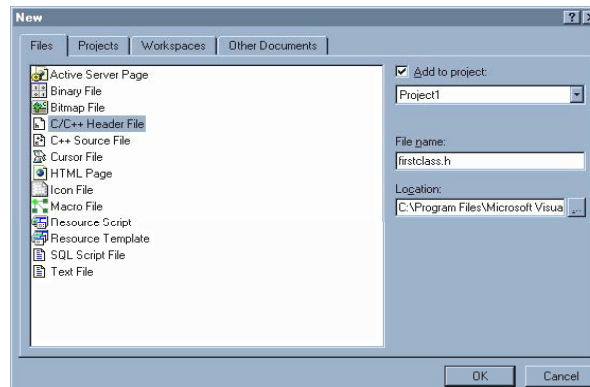


Figure 2-20: File Name Entered

Enter the name in the File name text box and click OK. It will automatically be saved in the Source Files group. Enter the code for firstclass.cpp from example 2.2 and save.

One thing left to do. Open the StdAfx.h file and add the line `#include "firstclass.h"` underneath the comment `// TODO: reference additional headers your program requires here`". Figure 2-23 shows the StdAfx.h file being edited.

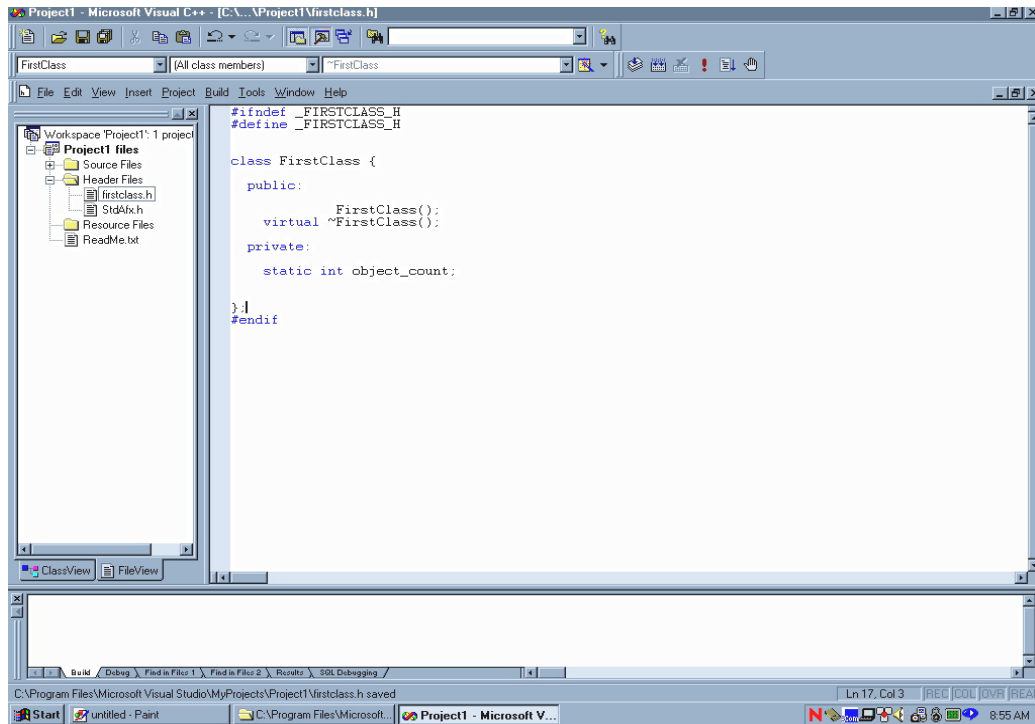


Figure 2-21: Editing firstclass.h

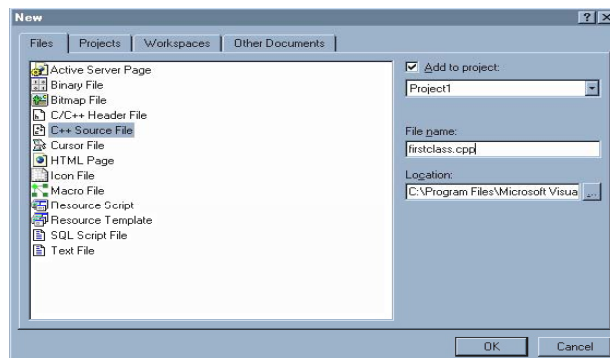


Figure 2-22: Creating a New C++ Source File

When you have finished editing the StdAfx.h file save the changes. You are now ready to execute the project.

Select Execute from the Build menu and click OK to dismiss the dialog that pops up asking you if you would like to create Project1.exe. Referring again to figure 2-23, as Visual C++ performs each stage of the compilation and build process on the project it will write a series of messages to the Build window (*refer to the series of tabbed windows along the bottom of the workspace*). In figure 2-23 you see the last of the messages, Linking..., and the report that Project1.exe was built with 0 errors and 0 warnings. Figure 2-24 shows the results of running Project1.exe.

INTERMISSION

So far I have shown you how to create simple, multi-file projects with two popular IDEs. As you use your IDE of choice to create projects of increasing complexity you will dive deeper into its features. A word of advice: Read the documentation that comes with your IDE. The information in this chapter is helpful but it is not a complete treatment of all the cool things your IDE can do.

A large part of the power of an IDE comes from its ability to track changes to dependent project files. For example, if a change is made to firstclass.h, every file that depends on the contents of firstclass.h will have to be recompiled. This chore is handled automatically for you by CodeWarrior™ and Visual C++. If you choose not to use an IDE

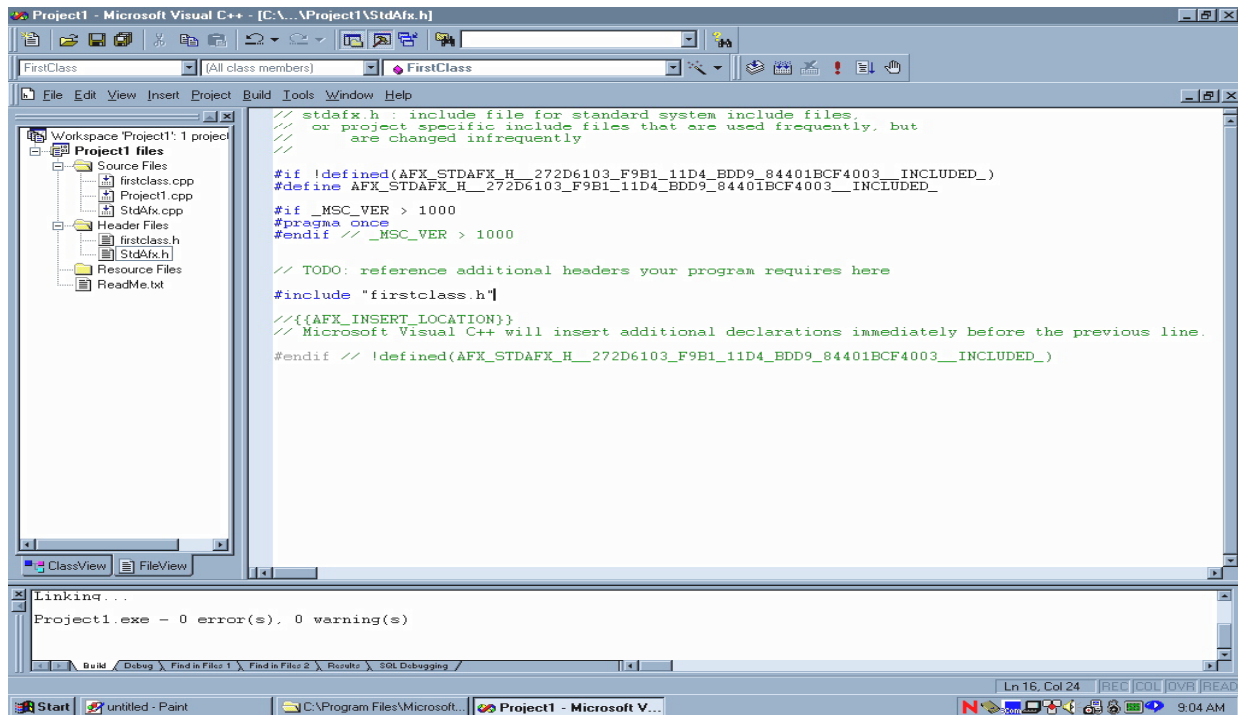


Figure 2-23: Linking...Message and Results of Building Project 1

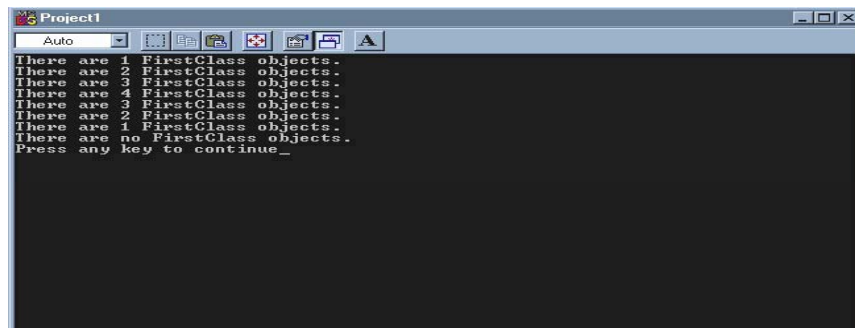


Figure 2-24: Running Project1.exe

you will have to learn how to use a utility like make to handle project management tasks. Let us take a look at a UNIX development environment and see the make utility in action.

TENON INTERSYSTEMS MACHTEN CodeBuilder™

CodeBuilder™ is a UNIX programming environment that runs on Macintosh™ computers. It comes with C, C++, Ada, Objective-C, Fortran, and Java compilers. CodeBuilder can be used to develop code for Macintosh™ PowerPC machines as well as Silicon Graphics, SUN, NeXT, or HP workstations.

CodeBuilder is not an integrated development environment, meaning that although it comes with all kinds of powerful development tools they are not tightly integrated like CodeWarrior or Visual C++. You will have to choose an editor with which to create your source files, learn how to call the compiler from the command line, and learn to use the make utility to help you manage multi-file projects.

ATTENTION LINUX USERS

If you have a PC running Linux the information in this section will help you too. Having said that, let's get started.

ORGANIZING PROJECT FILES

You can still apply the concept of a project to your UNIX development. In its simplest form a project is one or more related files located in one or more directories along with a makefile that defines the dependencies between the related files. Makefiles and the make utility will be covered below. Get into the habit of creating different directories for each project and work on that project from that directory.

CREATING SOURCE FILES

Select your UNIX text editor of choice to create your source files. CodeBuilder™ ships with Emacs, a powerful text editor ported to many different versions of UNIX. Figure 2-25 shows Emacs running in an X-Windows session being used to create firstclass.h.

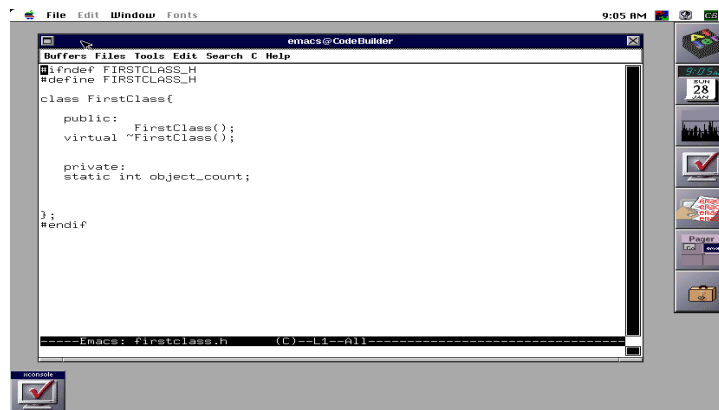


Figure 2-25: Creating firstclass.h with Emacs

Save firstclass.h in your designated project directory and create firstclass.cpp and main.cpp in similar fashion.

CREATING MAKEFILE

The next file you need to create is a file called makefile. The makefile contains commands that tell the make utility how to build your project. Like an IDE, the make utility will detect when project files have been modified and recompile all project files that depend on those files. Figure 2-26 shows Emacs being used to create the makefile.

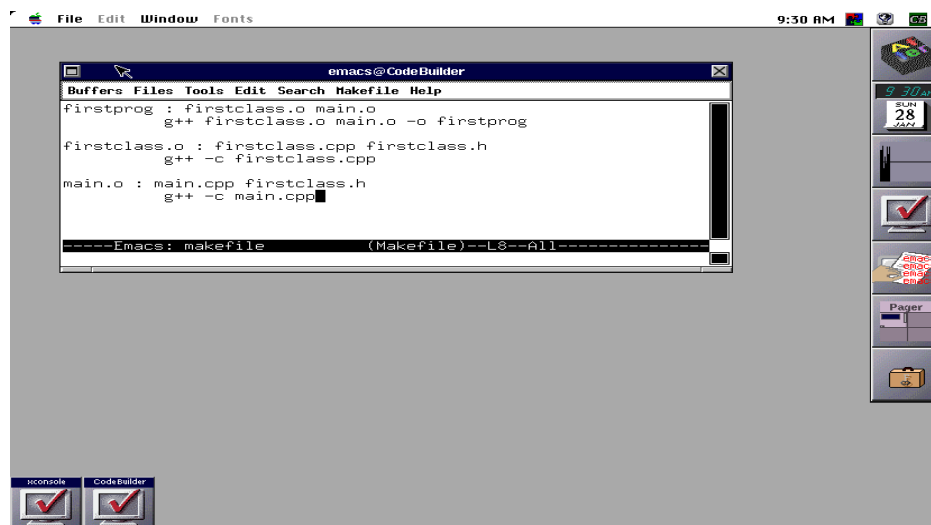


Figure 2-26: Creating makefile with Emacs

Let us take a closer look at the contents of makefile.

2.4 makefile

```

1 firstprog : firstclass.o main.o ← firstprog depends on firstclass.o &
2     g++ firstclass.o main.o -o firstprog     main.o
3
4 firstclass.o : firstclass.cpp firstclass.h ← firstclass.o depends on firstclass.cpp &
5     g++ -c firstclass.cpp                     firstclass.h
6
7 main.o : main.cpp firstclass.h ← main.o depends on main.cpp & first-
8     g++ -c main.cpp                             class.h

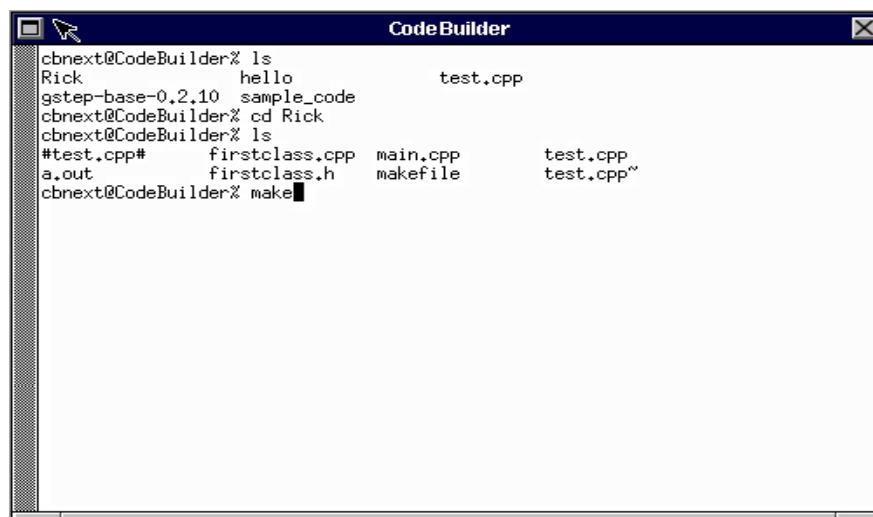
```

The first line of makefile says that the program named `firstprog` depends on two object files named `firstclass.o` and `main.o`. The second line is a GNU C++ compiler command that will link the two files `firstclass.o` and `main.o` to produce an executable named `firstprog`.

Line 4 of makefile says that `firstclass.o` depends on `firstclass.cpp` and `firstclass.h`. Should either of these two files change, the make utility will have to recompile `firstclass.cpp` to produce a new version of `firstclass.o`. This will cause `firstclass.o` to change which will cause the linking process to be invoked again to create the new version of `firstprog`. On line 5 the C++ compiler command is called with the `-c` flag. This will result in compilation only and no linking, since linking is done on line 2.

Line 7 says that `main.o` depends on `main.cpp` and `firstclass.h`. Should either of these two files change `main.o` will change which will cause the same chain reaction as described in the paragraph above. Notice how a change to `firstclass.h` will cause both `firstclass.o` and `main.o` to change.

To run the make utility simply type `make` at the command line. Refer to figure 2-27. Notice too in figure 2-27



```

CodeBuilder
cbnext@CodeBuilder% ls
Rick          hello          test.cpp
gstep-base-0,2,10  sample_code
cbnext@CodeBuilder% cd Rick
cbnext@CodeBuilder% ls
#test.cpp#    firstclass.cpp  main.cpp      test.cpp
a.out        firstclass.h    makefile      test.cpp~
cbnext@CodeBuilder% make

```

Figure 2-27: Running the make Utility

that the project files are located in a directory named `Rick`. A listing of that directory reveals the three project files `firstclass.h`, `firstclass.cpp`, `main.cpp`, and `makefile`, along with a few files left over from a `g++` test run.

Figure 2-28 shows the results of running the make utility as well as running the resulting executable.

```

Rick      hello      test.cpp
gstep-base-0,2,10  sample_code
cbnext@CodeBuilder% cd Rick
cbnext@CodeBuilder% ls
#test.cpp#      firstclass.cpp  main.cpp      test.cpp
a.out           firstclass.h    makefile      test.cpp^
cbnext@CodeBuilder% make
g++ -c firstclass.cpp
g++ -c main.cpp
g++ firstclass.o main.o -o firstprog
cbnext@CodeBuilder% ls
#test.cpp#      firstclass.h    main.cpp      test.cpp
a.out           firstclass.o    main.o        test.cpp^
firstclass.cpp  firstprog      makefile
cbnext@CodeBuilder% firstprog
There are 1 FirstClass objects.
There are 2 FirstClass objects.
There are 3 FirstClass objects.
There are 4 FirstClass objects.
There are 3 FirstClass objects.
There are 2 FirstClass objects.
There are 1 FirstClass objects.
There are no FirstClass objects.
cbnext@CodeBuilder%

```

Each g++ compiler command is echoed to the console...

New files...

Execute firstprog...

Figure 2-28: Results of Executing make Utility and firstprog

SUMMARY

This chapter introduced you to the program creation process and gave you an overview of how to create multifile projects in three different C++ development environments. It also explained the advantages and features of integrated development environments.

If your particular IDE was not explicitly covered here you will find that they all work basically the same way. You begin by creating a project and selecting a project type and then create or add your source files to the project.

Skill Building Exercises

1. **Create Multi-File Project:** Using your integrated development environment of choice create a multifile project using the three files `firstclass.h`, `firstclass.cpp`, and `main.cpp` listed in this chapter. Compile and run the project.
2. **Debugging:** Use the debugging facility to step through the execution of the project you created above. Learn how to set execution breakpoints and step into functions to trace execution.
3. **IDE Directory Structure:** Draw the directory structure of your integrated development environment. Locate the folders or directories where the library and header files are located.

SUGGESTED PROJECTS

1. **Read IDE Documentation:** Locate and read the documentation included with your integrated development environment.
2. **Obtain DOS Reference:** If you are new to programming and are programming on a PC running Microsoft Windows, find a good book on DOS and learn a few important commands.
3. **Obtain UNIX Reference:** If you are using Mac OSX or another UNIX-based operating system find a good book on the UNIX operating system.

SELF TEST QUESTIONS

1. Describe the program creation process.
2. What is the purpose of the C++ preprocessor?
3. What is the purpose of the compiler?
4. What is the purpose of the linker?
5. What is the primary benefit of using an integrated development environment?
6. List at least three features of an integrated development environment.
7. What is the purpose of the UNIX make utility?

REFERENCES

Metrowerks CodeWarrior Reference Documentation for Windows 95/98/NT and Apple Macintosh.

Tenon Intersystems Reference Documentation for MachTEN Unix.

Tenon Intersystems Reference Documentation for CodeBuilder™ .

Microsoft Visual C++ Reference Documentation

NOTES

CHAPTER 3



Seaside Rendezvous

PROJECT WALKTHROUGH: AN EXTENDED EXAMPLE

LEARNING OBJECTIVES

- *Apply the project approach strategy to help you systematically implement a program that satisfies the requirements of a given project specification*
- *Iteratively apply the development cycle to help you implement your programming projects*
- *List and describe the phases of the Project Approach Strategy*
- *List and describe the steps of the software development cycle*
- *List and describe the different development roles performed during the development cycle*
- *Translate a project specification into a software design that can be implemented in C++*
- *Implement a software design in C++ using a functional decomposition approach*
- *List and describe the steps involved with functional decomposition*
- *Describe how the development cycle can be employed in a tight spiral fashion*
- *State the importance of compiling and testing early during the development process*

INTRODUCTION

This chapter will walk through the creation of a programming project using the project approach strategy and development cycle discussed in chapter 1. The ideas presented here should not be considered dogmatic. I fully expect that as you gain confidence and experience as a developer you will formulate your own style of problem solving. I also expect that readers new to C++ may not understand all the language features utilized in this chapter. Don't worry. What I want you to gain from reading this material is an understanding of how to tackle a project, analyze it, design a solution, and implement the design. You can, and should, revisit different sections of this chapter as you progress through the text and build upon your C++ programming skills.

The approach I take in this chapter is procedural, meaning I am going to show you how to functionally decompose a problem and craft its solution from the viewpoint of functions rather than objects. I take this approach because even though you are learning C++ with the desire to become a competent object-oriented programmer, to do so requires you to understand fully procedural programming concepts. A sound understanding of procedural concepts will significantly help you when it comes time to design class functions.

THE PROJECT APPROACH STRATEGY

The project approach strategy areas discussed in chapter 1 are summarized in table 3-1 below. Keep these strategy areas in mind as you formulate your solution to a programming project. The purpose of having a project approach strategy is to kick start the creative process and perpetuate your creative momentum. I remind you once again that you can tailor this approach strategy to suit your individual taste. Modify it in any way you see fit.

Strategy Area	Explanation
Requirements	Determine and clarify exactly what purpose and features the finished project must have. Clarify your understanding of the requirements with your instructor if the project specification is not clear. <i>The result of pursuing this strategy area should be a clear definition of what problem must be solved.</i>
Problem Domain	Study the problem until you have a firm understanding of how to solve it. Optionally, express your understanding of the solution by writing a pseudocode algorithm that describes, step-by-step, how the problem can be solved. <i>The result of this strategy area should be a high-level solution statement that can be translated into a detailed application design.</i>
Language Features	Make a list of all the language features you must understand and use to draft a competent design and later implement your design. As you study each language feature check it off your list. Doing so will give you a sense of progress. <i>The result of this strategy area should be a complete understanding of all C++ language features required to effect a good design and solve the problem.</i>
Design (Plan)	Sketch out a rough application design. The design should address issues such as data structures, Input/Output, and how you plan to execute the problem solution you derived in the Problem Domain strategy area. <i>The result of this strategy area will be a clear understanding of what source code should be written.</i>

Table 3-1: Project Approach Strategy

THE DEVELOPMENT CYCLE

When you move into the design phase of your project you will start to employ the development cycle. It is good to have a broad, macro-level design idea to get you started, but don't make the mistake of trying to design everything up front. Design until you can begin coding and test some of your design ideas. The development cycle is summarized in the following table.

Development Cycle Step	Explanation
Plan	Do enough design to get you started with the implementation. Do not attempt to design everything up front. The idea here is to keep your design flexible and open to change.
Code	Implement what you have designed.
Test	Thoroughly test each section or module of source code. The idea here is to try and break it before it has a chance to break your application. Even in small projects you will find yourself writing little test case programs on the side to test something you have just finished programming.
Integrate	Add the tested piece of the application to the rest of the project.
Refactor	This step applies more to object-oriented programming than to procedural programming. It means to take a comprehensive look at your overall application architecture and migrate general functionality up into base, or even abstract, classes so the functionality can be utilized by more concrete derived classes.

Table 3-2: Development Cycle

The development cycle will be employed in a tight spiral fashion as depicted in figure 3-1. By tight spiral I mean you will begin with the plan step, followed by the code step, followed by the test step, followed by the integrate step, optionally followed by the factor step. Once you have finished a little piece of the project in this fashion, you go back to the Plan step and repeat the process. Each complete plan, code, test, integrate, and factor sequence is referred to as an iteration. As you iterate through the cycle you will begin to notice the time it takes to complete the cycle from the beginning of the plan step to the completion of the integrate step decreases. The development cycle spirals tighter and tighter as development progresses until you converge on the final solution.

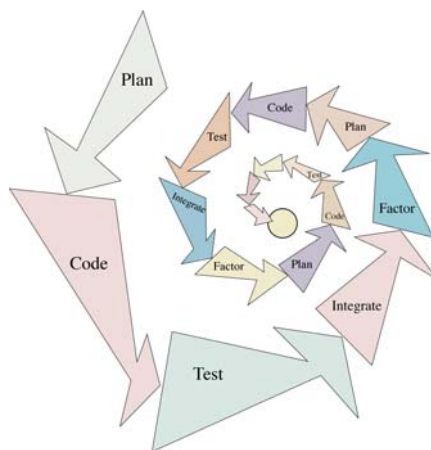


Figure 3-1: Tight Spiral Development Cycle Deployment

THE PROJECT SPECIFICATION

Keeping both the project approach strategy and development cycle in mind, let us look now at a typical project specification.

IST 156 Project 1 Robot Rat
<p>Objectives:</p> <p>Demonstrate your ability to utilize the following language features:</p> <ul style="list-style-type: none"> Arrays Program flow control structures Variables Constants Functions Simple iostream input and output Enumerated types Preprocessing directives <p>Demonstrate your ability to create multi-file projects.</p> <p>Task:</p> <p>You are in command of a robot rat! You will control the rat's movements around a 20 x 20 grid floor. The robot rat is equipped with a pen. The pen has two possible positions, up or down. When in the up position, the robot rat can move about the floor without leaving a mark. If the pen is down then as the robot rat moves through each grid it leaves a mark. Moving the robot rat about the floor with the pen up or down at various locations will result in a pattern. Write a C++ console program to control your robot rat.</p> <p>Hints:</p> <p>The robot rat can move in four directions: north, east, south, and west. Implement the floor as a two dimensional array of one of the following types: bool, int, or char. (Note: Depending on the type you choose for the array is a design decision which will affect how you implement various other features of your program.)</p> <p>At minimum, provide a text-based command menu with the following or similar command choices:</p> <ol style="list-style-type: none"> 1. Pen Up 2. Pen Down 3. Turn Right 4. Turn Left 5. Move Forward 6. Print Floor 7. Exit

Table 3-3: Project Specification

Another valid requirements question might focus on exactly what is meant by a multifile project. Since I personally feel it is extremely important for students to learn from the beginning how to create multifile projects I answer this question by clarifying the need for this project to be split between three files. You can name them anything you desire but I usually suggest the following file names: robot.h, robot.cpp, and main.cpp. The header file, robot.h, will contain all the function prototypes and any constant and enumerated type declarations you require to implement your project. The robot.cpp file will contain function definitions for functions declared in the robot.h file and any file scope variables deemed necessary. Finally, the main.cpp file will contain only the main() function which I recommend be kept as brief as possible.

What about error checking? Good question. In the real world, making sure an application behaves well under extreme user conditions and recovers gracefully in the event of some catastrophe consumes the majority of the programming effort. One area in particular that requires measures to ensure everything goes well is array processing. As the robot rat is moved around the floor care must be taken to prevent the program from letting it go beyond the bounds of the floor array.

Something else to consider is how to process a user's command. Since the project only calls for simple iostream input and output I recommend treating everything as a char on the input. Otherwise, I want you to concentrate on learning how to use fundamental language features as listed in the objectives section, so I promise not to try to break your program. For the purposes of this project it is safe to assume the user is perfect yet noting for the record that this is absolutely not the case in the real world!

Summarizing the requirements thus far:

- You are to write a program that models the movement of a robot rat around a floor,
- The robot rat is an abstraction represented by a collection of attributes, (I will discuss these attributes in the problem domain and design strategy areas)
- The floor is represented in the program as a two dimensional array of either bool, int, or char,
- Use just enough error checking, focusing on staying within the array boundaries,
- Assume the user is perfect,
- Read user command input as char,
- Split the project into three files.

Problem Domain

In this strategy area your objective is to learn as much as possible about what a robot rat is and how it works in order to gain insight into how to proceed with the project. A good technique to use to help jump-start your creativity is to go through the project specification and look for relevant nouns and verbs or verb phrases. A first pass at this activity will yield two lists. The list of nouns will suggest possible attributes or data structures and the list of verbs will suggest possible actions or functions required to implement the project.

Nouns & Verbs

A first pass at reviewing the project specification yields the following table of nouns and verbs.

Nouns	Verbs
robot rat	move
floor	set pen up
pen	set pen down
pen position (up, down)	mark
mark	turn right
program	turn left
pattern	print floor
direction (north, south, east, west)	exit
menu	

Table 3-4: Robot Rat Nouns and Verbs

This is a good starting list, and now that you have it, what should you do with it? Good question. As mentioned above, each noun is a possible candidate for either a variable, a constant, or some other data structure. Some nouns will not be used. Others will have a direct relationship to some data structure you might use to implement the program. Still, other nouns will look like they could be very useful but do not easily convert or map to a data structure. This seems to be the problem in this case.

The list of verbs come mostly from the suggested menu. Verbs will normally map directly to functions you will need to create as you write your program. The functions, which are derived from the verbs, will use the data structures which are derived from the noun list. Note here that this use, or manipulation, of data structures by functions exemplifies the procedural programming paradigm.

With the list of nouns gleaned from this project specification it appears as though you will have to do a little more analysis of the robot rat problem to see if you can come up with any more attribute candidates. I recommend taking a closer look at the noun robot rat. Just what is a robot rat from the attribute perspective? Since pictures are always helpful I suggest drawing a few. Here's one for your consideration.

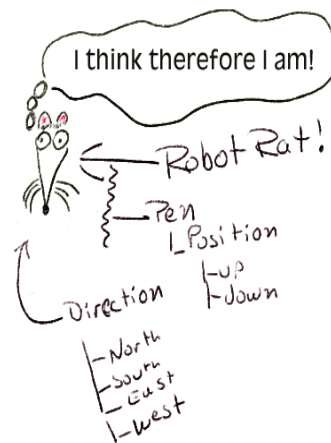


Figure 3-2: Robot Rat Viewed As Attributes

It looks like this picture suggests that a robot rat, as defined with the current list of nouns, consists of a pen which has two possible positions and the rat's direction. As described in the project specification and illustrated in figure 3-2, the pen can be either up or down. Regarding the robot rat's direction, it can face one of four ways: north, south, east, or west. Can more attributes be derived? Perhaps another picture will yield more information. I recommend drawing a picture of the floor and run through a few robot rat movement scenarios.

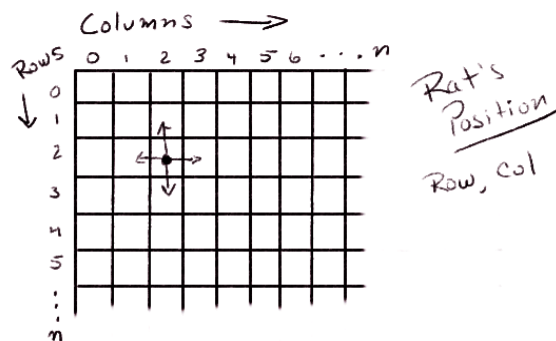


Figure 3-3: Robot Rat Floor Sketch

Figure 3-3 offers a lot of information about the workings of a robot rat. The floor is represented by a collection of cells arranged by rows and columns. As the robot rat is moved about the floor its current position on the floor can be determined by keeping track of its current row and column. These two nouns are good candidates to add to the list of relevant nouns and to the set of attributes that can be used to describe a robot rat. Before the robot rat can be moved its current position on the floor must be determined and upon completion of each move its current position must be updated.

We now have a better understanding of what attributes are required to represent a robot rat as illustrated in figure 3-4.

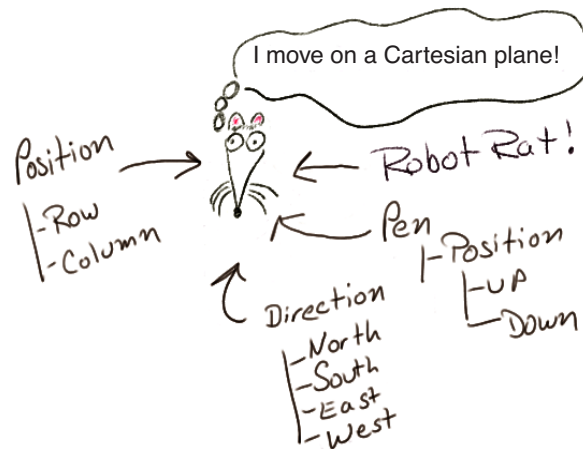


Figure 3-4: Complete Robot Rat Attributes

This seems to be a sufficient analysis of the problem at this point. You can return to this strategy area at any time should further analysis be required. It is now time to take a look at what language features must be understood to implement the procedure oriented solution.

LANGUAGE FEATURES

Let us pause a moment to review your progress. You have received a project specification. You clarified the requirements and studied the problem to be solved. You have now arrived at the most critical, and difficult, stage in the project approach strategy. You are at a point where you are to proceed with the design of the program but if you are new to C++, you haven't yet mastered, or perhaps even learned about, some of the language features required to start the design process.

Without the aid of the project approach strategy most students come to a complete halt right about here. They get overwhelmed because they do not yet know how to speak C++ effectively. It is a lot like being in a foreign country and knowing what you want to say but not having the language skills necessary to say what you are thinking.

The language features strategy area serves an important function in the overall project approach strategy: to sustain your sense of progress momentum. Take this time to list the language features you need to learn and check them off as you learn them.

In this case, the project specification gives you a good start. Refer to the project objectives first and then to any language features identified in the requirements and problem domain strategy areas. Generate a study checkoff list and check each language feature off the list as you complete your study of each feature. The following checkoff list could be used to study the language features for the robot rat project.

Checkoff	Language Feature
	Arrays, Multi-dimensional arrays: declaring, defining, initializing, processing
	Program flow control structures: while, do/while, for, if/else, switch/case
	Variables: declaring, defining, scoping, (limiting scope to file)
	Constants: declaring, defining
	Functions: declaring, defining, return types, argument passing,
	Simple I/O streams: cout, cin
	Enumerated types: declaring, defining, using
	preprocessor directives: #ifndef, #define, #endif
	Native language types: char, int, bool

Table 3-5: Language Feature Study Checkoff List For Robot Rat Project

When you have completed your study of the required language features you are ready to enter the design strategy area.

DESIGN (FIRST ITERATION)

The Design strategy area marks your entry into the development cycle. The objective here, in the first iteration, is to map out a macro-level design architecture with which to begin building your application. Design to the point where you can start coding. Since you will be applying the development cycle iteratively, as depicted in figure 3-1, you will revisit this strategy area upon entry into each iteration of the development cycle.

A good place to start is to state or describe the flow of the program and the actions you want it to perform using natural language statements referred to as pseudocode. Example 3.1 shows what the pseudocode might look like that describes how the robot rat program should run.

```

display menu
get user's menu choice
process user's menu choice
if user selects pen up
    change the rats pen position to up
if user selects pen down
    change the rats pen position to down
if user selects turn right
    change rats direction right
if user selects turn left
    change rats direction left
if user selects move forward
    move rat
if user selects print floor
    print floor pattern
if user selects exit
    exit the program

```

3.1 Robot Rat Pseudocode

Example 3-1 leaves out a lot of detail but that's O.K., the details will be added as the design progresses. If you compare example 3-1 with the robot rat project specification you will see most of its content derives from the menu description. Three statements have been added to indicate the need to display the menu, get the user's menu choice,

and process the menu choice. Stating the solution to a programming problem in terms of the highest-level functional module with the intention of refining the program by identifying and defining sub modules later in the design is a classic example of top-down functional decomposition. Figure 3-5 illustrates functional decomposition.

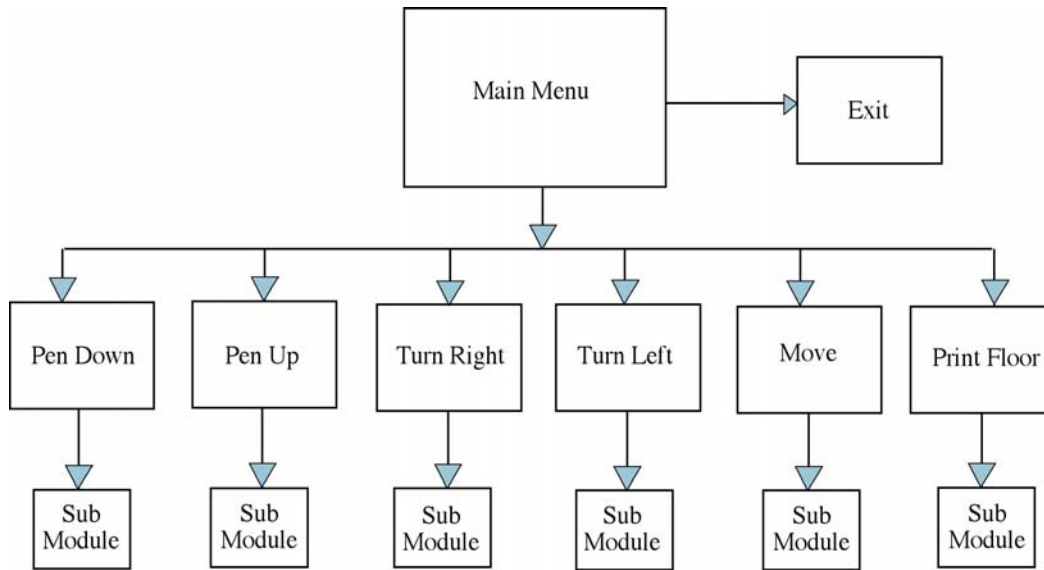


Figure 3-5: Functional Decomposition of Robot Rat Program

Notice how the arrows in figure 3-5 point downward from high-level program modules to lower-level modules. This tells you that the functionality of Main Menu depends on the functionality of modules Pen Down, Pen Up, Turn Right, Turn Left, Move, and Exit. These submodules, in turn, may depend on further submodules for their functionality. This is the dependency relation associated with the procedural programming paradigm.

At this point you are probably comfortable with “what” the robot rat program must do. It is now time to consider “how” you will get the program to do what it is designed to do. For example, how will you physically organize your program files? In what files will you locate various parts of your program? You don’t need to come up with all the answers up front, rather, you only need to lay out a foundation to get you going.

Table 3-6 lists some design considerations and the resulting decisions. This first attempt at design should take you to the point of being able to compile your project and test a particular feature. In any project it is a good idea to start by implementing the user interface (UI), which, in this case, is a text-based menu as described in the project specification and described in the pseudocode listing.

Design Consideration	Design Decision
Multifile project	Create a project in the Integrated Development Environment with the following files: robot.h, robot.cpp, & main.cpp.
display menu	Write a function called displayMenu() to display the menu on the screen

Table 3-6: First Iteration Feature Set

This is a good place to stop the first iteration of the design and move to the implementation phase of the development cycle.

IMPLEMENTATION (FIRST ITERATION)

In the first iteration of the implementation phase you will execute the two design considerations listed in Table 3-6. Figure 3-6 gives an overview of the process using Metrowerks CodeWarrior™.

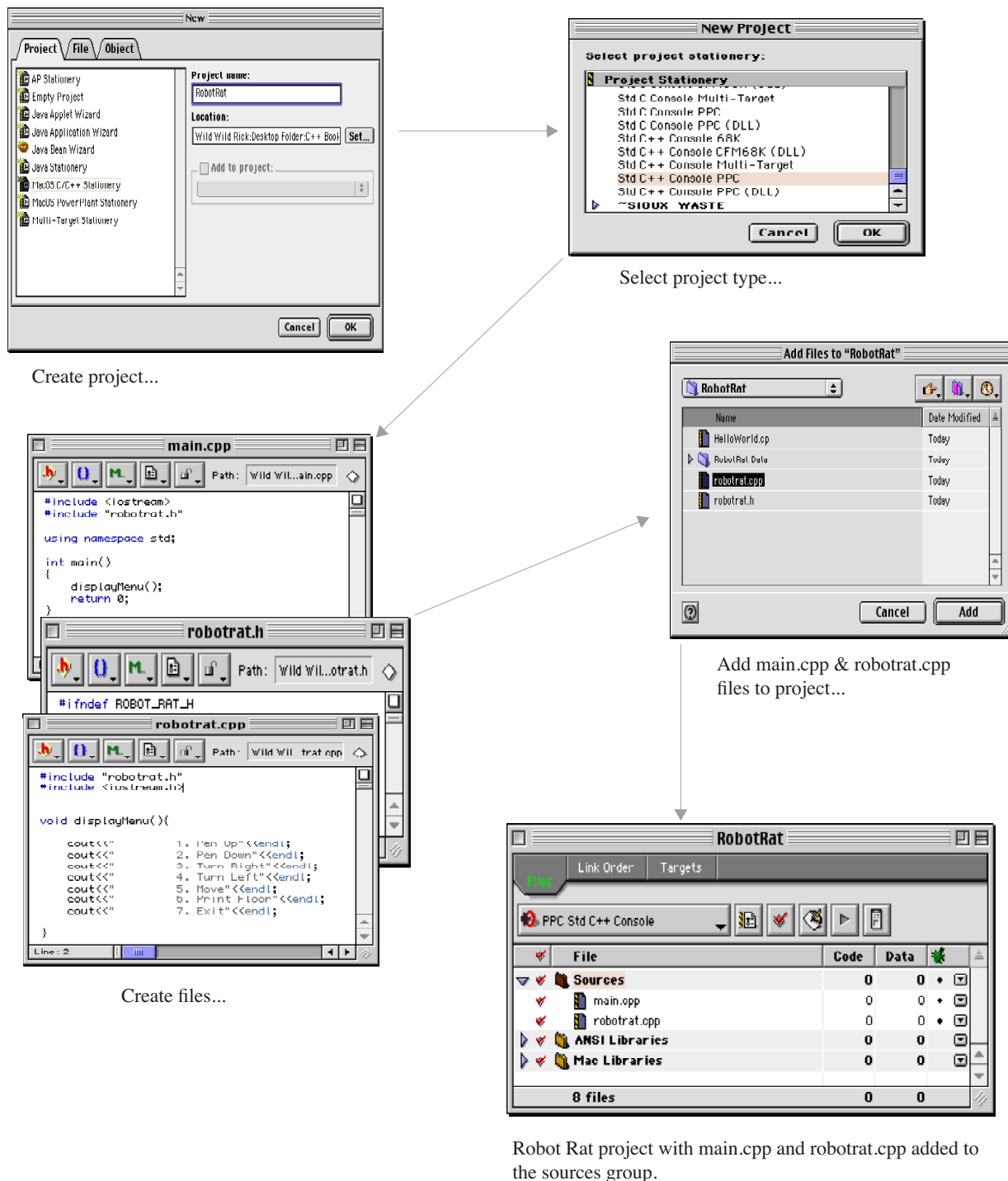


Figure 3-6: Overview of Project Creation Process

First, create the RobotRat project and select the desired project type. Since the project requires only simple ios-stream input and output the project will be a standard C++ console application.

Once the project is created the three files, robotrat.h, robotrat.cpp, and main.cpp must be created and added to the project. Creating the project and giving it a three-file structure lays a solid foundation for continued, smooth development. Splitting the program into three files may seem at first to make things unnecessarily complicated, however, it is much easier to deal with this small, increased level of organizational complexity at the start of a project than to try and split a project into multiple files later in the development cycle.

What goes in each file at this early stage? Since you are concerned with implementing the menu you need only concentrate on declaring the displayMenu() function, defining the displayMenu() function, and then using or calling the displayMenu() function somewhere in the program. Put the function declaration in the robotrat.h file. The code will look like figure 3-7.

```

robotrat.h
Path: Wild Wil...otrat.h

#ifndef ROBOT_RAT_H
#define ROBOT_RAT_H

void displayMenu();

#endif
  
```

Annotations in the image point to the preprocessor directives and the function declaration.

Figure 3-7: robotrat.h

Notice the preprocessor directives. Don't forget to use them in your header files to prevent multiple inclusion.

With the robotrat.h file complete you can now create the robotrat.cpp file. The purpose of this file is to define or implement the displayMenu() function declared in the robotrat.h file. The code for robotrat.cpp will look like figure 3-8.

```

robotrat.cpp
Path: Wild Wil...rat.cpp

#include "robotrat.h"
#include <iostream.h>

void displayMenu(){

    cout<<"      1. Pen Up"<<endl;
    cout<<"      2. Pen Down"<<endl;
    cout<<"      3. Turn Right"<<endl;
    cout<<"      4. Turn Left"<<endl;
    cout<<"      5. Move"<<endl;
    cout<<"      6. Print Floor"<<endl;
    cout<<"      7. Exit"<<endl;

}
  
```

Annotations in the image point to the include directives and the implementation of the displayMenu() function.

Figure 3-8: robotrat.cpp

As shown in figure 3-8, the displayMenu() function simply writes some menu choices to the console. That's all it does. Hence its name...displayMenu(). This is an example of a highly cohesive function. Cohesion and coupling is covered in detail later in the book, but for now, keep in mind that it is good design practice to keep the functionality of program modules focused to what it is they are supposed to do. In this case, displayMenu(), as its name implies, will display the menu on the screen. When the time comes to get the user's menu choice and process the user's menu choice you will need to create functions for those purposes.

Now that the displayMenu() function has been both declared and defined it is time to use it someplace. That place is the main() function. The main() function is located in the main.cpp file as shown in figure 3-9.

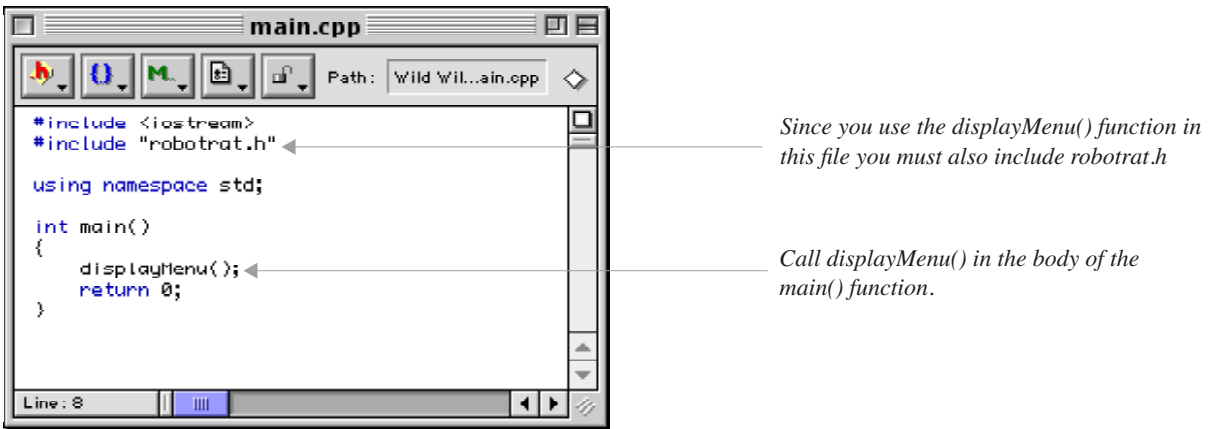


Figure 3-9: main.cpp

Every C++ program needs a `main()` function. The `main()` function represents the start of the first instruction of the robot rat program in memory. It is now time to test robot rat.

TESTING (FIRST ITERATION)

If, and there's always an if in programming, everything goes well the work completed on robot rat so far should compile and display the menu. Even though this sounds like small beans, getting the program to this point has taken considerable thought and effort.

The objective of testing the `displayMenu()` function is to see if the menu choices do get written to the screen as expected. Perhaps the most important reason for programming and testing little pieces of the program at a time is that it allows you to catch errors early in the development cycle. These same errors, if left to be discovered later, will be a whole lot harder to correct.

Compiling and running the robot rat project gives the result shown in figure 3-10.

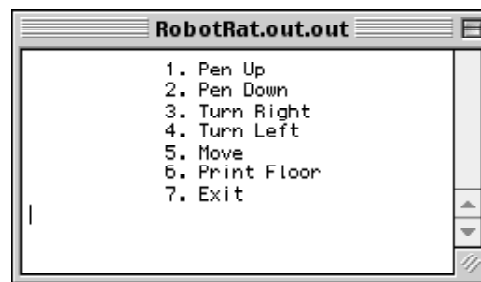


Figure 3-10: Robot Rat Menu

INTEGRATION (FIRST ITERATION)

Everything looks as if it runs fine. The menu displays on the screen and the program exits. That's all it does and that is all it is supposed to do. There is not much to integrate at this point since you started the project off on a good footing by splitting it into separate files. The `displayMenu()` function is located in the `main()`. It is O. K. to leave it there for now, and it may stay there for one or two more iterations of the development cycle, or at least until it makes sense to move it into another function as the program grows.

At this point you have come to the end of the first iteration of the development cycle. It is now time to return to the design phase and start the second iteration.

DESIGN (SECOND ITERATION)

You have completed the first iteration. You have made great progress on robot rat. The project is nicely organized into three files, the project compiles flawlessly, and the `displayMenu()` function writes the menu to the screen as expected. To proceed, you must now select one, or more, program features to implement that sustain your development effort momentum.

Since you have just completed the implementation and testing of the menu feature of robot rat it makes sense to proceed with adding the capability to accept and process user menu choices. Table 3-7 lists the features to design and implement.

Design Consideration	Design Decision
Accept user input for menu selection	Read user input from console using <code>iostreams</code> . Store user's input in a variable for later processing. Read the input as a <code>char</code> .
Process user input; determine which menu choice selected	<p>Compare input value against a set of constant values representing menu choices. Use <code>switch/case</code> statement to implement the comparison. Use function stubbing for testing purposes to defer detailed functional development.</p> <p><code>processMenuChoice()</code> will be the name of the function used to process the user's menu choice.</p> <p>Stub the following functions: <code>setPenUp()</code>, <code>setPenDown()</code>, <code>turnRight()</code>, <code>turnLeft()</code>, <code>move()</code>, <code>printFloor()</code></p> <p>Implement the following functions: <code>doDefault()</code>, <code>programExit()</code></p>

Table 3-7: Second Iteration Feature Set

This iteration of the development cycle is a critical one. Here you are attempting to implement an extremely critical piece of the robot rat program without knowing much, or anything at all, about how the subfunctions will ultimately be implemented. Specifically, you are going to implement the menu processing capability of the program that will let a user enter a menu choice for further processing, but you haven't yet written the code to set the pen up or down, or to turn the robot rat left or right. Luckily, there's an old programmer's trick you can use to help in just this situation. It is called function stubbing.

FUNCTION STUBBING

Function stubbing is the technique of writing functions with little or no substance and is an invaluable program testing tool. If a stubbed function contains any code at all it is usually just a simple message written to the screen indicating to the programmer that the function was called. This lets the programmer know that everything in the program worked fine up to the point of the function call.

OTHER CONSIDERATIONS

When you tested robot rat at the end of the first iteration the program exited immediately after calling the `displayMenu()` function. This was normal behavior for the program at that time. But now that you are going to implement the menu processing feature you will need to keep the program running until the user selects exit from the robot rat menu. It is a good time to use pseudocode again to generally describe the behavior of the program to help guide you in your design. Example 3.2 gives the pseudocode for how processing should occur.

3.2 Pseudocode For Processing
User Menu Choices

```

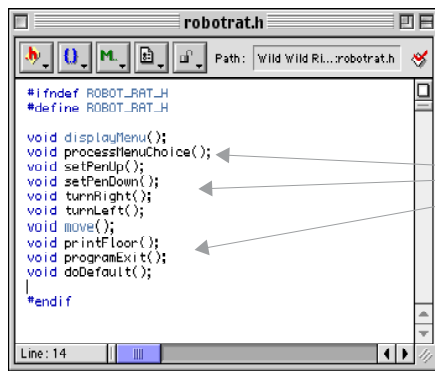
repeat
    display menu
    process user menu choice
    execute user menu choice
until user selects exit

```

Armed with a small set of features to implement and an idea of how to implement them you are now ready to move into the second iteration of the implementation phase.

IMPLEMENTATION (SECOND ITERATION)

The best place to start is in the `robotrat.h` header file. Edit the file and add the declarations for all the new functions you will need for this iteration. Figure 3-11 shows the `robotrat.h` file containing the new function declarations.



Add function declarations for all functions needed in second iteration.

Figure 3-11: `robotrat.h`

Next, edit the `robotrat.cpp` file and implement all the new functions you have just declared in the `robotrat.h` file. Start by implementing the stubbed functions first. Example 3.3 gives you the source code for `robotrat.cpp` with the functions implemented.

3.3 `robotrat.cpp`

```

#include "robotrat.h"
#include <iostream.h>
#include <stdlib.h>          //need stdlib.h for exit() function

void displayMenu() {

    cout<<"    1. Pen Up"<<endl;
    cout<<"    2. Pen Down"<<endl;
    cout<<"    3. Turn Right"<<endl;
    cout<<"    4. Turn Left"<<endl;
    cout<<"    5. Move"<<endl;
    cout<<"    6. Print Floor"<<endl;
    cout<<"    7. Exit"<<endl;
}

void setPenUp() {
    cout<<"The pen is up!"<<endl;
}

```

3.3 robotrat.cpp continued

```
void setPenDown() {
    cout<<"The pen is down!"<<endl;
}

void turnRight() {
    cout<<"Robot Rat turned right!"<<endl;
}

void turnLeft() {
    cout<<"Robot Rat turned left!"<<endl;
}

void move() {
    cout<<"Robot Rat moved!"<<endl;
}

void printFloor() {
    cout<<"Floor printed!"<<endl;
}

void programExit() {
    exit(0);
}

void doDefault() {
    cout<<"Please Enter A Valid Menu Choice: "<<endl;
}

void processMenuChoice() {

    char input = '0';
    cout<<"Please Enter Menu Choice: ";
    cin>>input;

    switch(input) {
        case '1': setPenUp();
                 break;
        case '2': setPenDown();
                 break;
        case '3': turnRight();
                 break;
        case '4': turnLeft();
                 break;
        case '5': move();
                 break;
        case '6': printFloor();
                 break;
        case '7': programExit();
        default : doDefault();
    } //end switch case
} //end processMenuChocie()
```

Figure 3-12 shows the contents of the main.cpp file. The main() function needs to be changed slightly to implement the program operation described in the pseudocode of example 3.2.

One way to loop forever...

```

#include <iostream>
#include "robotrat.h"

using namespace std;

int main()
{
    for(;;){
        displayMenu();
        processMenuChoice();
    }
    return 0;
}
    
```

Figure 3-12: main.cpp

Once all the additions are complete it is time to move on to testing.

TESTING (SECOND ITERATION)

Figure 3-13 shows the results of running robot rat and selecting menu choices 1 through 7. Each menu choice results in the execution of the corresponding function stub as evidenced by the message printed to the screen. The only thing left to be tested is the default case. In other words, what happens when a user enters a choice that's not on the menu? The default case in the switch statement along with the doDefault() function will handle bad user menu choices. Figure 3-14 shows the results of that test.

```

1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 1
The pen is up!
1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 2
The pen is down!
1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 3
Robot Rat turned right!
1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 4
Robot Rat turned left!
1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 5
Robot Rat moved!
1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 6
Floor printed!
1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 7
    
```

Figure 3-13: Test Results

```

1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 8
Please Enter A Valid Menu
Choice:
1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: A
Please Enter A Valid Menu
Choice:
1. Pen Up
2. Pen Down
3. Turn Right
4. Turn Left
5. Move
6. Print Floor
7. Exit
Please Enter Menu Choice: 7
    
```

8 not on menu!

A not on menu!

7 works fine...

Figure 3-14: Default Case Test

INTEGRATION (SECOND ITERATION)

Again, there's nothing to explicitly integrate. Actually, the act of integration has been taking place simultaneously with implementation. The program is well structured, making the addition of functionality easier than if the structure, or framework, of the program had been poorly designed.

Since robot rat tests are satisfactory it is time to return to the design phase and start the third iteration of the development cycle.

DESIGN (THIRD ITERATION)

With the menu processing functionality in place it is time to start adding meat to the program by implementing some of the data structures the robot rat will need to operate. The floor seems like a good place to focus development effort. Table 3-8 lists the features to be implemented during this iteration.

Design Consideration	Design Decision
The floor. What data type to use? How should each element be initialized? What scope should the floor array have?	The floor will be a two dimensional array of boolean. The floor array will have all elements initialized to false at the start of the program. The floor array will have static file scope in robotrat.cpp so that it is visible to all functions needing access to it.
Ensure variable names declared for use in robot rat don't conflict with variables names declared in the std namespace.	Put all variable declarations in a namespace called robotrat.
Print the floor pattern	Implement the printFloor() function to print the floor pattern when the user selects the Print Floor menu choice.
When the robot rat moves through a floor position with the pen down, how will the mark be recorded and preserved for future moves and floor printings?	If the robot rat's movement takes it through an array element and the pen is down, the boolean value of the array element will be changed to true.

Table 3-8: Third Iteration Feature Set

The floor is a critical data structure in the robot rat program. Design decisions regarding the floor will impact future development. How do you know if the design decision you make regarding the floor is good or bad? Good question. Just like real life, you will not know if you have made a good or bad design decision until you progress a little further with development. If you have to violate your program design architecture to fit something in then the design is less than optimal. Good design feels good, works good, and is easy to change without breaking things unexpectedly.

IMPLEMENTATION (THIRD ITERATION)

Proceed with the first three design decisions as described in table 3-8. The fourth design consideration can be evaluated after these are completed.

The first thing to do is to declare the floor array. Since it is going to have file scope you can declare it at the top of the file right above the displayMenu() function. Figure 3-15 shows the robotrat.cpp file with the necessary code added.

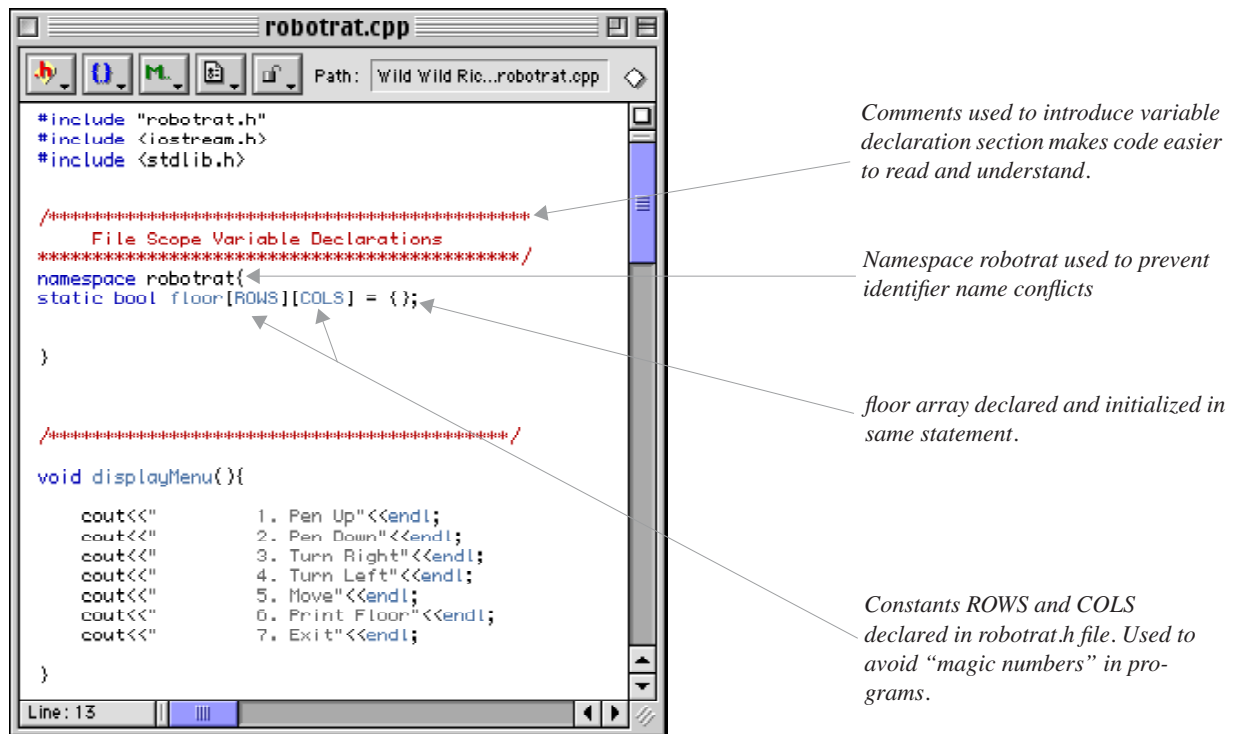


Figure 3-15: robotrat.cpp with Floor Array Declaration

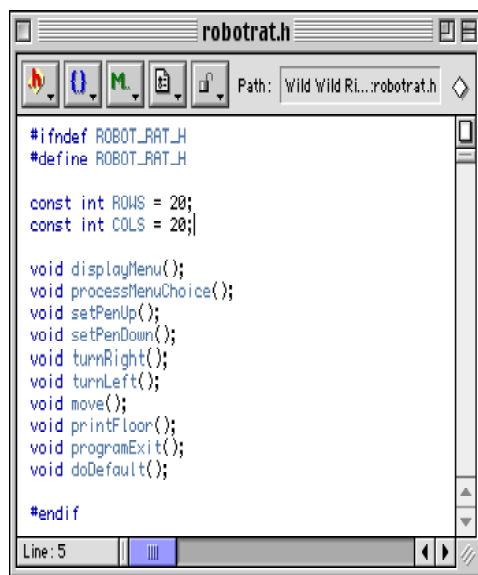


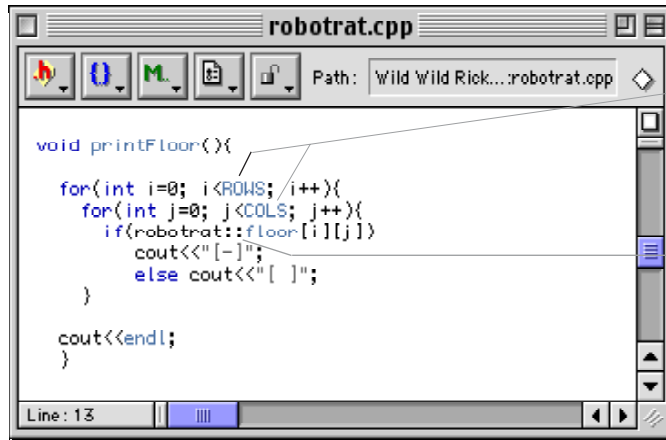
Figure 3-16: robotrat.h with ROWS & COLS Constants Declared

Figure 3-15 shows how C-style comments can be used to provide section headers in source code. The floor array is declared inside of the robotrat namespace. Two constants are used in the array declaration: ROWS and COLS. The names ROWS and COLS are good choices for these two constants since they lend another degree of abstraction to the robotrat solution. Both these constants will be used again in the printFloor(), and move() functions. Figure 3-16 shows the ROWS and COLS constants being declared in the robotrat.h file.

With the floor array work completed you can turn your attention to the business of printing the floor array to the screen. The printFloor() function is currently a stubbed function, which is a good thing since all you need do is add the code that will give printFloor() its intended functionality.

There are two things to consider when implementing printFloor(). First, how to access the floor array when its declaration appears in the robotrat namespace, and second, how to represent a marked or unmarked floor square when the floor array is printed to the screen.

The first consideration is resolved with the use of the scope resolution operator. The second consideration serves as an example of how an earlier design decision can affect later design decisions. Figure 3-17 shows the source code for the printFloor() function.



Constants *ROWS* and *COLS* used in the *for* statements

Scope resolution operator used to access floor inside the *robotrat* namespace.

Figure 3-17: The printFloor() Function

The printFloor() function is implemented using two for loops. The outer loop processes the *ROWS* of the floor array and the inner loop processes the *COLS* of the floor array. The meat of the function is the if statement that tests each element of the array. If the test is true, that is, if the boolean value located in that particular array element has been set to true by the robot rat, then it will be rendered on the screen as marked. In this case, a marked element is rendered as the string of characters "*[-]*". If an array element is not marked it will evaluate to false. An unmarked element is rendered on the screen as the string of characters "*[*".

With the work completed on the printFloor() function it is time to move to the testing phase.

TESTING (THIRD ITERATION)

Figure 3-18 shows the results of the robot rat program being run and the Print Floor menu choice being selected.

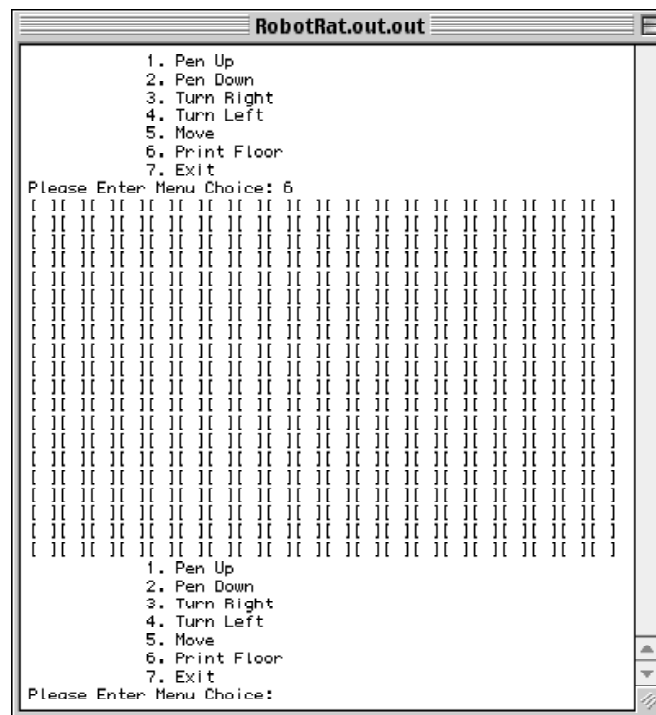


Figure 3-18: Robot Rat printFloor() Function Test

It appears everything works fine, at least when the array elements are false. It would be a good idea to write some code that sets a few of the array elements to true just to make sure everything is working properly. Figure 3-19 shows a temporary function called `setTestPattern()` being declared and defined within `robotrat.cpp`.

```

robotrat.cpp
Path: Wild\Wild Ri...botrat.cpp

/*****
Temporary Test Functions
*****/

void setTestPattern(); ← setTestPattern() declaration

void setTestPattern(){
  robotrat::floor[0][0] = true; ← Various floor array elements set to
  robotrat::floor[0][1] = true; ← true in the setTestPattern() function
  robotrat::floor[0][2] = true;
  robotrat::floor[0][3] = true;
  robotrat::floor[1][3] = true;
  robotrat::floor[2][3] = true;
  robotrat::floor[3][3] = true;
  robotrat::floor[4][3] = true;
  robotrat::floor[5][3] = true;
}

/*****/
Line: 81

```

Figure 3-19: `setTestPattern()` Function

The `setTestPattern()` function can then be used in the `printFloor()` function to set the test pattern before printing. Figure 3-20 shows the `setTestPattern()` function being called by `printFloor()`.

```

robotrat.cpp
Path: Wild\Wil...rat.cpp

void printFloor(){
  setTestPattern(); ← setTestPattern() called before printing the
  for(int i=0; i<ROWS; i++){
    for(int j=0; j<COLS; j++){
      if(robotrat::floor[i][j])
        cout<<"-";
      else cout<<" ";
    }
  }
  cout<<endl;
}
Line: 92

```

Figure 3-20: `setTestPattern()` Function Being Used for Testing in the `printFloor()` Function.

Figure 3-21 shows the results of the next program test. The pattern prints as expected. With testing complete the temporary code can be completely removed from the robot rat project or commented out. If you are certain you will not be needing the test code in the future, removal is best as it leaves your source code less cluttered.

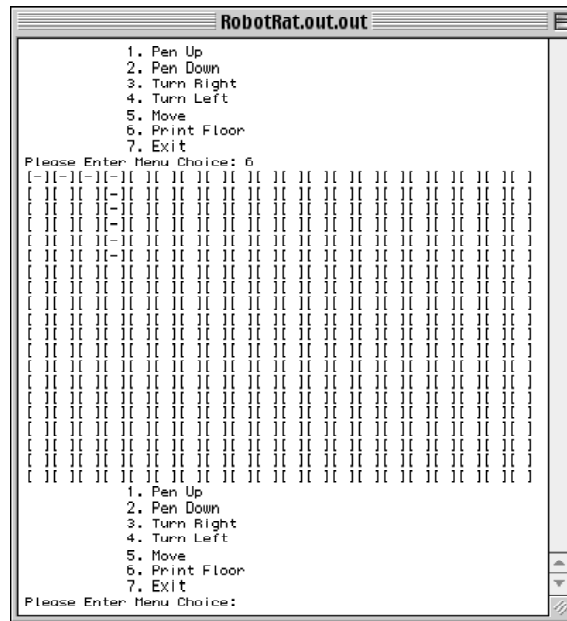


Figure 3-21: Robot Rat printFloor() Test with Test Pattern

INTEGRATION (THIRD ITERATION)

Integration, once again, took place in concert with implementation, the result being nothing to explicitly integrate into the robot rat program. This marks the completion of the third iteration of the development cycle.

DESIGN (FOURTH ITERATION)

How next to grow the design? You have a floor, and you can print the floor and any patterns it may contain. It now appears the next step is to design the move function. But, before you can move the robot rat you must know what direction it is facing. Also, before the floor can be marked the robot rat's pen position must be determined. Setting the robot rat's direction and pen position are two features that must be implemented before the move function can be implemented. Table 3-9 lists the design considerations and design decisions for the fourth iteration.

Design Considerations	Design Decisions
Setting and keeping track of the robot rat's direction	Use an enumerated type called Direction with four possible values, NORTH, SOUTH, EAST, and WEST. Declare a variable of type Direction called rats_direction to store the robot rat's current direction. The rats_direction variable will be set to a new value using either the turnRight() or turnLeft() functions. Its new value will depend on its current value. rats_direction will have an initial value of EAST.

Table 3-9: Fourth Iteration Design Consideration and Design Decisions

Design Considerations	Design Decisions
Setting and keeping track of the robot rat's pen position.	Use an enumerated type called PenPosition with two possible values, UP, and DOWN. Declare a variable of type PenPosition called pen_position to store the robot rat's current pen position. The pen_position variable will be set to a new value using the setPenUp() or setPenDown() functions. Its new value will be set regardless of its current value. pen_position will have an initial value of UP.

Table 3-9: Fourth Iteration Design Consideration and Design Decisions

Each of these design considerations deal with issues relating to two important robot rat attributes, namely, direction and pen position. As described in table 3-9, the variable `rats_direction` will only be allowed to have four possible values, NORTH, SOUTH, EAST, or WEST, and will be initialized to EAST. Said another way, the `rats_direction` variable can have four possible states and its initial state will be EAST. A state transition diagram can be used to visualize each state and show how the `rats_direction` variable will transition from state to state. Figure 3-22 shows the state transition diagram for `rats_direction`.

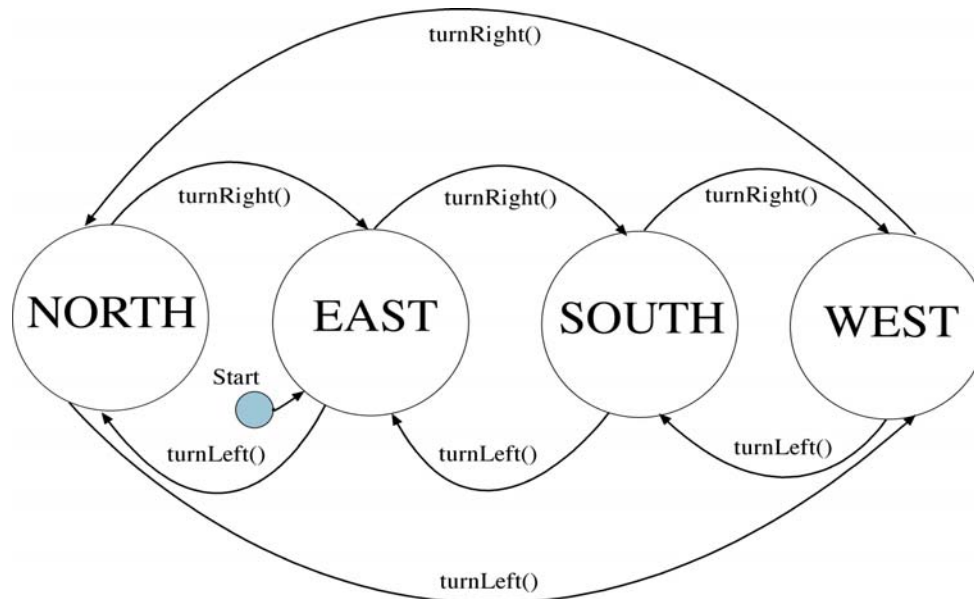


Figure 3-22: State Transition Diagram for `rats_direction` Variable.

When the robot rat program starts, `rats_direction` will be initialized to EAST. Each of four possible states are indicated by the large circles. To change `rats_position` state something must happen. Either the `turnRight()` or `turnLeft()` function must be called. To change the `rats_direction` to SOUTH, from the EAST state, the `turnRight()` function is called. Note the direction of the arrows pointing from one state to the next. To go back to the EAST state from the SOUTH state the `turnLeft()` function must be called.

The state transition diagram for `pen_position` is shown in figure 3-23. It is similar to `rats_direction` state transition diagram with the exception being a transition can occur that results in no change of state. When the robot rat program starts the `pen_position` variable is initialized to the UP state. It can be changed to the DOWN state by a call to the `setPenDown()` function. If, however, it is in the UP state and the `setPenUp()` function is called, its value is reset to UP, in which case no change of state occurs.

This is enough designing for now. It is time to implement these two state transition diagrams.

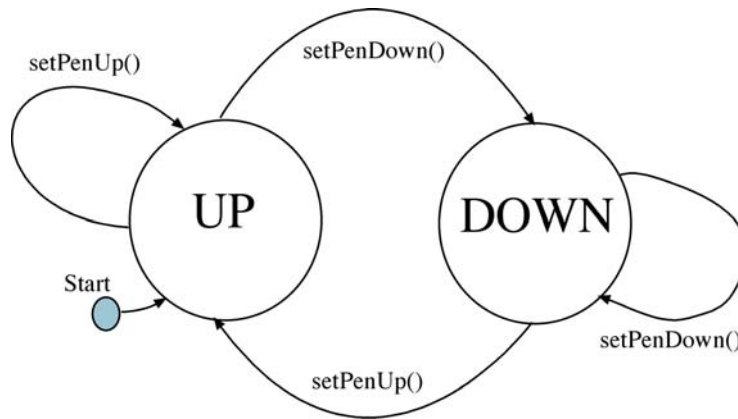


Figure 3-23: State Transition Diagram for pen_position

IMPLEMENTATION (FOURTH ITERATION)

Begin by declaring the enumerated types `Direction` and `PenPosition` in the `robotrat.h` file. Figure 3-24 shows the `robotrat.h` file after the addition.

Next, edit the `robotrat.cpp` file and declare and initialize the variables `pen_position` and `rats_position` in the `robotrat` namespace. Figure 3-25 shows the source code for `robotrat.cpp` after the modification.

```

robotrat.cpp
Path: Wild Wild Rick:De...Rat:robotrat.cpp

#include "robotrat.h"
#include <iostream.h>
#include <stdlib.h>

/*****
File Scope Variable Declarations
*****/
namespace robotrat{
static bool floor[ROWS][COLS] = {};
static PenPosition pen_position = UP;
static Direction rats_direction = EAST;
}
Line: 6
  
```

Figure 3-25: Declaration of pen_position & rats_position

```

robotrat.h
Path: Wild Wild Ri...robotrat.h

#ifndef ROBOT_RAT_H
#define ROBOT_RAT_H

const int ROWS = 20;
const int COLS = 20;

enum Direction {NORTH, SOUTH, EAST, WEST};
enum PenPosition {UP, DOWN};

void displayMenu();
void processMenuChoice();
void setPenUp();
void setPenDown();
void turnRight();
void turnLeft();
void move();
void printFloor();
void programExit();
void doDefault();

#endif
Line: 8
  
```

Figure 3-24: Direction and PenPosition Enum Types Added to robotrat.h

Once the variables are declared and initialized the functions `setPenUp()`, `setPenDown()`, `turnRight()`, and `turnLeft()` can be edited to implement their intended functionality. The first two functions, `setPenUp()` and `setPenDown()`, are the easiest. Simply replace the stub message statement with an assignment. Figure 3-26 shows both of these functions after modification.

Each of the functions `turnRight()` and `turnLeft()` can be implemented with a switch statement. Test the value of `pen_position` and compare it to the valid states as defined in the enumerated type `Direction` and set the new value according to the `rats_direction` state transition diagram. Figure 3-27 shows the `turnRight()` function and figure 3-28 shows the `turnLeft()` function.

Once all function modifications are complete you can move to the testing phase.

```

void setPenUp(){
    robotrat::pen_position = UP;
}

void setPenDown(){
    robotrat::pen_position = DOWN;
}

```

Figure 3-26: setPenUp() & setPenDown() Functions

```

void turnRight(){
    switch(robotrat::rats_direction){
        case NORTH: robotrat::rats_direction = EAST;
                    break;
        case EAST:  robotrat::rats_direction = SOUTH;
                    break;
        case SOUTH: robotrat::rats_direction = WEST;
                    break;
        case WEST:  robotrat::rats_direction = NORTH;
                    break;
        default:    robotrat::rats_direction = EAST;
    }
}

```

Figure 3-27: turnRight() Function

```

void turnLeft(){
    switch(robotrat::rats_direction){
        case NORTH: robotrat::rats_direction = WEST;
                    break;
        case EAST:  robotrat::rats_direction = NORTH;
                    break;
        case SOUTH: robotrat::rats_direction = EAST;
                    break;
        case WEST:  robotrat::rats_direction = SOUTH;
                    break;
        default:    robotrat::rats_direction = EAST;
    }
}

```

Figure 3-28: turnLeft() Function

TESTING (FOURTH ITERATION)

Well...you could test the changes you just made but unless you add a few lines of code for testing purposes you will not see any results of changing the robot rat's pen position or its direction. Using the `turnLeft()` function as an example, you can add statements to each case to print a message when robot rat's direction has changed. Figure 3-29 shows the additions to the `turnLeft()` function.

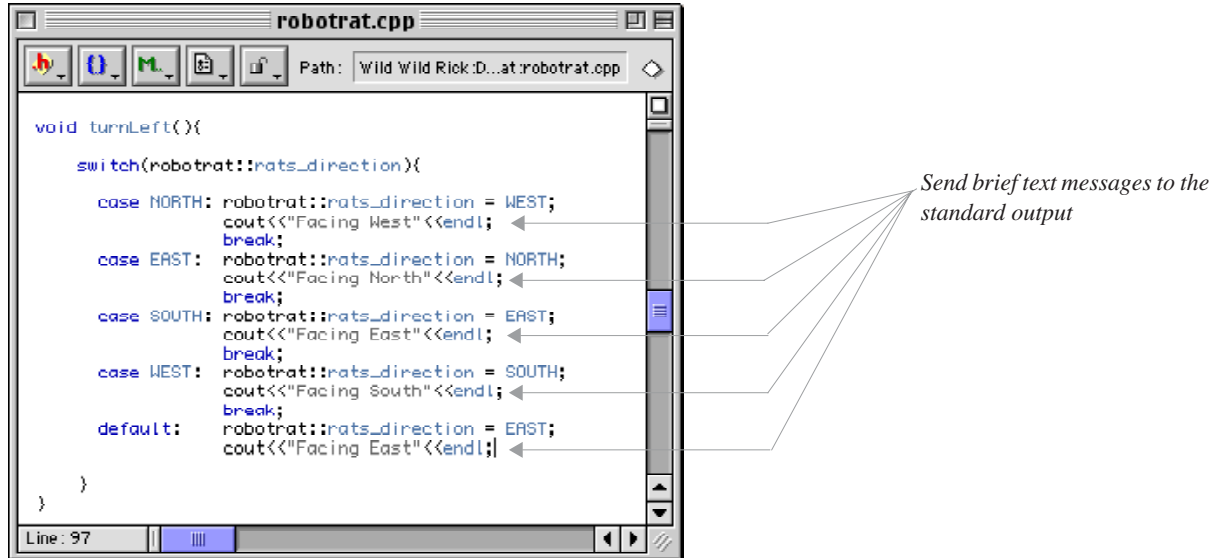


Figure 3-29: `turnLeft()` Function with `cout` Statements

Figure 3-30 shows the results of testing the `turnLeft()` function. You may want to add similar test statements to `turnRight()`, `setPenUp()`, and `setPenDown()` and test everything for proper operation. Again, when you have completed all testing for this iteration you can remove the test statements from your source code.

INTEGRATION (FOURTH ITERATION)

No integration is necessary for this iteration. Time to move on to the fifth iteration of the development cycle.

DESIGN (FIFTH ITERATION)

The robot rat project is nearly complete save for the `move()` function. Looking back at the initial analysis of robot rat attributes you will discover two that have yet to be implemented. They are current row and current column. These should complete the attribute set required to define the state of the robot rat at any time during the execution of the program. Using all attributes together you can determine the robot rat's position by row and column, what direction it is facing, and its pen position. Yet there's still some work to do to determine how to implement the `move()` function.

How should a move be executed? How should the robot rat respond when instructed to move past the boundaries of the floor? These are great questions that deal with the robot rat's behavior. The `move()` function is where robot rat's behavior will be defined.

Table 3-10 list the design considerations and decisions for this iteration of the development cycle.

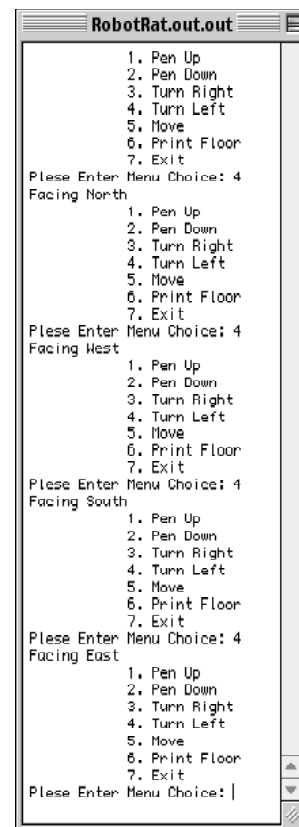


Figure 3-30: `turnLeft()` Test

Design Considerations	Design Decisions
How should the move() function be structured to determine the robot rat's direction and pen position.	Use nested switch statements to determine the state of the pen_position and rats_direction.
How will a command to move past floor boundaries be handled?	Move up to the floor boundary and then stop to wait for another move command.
How will floor array cells be marked during a move?	If the pen is down, set the floor array element at the indicated position to true. If the pen is up don't worry about marking the floor.
What does it mean to move north, south, east, or west in terms of rows and columns?	North: row position decreases, col stays the same. (row--, col) South: row position increases, col stays the same. (row++, col) East: row position stays the same, col increases. (row, col++) West: row position stays the same, col decreases. (row, col--)

Table 3-10: Design Considerations and Decisions: Fifth Iteration

The move function is fairly complex. In it you must check the state of the robot rat to determine what direction it is facing and its pen position. You must determine how many spaces the user wants to move and ensure the move doesn't go outside the array boundaries. A good idea at this point would be to develop a pseudocode listing of the move function. Example 3.4 provides the pseudocode for the framework of the move() function.

```

Get spaces to move from user
determine position of pen
if pen_position is up
determine direction robot rat is facing
if rats_direction is NORTH
    execute movement north (adjust current_row, no mark)
else if rats_direction is SOUTH
    execute movement south (adjust current_row, no mark)
    else if rats_direction is EAST
        execute movement east (adjust current_col, no mark)
    else if rats_direction is WEST
        execute movement west (adjust current_col, no mark)
if pen_position is down
determine direction robot rat is facing
if rats_direction is NORTH
execute movement north (adjust current_row, mark cells)
else if rats_direction is SOUTH
    execute movement south (adjust current_row, mark cells)
    else if rats_direction is EAST
        execute movement east (adjust current_col, mark cells)
    else if rats_direction is WEST
        execute movement west (adjust current_col, mark cells)

```

3.4 move() function pseudocode

According to the pseudocode, the move() function will first get the number of spaces to move from the user. It will then perform a move according to the position of the robot rat's pen. If the pen is up marking the floor is not required. The move then becomes a matter of setting the value of current_row or current_col to the new position. Error checking must be employed to make sure the move stays within the floor array boundaries.

If the robot rat's pen position is down the floor must be marked, meaning each floor array cell affected by the move must be set to true.

The most complex part of the `move()` function will no doubt be the error checking code required to make sure the moves stay within the floor boundaries. Again, pseudocode will help you in your design. Example 3.5 gives the pseudocode for the movement in the NORTH direction with the pen in the UP position.

*3.5 NORTH move pseudocode
pen in the UP position*

```
if current_row minus spaces to move is greater than zero
    set current_row to current_row minus spaces
else
    set current_row to zero
```

Movement in the NORTH direction is with the pen DOWN will be more involved because each floor array element along the path of movement must be set to true. Example 3-6 gives the pseudocode for movement in the NORTH direction with the pen in the DOWN position.

*3.6 NORTH move pseudocode
pen in DOWN position*

```
calculate number of spaces left to move north from current row
if spaces left to move is less than or equal to zero
    set spaces to current_row
while there are spaces left to move
    set floor[current_row][current_col] to true
    decrement current_row by one
    decrement spaces by one
```

After completing your analysis of the `move()` function you are ready to move on to the implementation phase.

IMPLEMENTATION (FIFTH ITERATION)

Since the `move()` function is already stubbed all you need do is remove the stubbing message and replace it with the source code that will give the `move()` function its required functionality.

First order of business is to get the number of spaces from the user. When the user selects menu item 5, the `move()` function will be called. That would be a good place to ask for the number of spaces the user wants the robot rat to move. The user's entry will need to be stored for further processing but will not be needed outside of the `move()` function. A local variable named `spaces` will do the trick. Once the user enters the spaces the `move()` function can do its job. Figure 3-31 shows the code for the top half of the `move()` function. This part of the source code includes the declaration of the `spaces` variable, the request for the user to enter the number of spaces to move and the assignment of that value to the `spaces` variable using the `cin` object, and the switch statements that determine the position of the pen and the robot rat's direction.

The complete source code for the rest of the `move()` function is listed at the end of the chapter.

TESTING (FIFTH ITERATION)

When you have completed implementing the `move()` function you need to test it thoroughly. Move with the pen up and down in all directions. You must be absolutely sure that movement in any direction stays within the floor array boundaries. Figure 3-32 shows the robot rat program after a few movements have been executed.

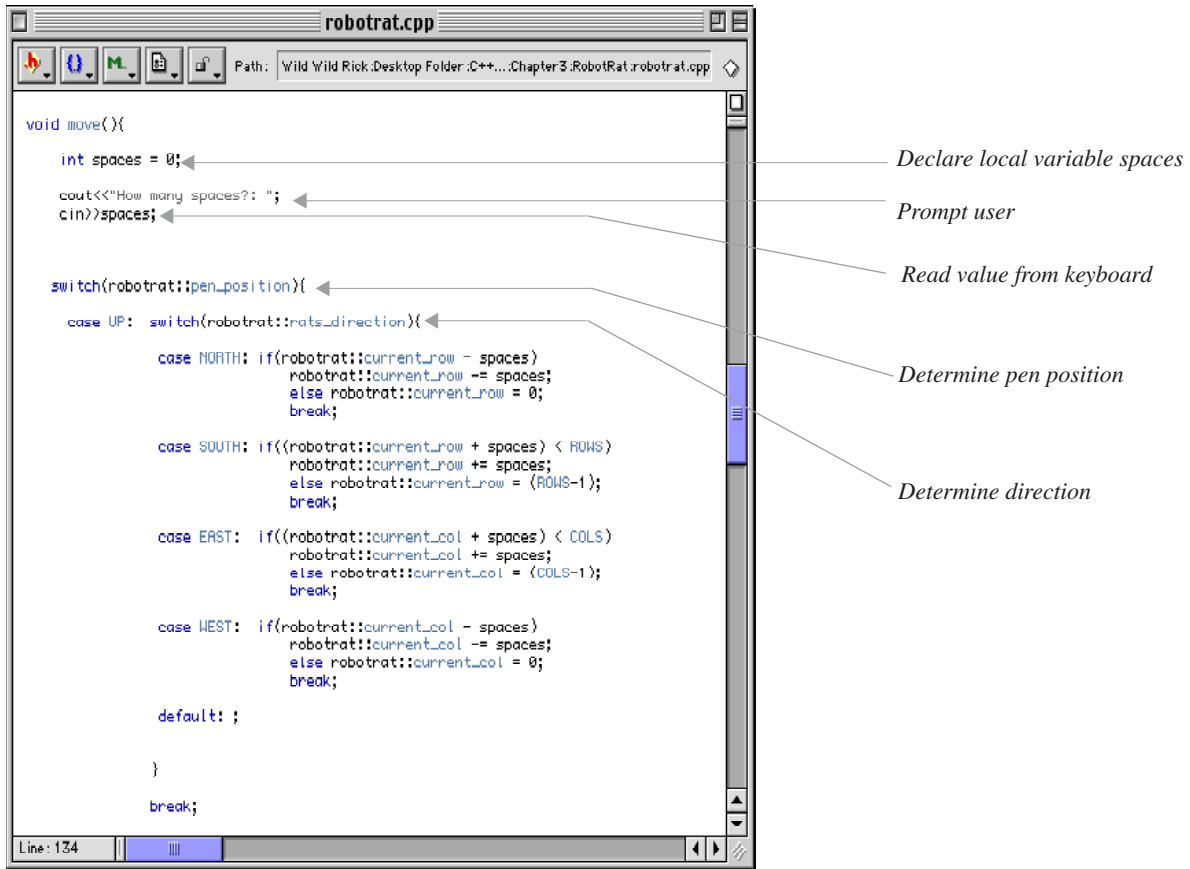


Figure 3-31: move() Function, Top Half

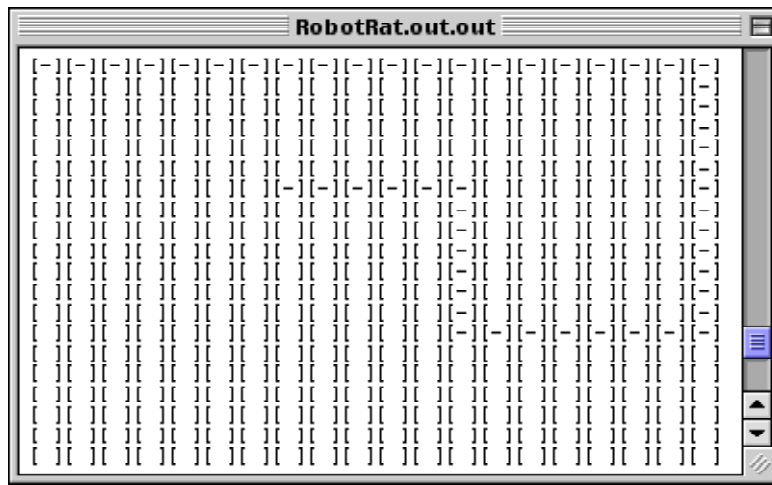


Figure 3-32: move() Function Test

INTEGRATION (Fifth Iteration)

Integration has again taken place along with implementation.

WRAPPING UP THE PROJECT

The implementation and testing of the `move()` function marks the beginning of the end of the robot rat project. You must now give the complete program a thorough test of all functionality. Test until you are absolutely sure everything runs according to specification and that it offers no rude surprises to a user. Table 3-11 lists a few things you will want to double-check before handing in your project.

Double-Check...	To ensure...
Source code formatting	...it is neat, logically aligned, and indented.
Comments	...they are not overdone. Remember, if you used good names for functions, variables, and constants, your code will be largely self-commenting.
File Comment Header	...it is at the top of every source file and lists your name and the name of the project. Check with your instructor for additional information required to be placed in the file comment header.
When printing source code on paper	...that it fits on the page. If long lines wrap to the next line adjust the font, print in the landscape mode, or split the line into smaller pieces in the source file.

Table 3-11: Things To Double-Check Before Handing In Project

COMPLETE ROBOT RAT SOURCE CODE LISTING

```

/*****
File:      robotrat.h
Student Name:
Project:
Class:

...and any additional header info

*****/
#ifndef ROBOT_RAT_H
#define ROBOT_RAT_H

const int ROWS = 20;
const int COLS = 20;

enum Direction {NORTH, SOUTH, EAST, WEST};
enum PenPosition {UP, DOWN};

void displayMenu();
void processMenuChoice();
void setPenUp();
void setPenDown();
void turnRight();
void turnLeft();
void move();
void printFloor();
void programExit();
void doDefault();

#endif

```

*3.7 Complete Robot Rat
Source Code Listing*

```

/*****
File:      robotrat.cpp
Student Name:
Project:
Class:

...and any additional header info
*****/

#include "robotrat.h"
#include <iostream.h>
#include <stdlib.h>

/*****
File Scope Variable Declarations
*****/
namespace robotrat{
static bool floor[ROWS][COLS] = {};
static PenPosition pen_position = UP;
static Direction rats_direction = EAST;
static int current_row = 0;
static int current_col = 0;
}

/*****
Function Definitions
*****/

void displayMenu(){
cout<<"1. Pen Up"<<endl;
cout<<"2. Pen Down"<<endl;
cout<<"3. Turn Right"<<endl;
cout<<"4. Turn Left"<<endl;
cout<<"5. Move"<<endl;
cout<<"6. Print Floor"<<endl;
cout<<"7. Exit"<<endl;
}

void setPenUp(){

    robotrat::pen_position = UP;
}

void setPenDown(){
    robotrat::pen_position = DOWN;
}

void turnRight(){
switch(robotrat::rats_direction){
    case NORTH: robotrat::rats_direction = EAST;
                break;
    case EAST:  robotrat::rats_direction = SOUTH;
                break;
    case SOUTH: robotrat::rats_direction = WEST;
                break;
    case WEST:  robotrat::rats_direction = NORTH;
                break;
    default:    robotrat::rats_direction = EAST;
}
}

void turnLeft(){
switch(robotrat::rats_direction){
    case NORTH: robotrat::rats_direction = WEST;
                break;
    case EAST:  robotrat::rats_direction = NORTH;
                break;
    case SOUTH: robotrat::rats_direction = EAST;
                break;
    case WEST:  robotrat::rats_direction = SOUTH;
                break;
    default:    robotrat::rats_direction = EAST;
}
}
}

```

```

void move() {
    int spaces = 0;
    cout<<"How many spaces?: ";
    cin>>spaces;

    switch(robotrat::pen_position){
    case UP: switch(robotrat::rats_direction){
        case NORTH: if(robotrat::current_row - spaces)
            robotrat::current_row -= spaces;
            else robotrat::current_row = 0;
            break;

        case SOUTH: if((robotrat::current_row + spaces) < ROWS)
            robotrat::current_row += spaces;
            else robotrat::current_row = (ROWS-1);
            break;

        case EAST: if((robotrat::current_col + spaces) < COLS)
            robotrat::current_col += spaces;
            else robotrat::current_col = (COLS-1);
            break;

        case WEST: if(robotrat::current_col - spaces)
            robotrat::current_col -= spaces;
            else robotrat::current_col = 0;
            break;

        default: ;
    }
    break;

    case DOWN: switch(robotrat::rats_direction){

        case NORTH: if((robotrat::current_row - spaces)<=0)
            spaces = robotrat::current_row;

            while(spaces){
                robotrat::floor[robotrat::current_row--][robotrat::current_col] = true;
                --spaces;
            }

            break;

        case SOUTH: if( (robotrat::current_row + spaces) > ROWS)
            spaces = ((ROWS-1) - robotrat::current_row);

            while(spaces){
                robotrat::floor[robotrat::current_row++][robotrat::current_col] = true;
                --spaces;
            }

            break;

        case EAST: if((robotrat::current_col + spaces) >= COLS)
            spaces = ((COLS-1) - robotrat::current_col);

            while(spaces){
                robotrat::floor[robotrat::current_row][robotrat::current_col++] = true;
                --spaces;
            }

            break;

        case WEST: if(robotrat::current_col - spaces<=0)
            spaces = robotrat::current_col;

            while(spaces){
                robotrat::floor[robotrat::current_row][robotrat::current_col--] = true;
                --spaces;
            }

            break;

        default: ;
    }
    break;
    default: ;
}
}

```

```

void printFloor() {
    for(int i=0; i<ROWS; i++){
        for(int j=0; j<COLS; j++){
            if(robotrat::floor[i][j])
                cout<<"-]";
            else cout<<"[ ]";
        }
        cout<<endl;
    }
}

void programExit(){
    exit(0);
}

void doDefault(){
    cout<<"Please Enter A Valid Menu Choice: "<<endl;
}

void processMenuChoice() {
    char input = '0';

    cout<<"Please Enter Menu Choice: ";

    cin>>input;

    switch(input){
        case '1': setPenUp();
                 break;

        case '2': setPenDown();
                 break;

        case '3': turnRight();
                 break;

        case '4': turnLeft();
                 break;

        case '5': move();
                 break;

        case '6': printFloor();
                 break;

        case '7': programExit();

        default : doDefault();
    }
}

/*****
File:      main.cpp
Student Name:
Project:
Class:

...and any additional header info
*****/

#include <iostream>
#include "robotrat.h"

using namespace std;

int main()
{
    for(;;){
        displayMenu();
        processMenuChoice();
    }
    return 0;
}

```

SUMMARY

Use the project approach strategy to help you sustain development momentum. Apply the development cycle iteratively. Don't try to program everything at once. Break the problem into small pieces, solve the individual pieces, and combine them into the total solution. Test, test, test!

Skill Building Exercises

1. **Create Robot Rat Project:** Using the source code from the robot rat project in this chapter, create a robot rat project in your IDE, enter the source code, then compile and run the project.
2. **Obtain Project Specifications:** Obtain the project specifications or handouts for all the projects required for this class. Apply the first phase of the project approach strategy to each one to ensure you understand the project requirements.

SUGGESTED PROJECTS

1. **Research:** Research other software development methodologies. Compare them with the approach suggested in this chapter. What are their similarities? What are their major differences?

SELF TEST QUESTIONS

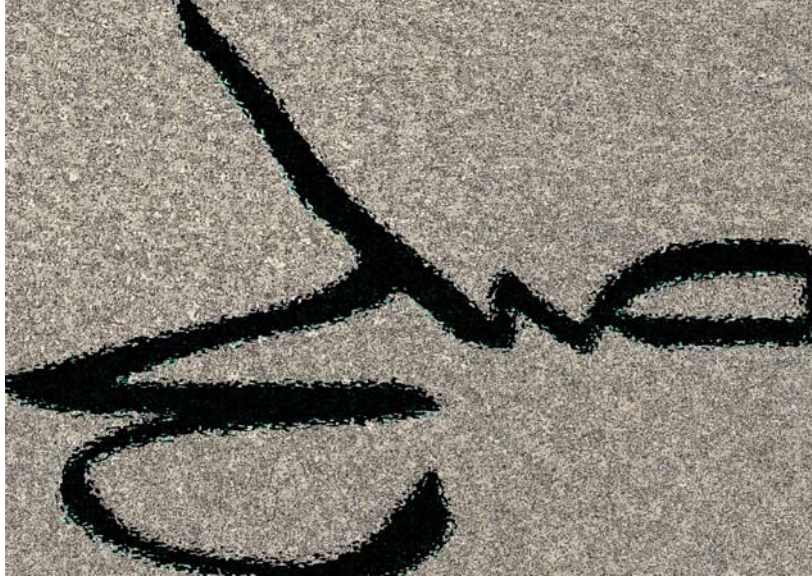
1. What is the purpose of the project approach strategy?
2. What is the purpose of the development cycle?
3. Describe how to apply the project approach strategy and development cycle in an iterative fashion.
4. Why is it a good idea to do just enough design to get started coding? Does this approach have practical application in the real programming world? What future problems regarding application design does using this approach help to avoid?
5. How is function stubbing used in the robot rat project?
6. Why is component testing important?
7. Why is frequent component integration important?
8. What is the purpose of pseudocode?
9. What is the purpose of a state transition diagram?
10. What C++ flow control structure can be used to implement the functionality described by a state transition diagram?

REFERENCES

Metrowerks CodeWarrior Reference Documentation for Microsoft Windows 95/98/NT and Apple Macintosh.

NOTES

CHAPTER 4



Lines In The Sand

COMPUTERS, PROGRAMS, & ALGORITHMS

LEARNING OBJECTIVES

- *Define the concept of a computer*
- *Explain why the computer is a remarkable device*
- *Explain how a computer differs from other machines*
- *Explain how a computer stores and retrieves programs for execution*
- *State the difference between a computer and a computer system*
- *State the purpose of a microprocessor and the role it plays in a computer system*
- *Define the concept of a program from the human perspective and the computer perspective*
- *Describe how programs are represented in a computer's memory*
- *List and describe the nine stages of the C++ program transformation process*
- *List and describe the four steps of the processing cycle*
- *State the purpose and objective of a computer's memory system*
- *Define the concept of an algorithm*
- *List the characteristics of a good algorithm*

INTRODUCTION

Computers, programs, and algorithms are three closely related topics that deserve your attention before you start learning about C++ proper. Why? Simply put, computers execute programs, and programs implement algorithms. As a programmer, you will live your life in the world of computers, programs, and algorithms.

As you progress through your studies you will find it extremely helpful to understand what makes a computer a computer, what particular feature makes a computer a truly remarkable device, and how one functions from a programmer's point of view. You will also find it helpful to know how humans view programs and how a human readable program is translated into a computer executable form. Lastly, it will be imperative for you to thoroughly understand the concept of an algorithm and how good and bad algorithms ultimately affect program performance.

WHAT IS A COMPUTER?

A computer is a device whose function, purpose, and behavior is prescribed, controlled, or changed via a set of stored instructions. A computer can also be described as a general purpose machine. One minute a computer may execute instructions making it function as a word processor or page layout machine. The next minute it might be functioning as a digital canvas for an artist. Again, this functionality is implemented as a series of instructions. Indeed, the only difference between the computer functioning as a word processor and the same computer functioning as a digital canvas is in the set of instructions the computer is executing in each case.

COMPUTER VS. COMPUTER SYSTEM

Due to the ever shrinking size of the modern computer it is often difficult for students to separate the concept of the computer from the computer system in which it resides. As a programmer, you will be concerned with both. By that I mean you will need to understand issues that have a direct bearing on the particular processor that powers a computer system in addition to issues related to the computer system as a whole. Luckily though, as a C++ programmer, you can be extremely productive armed with only a high-level understanding of each. Ultimately, I do recommend spending the time required to get intimately familiar with your programming platform choice. For this chapter I will use the Apple Power Mac G4 as an example, but the concepts are the same for any computer or computer system.

COMPUTER SYSTEM

A typical Power Mac G4 computer system is pictured in figure 4-1. The computer system comprises the system



Figure 4-1: Typical Power Mac G4 System

unit, monitor, speakers, keyboard, and mouse. The computer system also includes any operating system or utility software required to make all the components work together.

The system unit houses the processor, power supply, internal hard disk drives, memory, and other system components required to interface the computer to the outside world. These interface components consume the majority of available space within the system unit as shown in figure 4-2.



Figure 4-2: System Unit

The processor is connected to the system unit main logic board. Electronic pathways called buses connect the processor to the various interface components. Other miscellaneous electronic components are located on the main logic board to control the flow of communication between the processor and the outside world. Figure 4-3 is a block diagram of a Power Mac G4 main logic board.

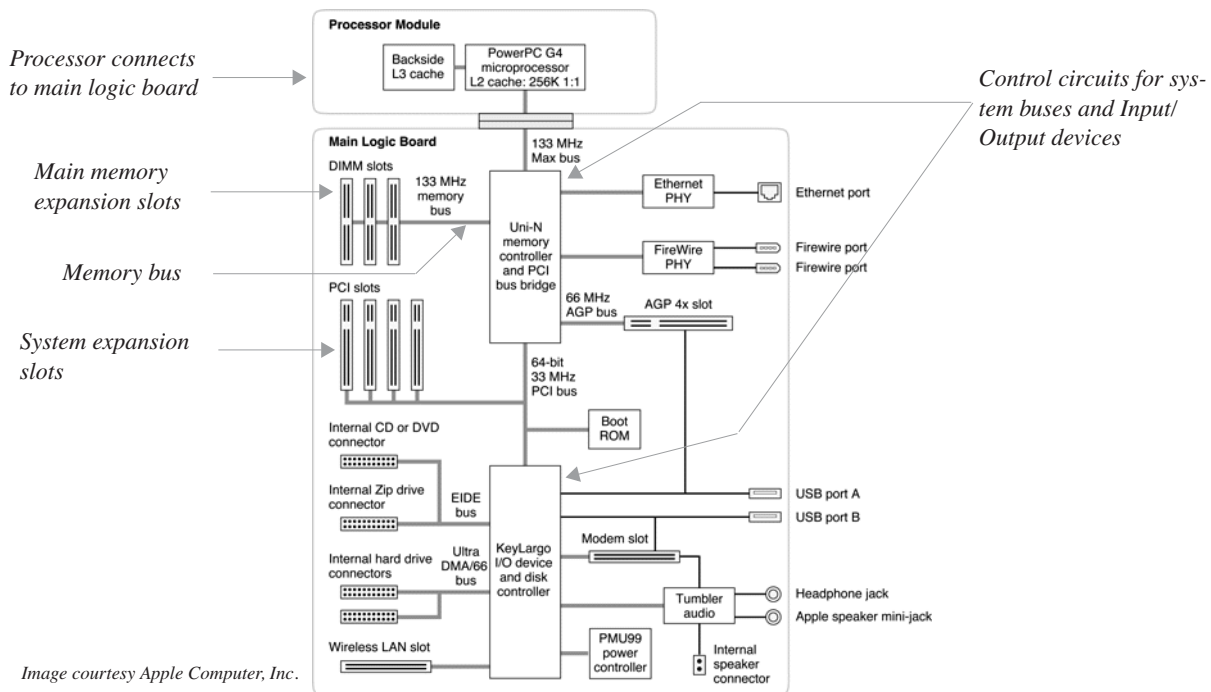


Figure 4-3: Main Logic Board Block Diagram

Figure 4-3 does a good job of highlighting the number of computer system support components required to help the processor do its job. The main logic board supports the addition of main memory, auxiliary storage devices, communication devices such as a modem, a wireless local area network card as well as a standard Ethernet port, keyboard, mouse, speakers, microphones, Firewire devices, and, with the insertion of the appropriate third party system expansion card, just about any other functionality you can imagine. All this system functionality is supported by the main logic board, but the heart of the system is the PowerPC G4 processor. Let us take a closer look.

PROCESSOR

If you lift up the heat sink pictured in figure 4-2 you'd see a PowerPC G4 processor like the one shown in figure 4-4.

The PowerPC G4 7400 microprocessor pictured here runs at speeds between 350 and 500 megahertz with a Millions of Instructions per Second (MIPS) rating of 917 MIPS at 500 megahertz. All this means the G4 is a powerful processor, yet at the time of this writing there are more powerful processors on the market, namely the G5!

The 7400 processor is a Reduced Instruction Set Computer (RISC) meaning its architecture is optimized to execute instructions in as few clock cycles as possible. The block diagram for the 7400 is shown in figure 4-5 and is even more impressive than that of the main logic board. The G4 is a superscalar, pipelined processor. It is superscalar in that it can pull two instructions from the incoming instruction stream and execute them simultaneously. It is pipelined in that instruction execution is broken down into discrete steps meaning several instructions can be in the pipeline at any given moment at different stages of execution.



Figure 4-4: PowerPC G4 Processor

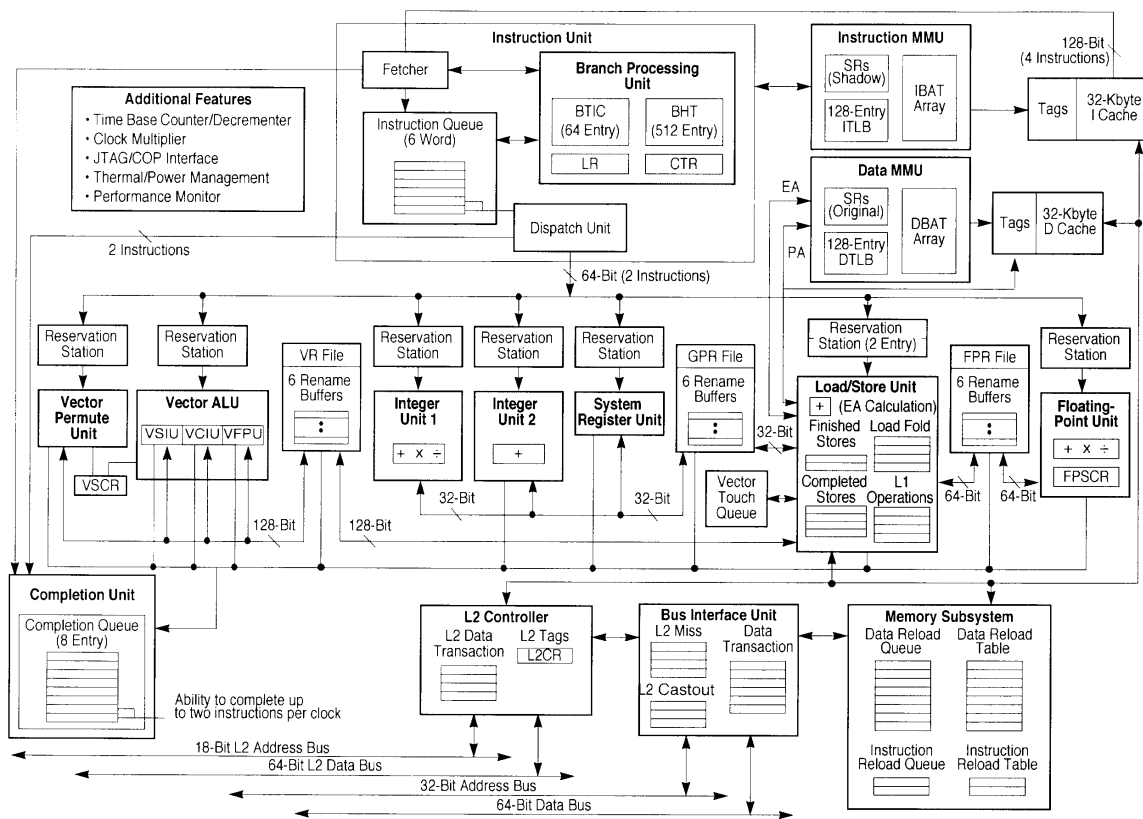


Figure 4-5: Motorola PowerPC 7400 Block Diagram

THREE ASPECTS OF COMPUTER ARCHITECTURE

There are generally three aspects of processor architecture programmers should be aware of: feature set, feature set implementation, and feature set accessibility.

FEATURE SET

A processor's feature set is derived from its design. Can floating point arithmetic be executed in hardware or must it be emulated in software? Must all data pass through the processor or can input/output be handled off chip while the processor goes about its business? How much memory can the processor access? How fast can it run? How much data can it process per unit time? A processor's design determines the answers to these and other feature set issues.

FEATURE SET IMPLEMENTATION

This aspect of computer architecture is concerned primarily with how processor functionality is arranged and executed in hardware. How does the processor implement the feature set? Is it a Reduced Instruction Set Computer (RISC), or Complex Instruction Set Computer (CISC)? Is it superscalar and pipelined? Does it have a vector execution unit? Is the floating-point unit on the chip with the processor or does it sit off to the side? Is the super fast cache memory part of the processor or is it located on another chip? These questions all deal with how processor functionality is achieved or how its design is executed.

FEATURE SET ACCESSIBILITY

Feature set accessibility is the aspect of a processor's architecture you are most concerned with as a programmer. Processor designers make a processor's feature set available to programmers via the processor's instruction set. A valid instruction in a processor's raw instruction set is a set of voltage levels that, when decoded by the processor, have special meaning. A high voltage is usually translated as "on" or "1" and a low voltage is usually translated as "off" or "0". A set of on and off voltages is conveniently represented to humans as a string of 1's and 0's. Instructions in this format are generally referred to as machine instructions or machine code. However, as processor power increases, the size of machine instructions grow as well, making it extremely difficult for programmers to deal directly with machine code.

FROM MACHINE CODE TO ASSEMBLY

To make a processor's instruction set easier for humans to understand and work with each machine instruction is represented symbolically in a set of instructions referred to as an assembly language. To the programmer, assembly language represents an abstraction or a layer between programmer and machine intended to make the act of programming more efficient. Programs written in assembly language must be translated into machine instructions before being executed by the processor. A program called an assembler translates assembly language to machine code.

Although assembly language is easier to work with than machine code it requires a lot of effort to crank out a program in assembly code. Assembly language programmers must busy themselves with issues like register usage and stack conventions. High-level languages like C++ add yet another layer of abstraction. C++, with its object-oriented language features, let programmers think in terms of solving the problem at hand, not in terms of the processor.

When you program in C++ you are targeting an abstract machine as defined by the ANSI C++ standard and implemented in a particular compiler and supporting standard libraries.

WHAT IS A PROGRAM?

Intuitively you already know the answer to this question. A program is something that runs on a computer. This simple definition works well enough but as a programmer you will need to arm yourself with a better understanding of exactly what makes a program a program. In this section I will discuss programs from two aspects: the computer and the human. You will find this information extremely helpful and it will tide you over until you take a formal course on computer architecture.

TWO VIEWS OF A PROGRAM

A program is a set of programming language instructions and any data the instructions act upon or manipulate. This is a reasonable definition and if you are a human it will do; If you are a processor it just will not fly. That is because humans are great abstract thinkers and computers are not, so it is helpful to view the definition of a program from two points of view.

THE HUMAN PERSPECTIVE

Humans are the masters of abstract thought; it is the hallmark of our intelligence. High-level, object-oriented languages like C++ give us the ability to analyze a problem abstractly and symbolically express its solution in a form that is both understandable by humans and readable by other programs. By other programs I mean the C++ code a programmer writes must be translated from C++ into machine instructions recognizable by a particular processor. This translation is effected by running a compiler that converts the C++ code to object code targeted to a specific machine.

To a C++ programmer a program is a collection of classes that model the behavior of objects in a particular problem domain. These classes model object behavior by defining object attributes (data) and class methods to manipulate the class attributes. On an even higher level, a program can be viewed as an interaction between objects. This view of a program is convenient for humans.

THE COMPUTER PERSPECTIVE

From the computer's perspective a program is simply machine instructions and data. Usually both the instructions and data reside in the same memory space. This is referred to as a Von Neumann architecture. In order for a program to run it must be loaded into main memory and the address of the first instruction of the program given to the processor. In the early days of computing programs were coded into computers by hand and then executed. Nowadays all the nasty details of loading programs from auxiliary memory into main memory are handled by an operating system, which, by the way, is a program.

Since both instructions and data reside in main memory how does a computer know when it is dealing with an instruction or with data? The answer to this question will be discussed in detail below but here's a quick answer: It depends on what the computer is expecting. If a computer reads a memory location expecting to find an instruction and it does, everything runs fine. The instruction is decoded and executed. If it reads a memory location expecting to find an instruction but it is not an instruction, then the decode fails and the computer might lock up!

CONCEPT OF OBSERVABLE BEHAVIOR

When you write a C++ program you will use a compiler to translate the source code into a machine readable form. The compiler you use represents an implementation instance of an abstract machine as defined by the ANSI C++ standard.

Various aspects of the abstract machine fall into three categories. They are either implementation-defined, unspecified, or undefined. What does this mean to you the programmer? It means that each compiler writer may implement the abstract machine differently but, given a well-formed C++ program, each implementation should produce the same observable behavior. The best definition of observable behavior comes straight from the ANSI C++ standard.

The observable behavior of the abstract machine is its sequence of reads and writes to volatile data and calls to library I/O functions.

In other words, if you write a well-formed C++ program and compile it with compiler A, and again with another compiler B, both compilers A and B should produce an executable program that when run demonstrates the same observable behavior regardless how the writers of each compiler implemented the operation of the abstract machine. Each compiler manufacturer documents how their product implements the abstract machine. I recommend you get to know your development environment inside and out.

THE C++ PROGRAM TRANSFORMATION PROCESS

C++ programs are translated into executable modules via a nine phase process. Each phase is briefly discussed below and illustrated in figure 4-6.

PHASE 1

In phase 1, physical source file characters are mapped to the basic source character set. The basic source character set includes the following 91 graphical characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " ' "
```

In addition to these there are five non-graphical characters: space, horizontal tab, vertical tab, form feed, and new line.

Next, all trigraph sequences are replaced by corresponding single-character internal representations. Table 4-1 gives the trigraphs and their single replacement characters.

Trigraph	Replacement
??=	#
??/	\
??'	^
??([
??)]
??!	
??<	{
??>	}
??-	~

Table 4-1: Trigraph Replacement

Trigraphs are used by programmers who are programming in C++ on terminals that lack the special characters required by the basic source character set.

PHASE 2

In phase 2, new-line characters and preceding backslashes are deleted.

PHASE 3

In phase 3, the source file is decomposed into preprocessing tokens and sequences of white-space characters. Each comment is replaced by a single space character.

PHASE 4

In phase 4, the preprocessing directives are executed and macros are expanded.

PHASE 5

In phase 5, each source character set member, escape sequence, or universal-character-name in character or string literals is converted to a member of the execution character set. The execution character set is the basic source character set plus the control characters plus a null character.

You were introduced to the basic source character set in phase 1. An escape sequence can be either simple, octal, or hexadecimal. Table 4-2 gives the valid escape sequences.

Character	Abbreviation	Escape Sequence
newline	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
alert	BEL	\a
backslash	\	\\
question mark	?	\?
single quote	'	\'
double quote	"	\"
octal number	ooo	\ooo
hexadecimal number	hhh	\xhhh

Table 4-2: Escape Sequences

A universal character name provides a way to name other characters. A universal character name is formed by a backslash character followed by a lower case u or upper case U, followed by a sequence of 4 or 8 hexadecimal characters. The following is an example of a universal character name:

```
\uAAAAAAAA
```

PHASE 6

In phase 6, adjacent character or string literals are concatenated.

PHASE 7

In phase 7, preprocessing tokens are converted to tokens. The tokens are then syntactically and semantically analyzed and translated.

PHASE 8

In phase 8, translated translation units and instantiation units are combined.

A translation unit consists of a source file, its headers and any source files included via the `#include` preprocessor directive, minus any source lines skipped by conditional compilation.

If a translation unit instantiates template functions or classes then the required templates are located and the instantiations performed.

PHASE 9

In phase 9, external object and function references are resolved, and library components are linked. All output from the translation is combined into a program image. Once the C++ source code has been translated into an executable module targeted for a specific processor it can be executed. The entry point for all C++ programs is the `main()` function.

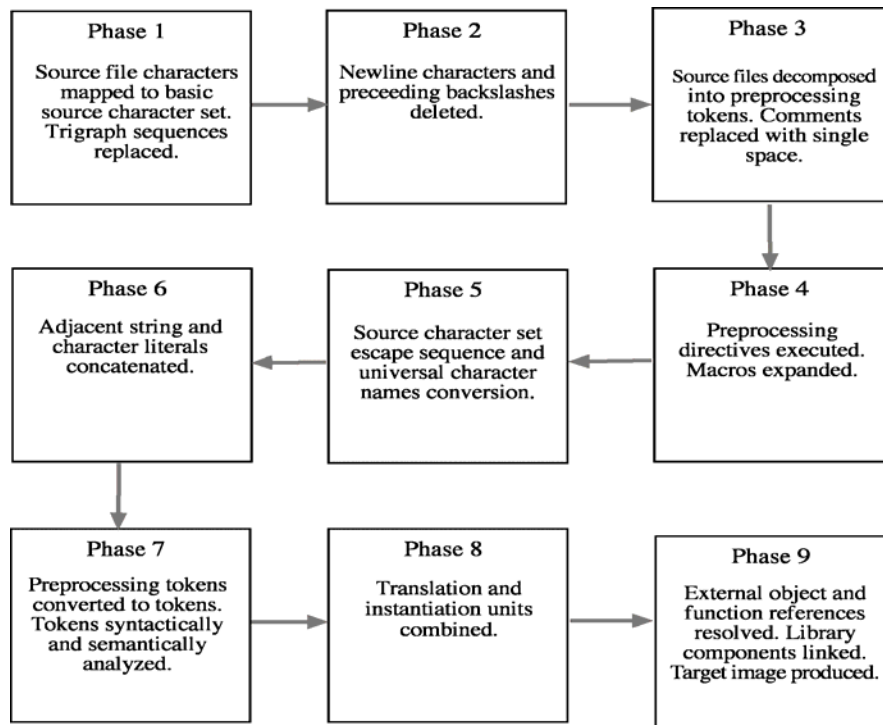


Figure 4-6: C++ Translation Phases

THE PROCESSING CYCLE

Computers are powerful because they can do repetitive things really fast. When the executable code is loaded into main memory the processor can start executing the machine instructions. When a computer executes or runs a program it constantly repeats a series of processing steps commonly referred to as the processing cycle. The processing cycle consists of four primary steps: instruction fetch, instruction decode, instruction execution, and result store. The step names can be shortened to simply fetch, decode, execute, and store. Different processors will implement the processing cycle differently but for the most part these four processing steps are being carried out in some form or another.

FETCH

In the fetch step, an instruction is read from main memory and presented to the decode section of the processor. If the requested memory address contents resides in cache memory the read operation will happen quickly. Otherwise, the processor will have to wait for the data to be accessed from main memory, which has a slower access time than does cache memory.

DECODE

In the decode step, the instruction fetched from main memory is decoded. If the computer thinks it is getting an instruction but instead it gets garbage there will be problems. A computer system's ability to recover from such situations is generally a function of a robust operating system.

EXECUTE

If the fetched instruction is successfully decoded, meaning it is a valid instruction in the processor's instruction set, it is executed. A computer is a bunch of switches. Executing an instruction means the computer's electronic switches are switched either on or off, and clocked in a particular fashion to carry out the particular instruction.

STORE

After an instruction is executed the results of the execution, if any, must be stored somewhere. Most arithmetic instructions leave the result in one of the processor's onboard registers. Memory write instructions would then be used to transfer the results to main memory. Keep in mind that there is only so much storage space inside of a processor. At any given time, almost all data and instructions reside in main memory, and are only loaded into the processor when needed.

Why A PROGRAM CRASHES

Notwithstanding catastrophic hardware failure, a computer crashes or locks up because what it was told was an instruction was not! The faulty instruction loaded from memory turns out to be an unrecognizable string of 1's and 0's and when it fails to decode into a proper instruction the computer halts.

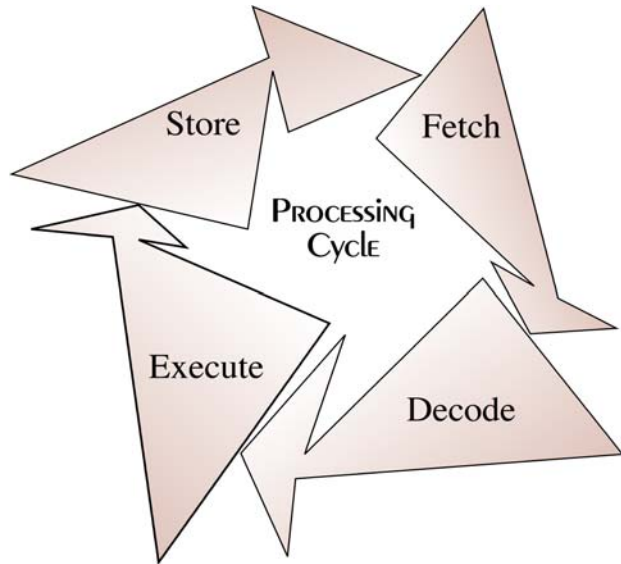


Figure 4-7: Processing Cycle

MEMORY ORGANIZATION

Most modern computer systems have similar memory organizations and as a programmer you should be aware of how computer memory is organized and accessed. The best way to get a good feel for how your computer works is to poke around in memory and see what's in there for yourself. This section provides a brief introduction to computer memory concepts to help get you started.

MEMORY BASICS

A computer's memory stores information in the form of electronic voltages. There are two general types of memory: volatile and non-volatile. Volatile memory will lose any information stored there if power is removed for any length of time. Main memory and cache memory, two forms of Random Access Memory (RAM), are examples of

volatile memory. Auxiliary storage devices such as CD ROMs, DVDs, hard disk drives, floppy disks, and tapes, are examples of non-volatile memory.

MEMORY HIERARCHY

Computer systems contain several different types of memory. These memory types range from slow and cheap to fast and expensive. The proportion of slow cheap memory to fast expensive memory can be viewed in the shape of a pyramid commonly referred to as the memory hierarchy as shown in figure 4-8.

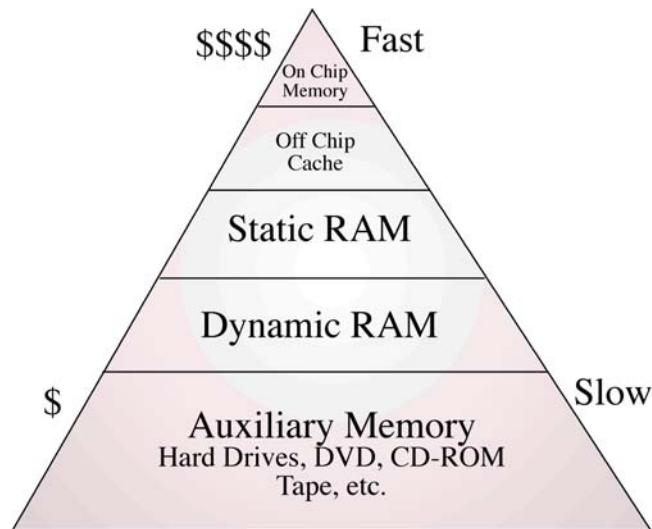


Figure 4-8: Memory Hierarchy

The job of a computer system designer with regards to memory subsystems is to make the whole computer perform as if all the memory were fast and expensive. Thus they utilize cache memory to store frequently used data and instructions and buffer disk reads to memory to give the appearance of faster disk access. Figure 4-9 shows a block diagram of the different types of memory used in a typical computer system.

Faster memory is checked for the requested data or instruction first. If it is not there, a performance penalty is extracted in the form of longer overall access times required to retrieve the information from a slower memory source.

Bits, Bytes, Words

Program code and data are stored in main memory as electronic voltages. Since I'm talking about digital computers the voltages levels represent two discrete states depending on the level. Usually low voltages represent no value, off, or 0, while a high voltage represents on, or 1.

When program code and data is stored on auxiliary memory devices, electronic voltages are translated into either electromagnetic fields (tape drives, floppy and hard disks) or bumps that can be detected by laser beam (CDs, DVDs, etc.).

Bit

The bit represents one discrete piece of information stored in a computer. On most modern computer systems bits cannot be individually accessed from

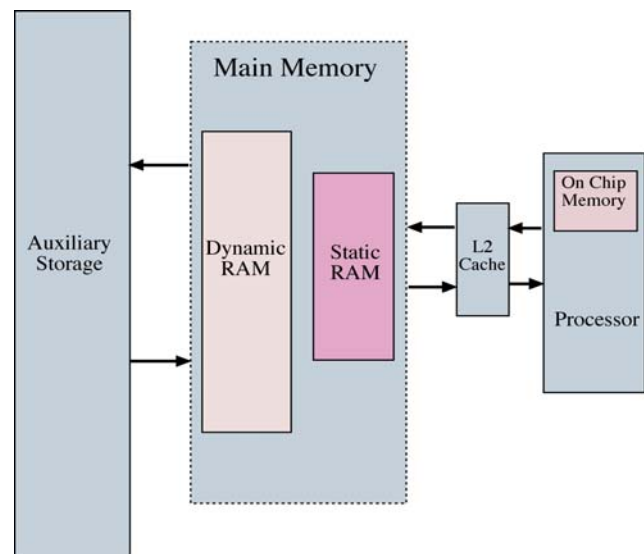


Figure 4-9: Simplified Memory Subsystem Diagram

memory. However, after the byte to which a bit belongs is loaded into the processor the byte can be manipulated to access a particular bit.

Byte

A byte comprises 8 bits. Most computer memory is byte addressable although as processors become increasingly powerful and can manipulate wider memory words, loading bytes by themselves into the processor becomes increasingly inefficient. This is the case with the G4 processor and for that reason the fastest memory reads can be done a word at a time.

Word

A word is a collection of bytes. The number of bytes that comprise a word is computer system dependent. If a computer's data bus is 32 bits wide and its processor's registers are 32 bits wide then the word size would be 4 bytes long. Bigger computers will have larger word sizes meaning they can manipulate more information per unit time than a computer with a smaller word size.

ALIGNMENT AND ADDRESSABILITY

C++ programmers can expect to find the memory on their systems to be byte addressable and word aligned. Figure 4-10 shows a simplified diagram of a main memory divided into bytes and the different buses connecting it to the processor.

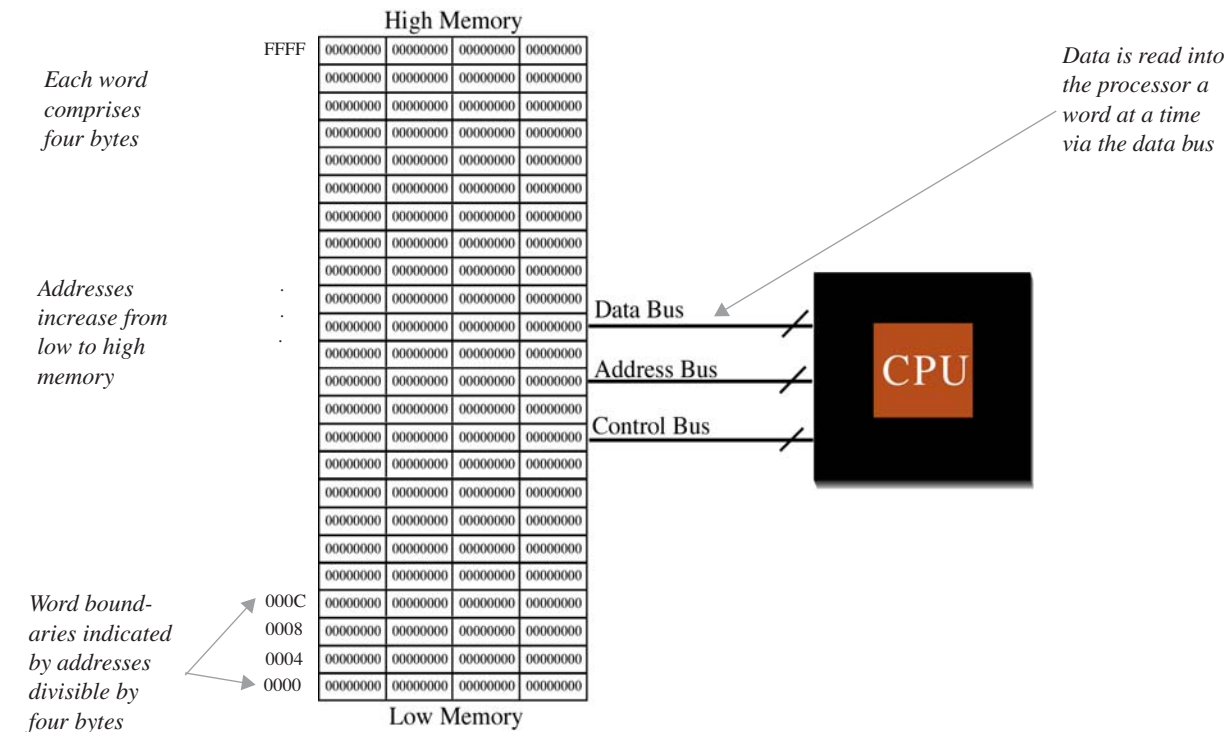


Figure 4-10: Simplified Main Memory Diagram

The memory is byte addressable in that each byte can be individually accessed although the entire word that contains the byte is read into the processor. Data in memory can be aligned for efficient manipulation. Alignment can be to natural or some other boundary. For example, on a PowerPC system, contents of memory assigned to instances of structures is aligned to natural boundaries meaning a one byte data element will be aligned to a one byte boundary. A two byte element would be aligned to a two byte boundary. Individual data elements not belonging to structures are usually aligned to four byte boundaries.

Understanding alignment will come in handy when you start trying to understand where your compiler is putting your program variables. Figure 4-11 shows a project settings window for Metrowerks CodeWarrior™. Notice the tab pointed to by the arrow that says Struct Alignment. The options for this tab are listed in table 4-3.

You can specify structure memory alignment!

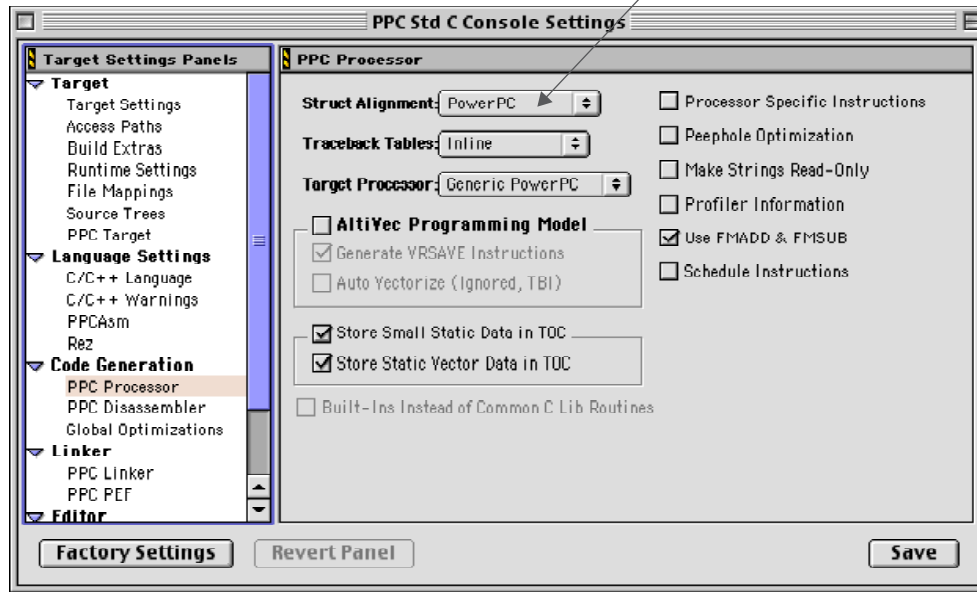


Figure 4-11: CodeWarrior Code Generation Settings Window

Tab Selected...	Results In...
68K (meaning a 68000 processor)	2-byte boundaries, unless a field is only 1-byte long. This is the standard alignment for 68K Macintosh computers.
68K 4-byte	4-byte boundaries
PowerPC	Its natural boundary. For example, it aligns a 1-byte character on a 1-byte boundary and a 16-bit integer on a 2-byte boundary. The compiler applies this alignment recursively to structured data and arrays containing structured data. So, for example, it aligns an array of structured types containing an 8-byte floating point member on an 8-byte boundary. This is the standard alignment for Mac OS computers.

Table 4-3: CodeWarrior Structure Alignment

Algorithms

Computers run programs; programs implement algorithms. A good working definition of an algorithm for the purposes of this book is that an algorithm is a recipe for getting something done on a computer. Pretty much every line of source code you write is considered part of an algorithm. What I'd like to do in this brief section is bring to your attention the concept of good vs. bad algorithms.

Good vs. Bad Algorithms

There are good ways to do something in source code and there are bad ways to do the same exact thing. A good example of this can be found in the act of sorting. Suppose you want to sort in ascending order the following list of integers:

1, 10, 7, 3, 9, 2, 4, 6, 5, 8, 0, 11

One algorithm for doing the sort might go something like this:

Step 1: Select the first integer position in the list

Step 2: Compare the selected integer with its immediate neighbor

Step 2.2: If the selected integer is greater than its neighbor swap the two integers

Step 2.3: Else, leave it where it is.

Step 3: Continue comparing selected integer position with all other integers repeating steps 2.2 - 2.3

Step 4: Select the second integer position on the list and repeat the procedure beginning at step 2.

Continue in this fashion until all integers have been compared to all other integers in the list and have been placed in their proper position.

This algorithm is simple and straightforward. It also runs pretty fast for small lists of integers but it is really slow given large lists of integers to sort. Another sorting algorithm to sort the same list of integers may go as follows:

Step 1: Split the list into two equal sublists

Step 2: Repeat step 1 if any sublist contains more than two integers

Step 3: Sort each sublist of two integers

Step 4: Combine sorted sublists until all sorted sublists have been combined

This algorithm runs a little slow on small lists because of all the list splitting going on but sorts large lists of integers way faster than the first algorithm. The first algorithm lists the steps for a routine I call dumb sort. Example 4.1 gives the source code for a short program that implements the dumb sort algorithm.

4.1 Dumb Sort Test Program

```

1  #include <iostream.h>
2
3  int main(){
4      int a[] = {1,10, 7, 3, 9, 2, 4, 6, 5, 8, 0, 11};
5      int innerloop = 0;
6      int outerloop = 0;
7      int swaps = 0;
8
9      for(int i = 0; i<12; i++){
10         outerloop++;
11         for(int j = 1; j<12; j++){
12             innerloop++;
13             if(a[j-1] > a[j]) {
14                 int temp = a[j-1];
15                 a[j-1] = a[j];
16                 a[j] = temp;
17                 swaps++;}}
18
19         for(int i = 0; i<12; i++)
20             cout<<a[i]<<" ";
21
22         cout<<endl;
23         cout<<"Outer loop executed "<<outerloop<<" times."<<endl;
24         cout<<"Inner loop executed "<<innerloop<<" times."<<endl;
25         cout<<swaps<<" swaps completed."<<endl;
26         return 0;}

```

Included in the dumb sort test source code are a few variables intended to help collect statistics during execution. These are `innerloop`, `outerloop`, and `swaps` declared on lines 5, 6, and 7 respectively. Figure 4-12 gives the results from running the dumb sort test program.

Notice that the inner loop executed 132 times and that 30 swaps were conducted. Can the algorithm run any better? One way to check is to rearrange the order of the integers in the array. What if the list of integers is already sorted? Figure 4-13 gives the results of running dumb sort on an already sorted list of integers:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

It appears that both the outer loop and inner loop are executed the same number of times in each case, which is of course the way the source code is written, but it did run a little faster because fewer swaps were necessary.



Figure 4-12: Dumb Sort Results 1

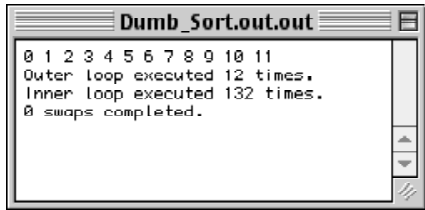


Figure 4-13: Dumb Sort Results 2

Can the algorithm run any worse? What if the list of integers is completely unsorted? Figure 4-14 gives the results of running dumb sort on a completely unsorted list:

11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

The outer loop and inner loop executed the same number of times but 66 swaps were necessary to put everything in ascending order. So it did run a little slower this time.

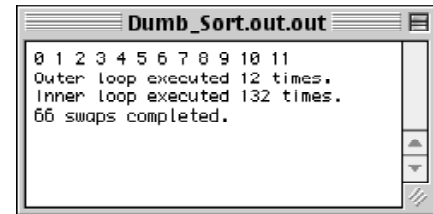


Figure 4-14: Dumb Sort Results 3

In dumb sort, because we're sorting a list of 12 integers, the inner loop executes 12 times for every time the outer loop executes. If Dump Sort needed to sort 10,000 integers then the inner loop would need to execute 10,000 times for every time the outer loop executed. To generalize the performance of dumb sort you could say that for some number N integers to sort, dumb sort executes the inner loop roughly $N \times N$ times. There is some other stuff going on besides loop iterations but when N gets really large, the loop iteration becomes the overwhelming measure of dumb sort's performance as a sorting algorithm. Computer scientists would say that dumb sort has order N^2 performance. Saying it another way, for a really large list of integers to sort, the time it takes dumb sort to do its job is approximately the square of the number N of integers that need to be sorted.

When an algorithm's running time is a function of the size of its input the term used to describe the growth in time to perform its job vs. the size of the input is called the growth rate. Figure 4-15 shows a plot of algorithms with the following growth rates: $\log n$, n , $n \log n$, n^2 , n^3 , n^n

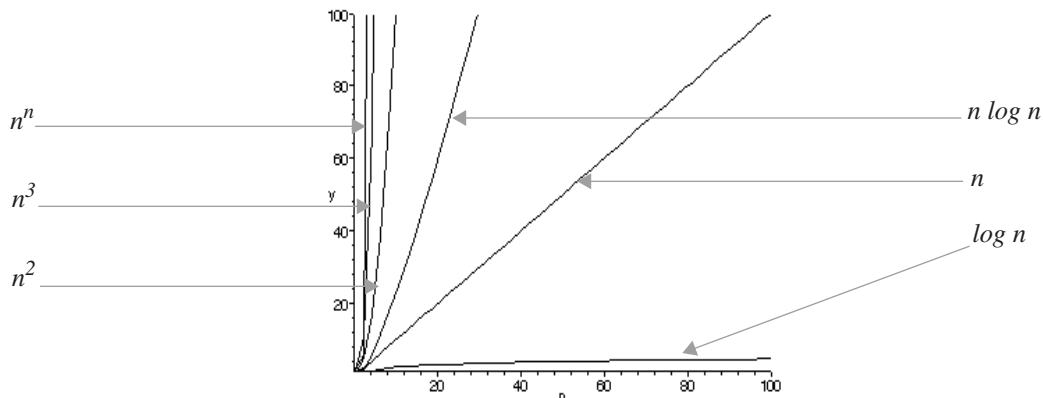


Figure 4-15: Algorithmic Growth Rates

As you can see from the graph, dumb sort, with a growth rate of n^2 , is a bad algorithm, but not as bad as some other algorithms. The good thing about Dumb Sort is that no matter how big its input grows, it will eventually sort all the integers. Sorting problems are easily solved. There are some problems, however, that defy straightforward algorithmic solution.

DON'T REINVENT THE WHEEL!

If you are new to programming the best advice I can offer is for you to seek the knowledge of those who have come before you. There are many good books on algorithms, some of which are listed in the reference section. Studying good algorithms helps you write better code.

SUMMARY

Computers run programs; programs implement algorithms. As a programmer you need to be aware of development issues regarding your computer system, the processor it is based on, and the programming language.

A computer system comprises a processor, I/O devices, and supporting operating system software. The processor is the heart of the computer system.

Programs can be viewed from two perspectives: human and computer. From the human perspective, programs, are a high-level solution statement to a particular problem. Object-oriented languages like C++ help humans model extremely complex problems algorithmically. C++ programs can also be viewed as the interaction between objects in a problem domain.

To a computer, programs are a sequence of machine instruction and data located in main memory. Processors run programs by rapidly executing the processing cycle of fetch, decode, execute, and store. If a processor expects an instruction and gets garbage it is likely to halt processing. Robust operating systems can mitigate this problem to a certain degree.

C++ programs are translated into machine code via a nine phase process.

There are bad algorithms and good algorithms. Study from the pros and you will improve your code writing skills.

Skill Building Exercises

1. **Research Sorting Algorithms:** The second sorting algorithm listed on page 86 gives the steps for a merge sort. Obtain a book on algorithms, look for some C++ code that implements the merge sort algorithm, and compare it to Dumb Sort. What's the growth rate for a merge sort algorithm? How does it compare to Dumb Sort's growth rate?
2. **Research Sorting Algorithms:** Look for an example of a bubble sort algorithm. How does the bubble sort algorithm compare to Dumb Sort? What small changes can be made to Dumb Sort to improve its performance to that of bubble sort? What percentage of improvement is obtained by making the code changes? Will it make difference for large lists of integers?

SUGGESTED PROJECTS

1. **Research Computer System:** Research your computer system. List all its components including the type of processor. Go to processor manufacturer's web site and download developer information for your systems processor. Look for a block diagram of the processor and determine how many registers it has and their sizes. How does it get instructions and data from memory? How does it decode the instructions and process data.
2. **Compare Different Processors:** Select two different microprocessors and compare them to each other. List the feature set of each and how the architecture of each implements the feature set.

SELF TEST QUESTIONS

1. List at least five components of a typical computer system.
2. What device do the peripheral components of a computer system exist to support?
3. From what two perspectives can programs be viewed? How does each perspective differ from the other?
4. List the nine phases of the C++ translation process.
5. What are the function of trigraphs?
6. What is a C++ translation unit?
7. List and describe the four steps of the processing cycle?
8. State in your own words the definition of an algorithm.
9. How does a processor's architecture serve to implement its feature set?
10. How can programmers access a processor's feature set?

REFERENCES

Motorola. PowerPC 601 RISC Microprocessor User's Manual

Motorola. PowerPC 7400 RISC Microprocessor User's Manual

Sedgewick, Robert. *Algorithms in C++*, Addison-Wesley Publishing Company, Reading Massachusetts, 1992, ISBN 0-201-51059-6.

Corman, Thomas, et. al. *Introduction To Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990, ISBN 0-262-03141-8

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

NOTES

PART II: C++ LANGUAGE FUNDAMENTALS

CHAPTER 5



Fiery DESERT Night

Simple PROGRAMS

LEARNING OBJECTIVES

- Describe what constitutes a minimum well-formed C++ program
- List the keywords reserved for use by the C++ language
- State the purpose of variables, constants, expressions, and statements
- Demonstrate your ability to declare, define, and use variables
- Demonstrate your ability to declare, define, and use constants
- List and describe the purpose of the C++ fundamental data types
- Determine data type sizes with the sizeof operator
- Utilize variables and constants in simple C++ programs
- List the native C++ operators and state their precedence
- Write C++ programs using simple and compound statements
- Describe variable scoping and state how the block structure of C++ can affect variable visibility
- Utilize simple input and output techniques using the cin and cout objects
- Describe the required parts of a minimal C++ program
- Utilize an IDE's disassembly tool to gain deeper understanding of C++ program structure
- List and describe the parts of a typical C++ program to include source files, main() function, library files, and preprocessor directives

INTRODUCTION

This is the most important chapter in this book! A thorough understanding of fundamental C++ language features such as variable and constant declaration, variable scoping, operator usage, and statement construction, will provide you with the background required to enhance your understanding of the more complicated aspects of the language.

My intent in this chapter is to present to you material that in most C++ courses is either glossed over or skipped entirely because it is assumed the student will learn it along the way. This mode of thinking isn't entirely faulty, but the unfortunate result from such an approach is the formulation of bad programming habits early in a student's career. Bad programming habits are difficult to correct and inhibit one's ability to understanding the more salient, and powerful, features of C++.

Keeping the above in mind, resist the urge to skip over material you think seems too easy. If you must challenge the chapter, proceed to the skill building exercises, suggested projects, or self test question sections to see if you are in fact ready to move along with your studies. You cannot make a mistake by giving yourself extra time to dwell on the material covered in this chapter.

A MINIMAL C++ PROGRAM

The following source code is an example of the smallest C++ program you can write.

```
int main() { }
```

The program doesn't do much yet everything is there. Every C++ program must have one, and only one, main() function. The main() function must return an integer value. The value returned by the main() function is a result code intended for use by the operating system. The result code gives some indication of how the program terminated. Readers with some experience will note the absence of a return statement in the body of main(). A return statement is not necessary because the compiler does the job for you by default. You can compile and run this minimal program but on most computers you'd receive no indication it had executed.

Example 5.1 gives a disassembled version of the minimal main() function targeted for a PowerPC processor using Metrowerks CodeWarrior's disassembly feature.

*5.1 Disassembled Minimal
main() Function*

```
Names:
    1: .main
    2: main
    3: TOC

Hunk:  Kind=HUNK_GLOBAL_CODE   Align=4   Class=PR   Name=".main" (1)   Size=8
00000000: 38600000  li          r3,0
00000004: 4E800020  blr

Hunk:  Kind=HUNK_GLOBAL_IDATA   Align=4   Class=DS   Name="main" (2)   Size=8
00000000: 00 00 00 00 00 00 00 00  '.....'
XRef:  Kind=HUNK_XREF_32BIT      Offset=$00000000   Class=PR   Name=".main" (1)
XRef:  Kind=HUNK_XREF_32BIT      Offset=$00000004   Class=TC0  Name="TOC" (3)

Hunk:  Kind=HUNK_GLOBAL_IDATA   Align=4   Class=TC0  Name="TOC" (3)   Size=0
```

If you feel intimidated by example 5.1 don't be. It looks complicated because to understand everything shown requires some knowledge of the PowerPC processor and the Macintosh G4 system architecture, but for now, a quick explanation of what's going on will suffice.

Example 5.1 contains five sections but I will only discuss the first two sections for now. Starting from the top, the first section is a name table containing three entries: `.main`, `main`, and `TOC`. (TOC stands for Table of Contents) The next section, prefixed with the label `Hunk`: is the code section. Its class is `PR`, meaning it is a Read/Write Control Section and its name is `.main`. In this section there appear two instructions. The first instruction, `li`, located at offset `00000000`, is an extended PowerPC mnemonic meaning Load Immediate. Further to the right of the `li` instruction are two operands, `r3`, which stands for register 3, and `0` which is the integer value zero. When this line is executed the integer value zero will be loaded into register 3.

The next line contains the extended mnemonic `blr` which stands for Branch Link Register. The link register is a special register in the PowerPC processor that will contain the memory address of the next instruction of the calling routine. When this line is executed the program will return, or jump back to the program that called it. The fact that the calling routine requires the result code to be located in register 3 upon program termination is a matter of protocol.

Putting the two instructions together yields the following behavior from the minimal `main()` function: When the program is run, the value zero is loaded into general purpose register 3 and the program jumps back from whence it came. The calling program may or may not make use of the result code.

DISASSEMBLY IS A GREAT LEARNING TOOL

A great way to dig deeper into the workings of your computer and the C++ language is to disassemble your source code. Example 5-1 was generated with CodeWarrior's disassembly feature. To set the desired output format for disassembled code listings from the Edit menu select the PPC Std C++ Console Settings... item as shown in Figure 5-1.

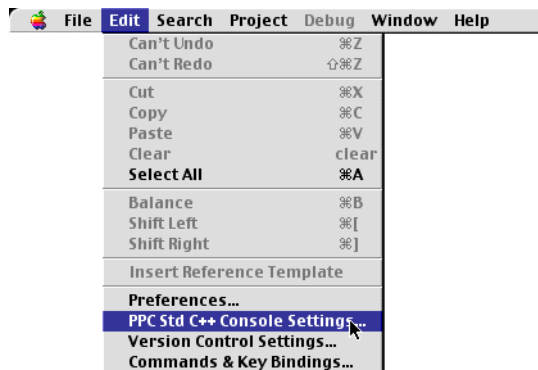


Figure 5-1. Selecting Std C++ Console Settings

If you are targeting a PC then the options you will see when editing the Std C++ Console Settings will be different from those shown here. Regardless, the concepts are the same.

Once you have selected the desired disassembler options save the settings and exit the Std C++ Console Settings dialog. You are now ready to disassemble a source file.

This will bring up the PPC Std C++ Console Settings dialog box shown in figure 5-2. In the Code Generation section select the PPC Disassembler and select the options shown. It is also a good idea to select the other options and then disassemble the source code to see what effects each option has on code generation.

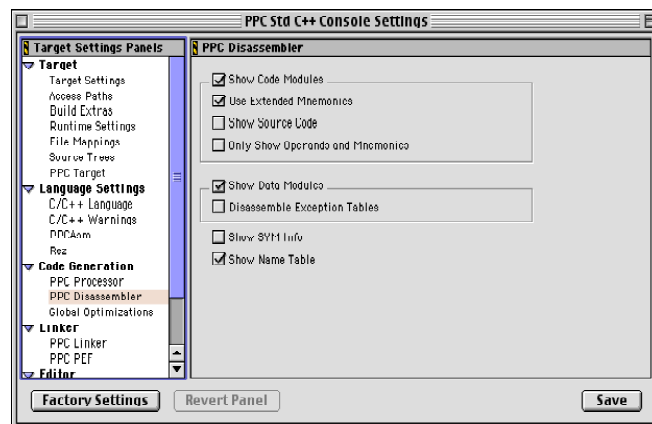


Figure 5-2. PPC Std C++ Console Settings Dialog

Select the source file you wish to disassemble from those listed in your project window. For this example I created a project called Minimal Program, created one file called `main.cpp` that contains the simple `main()` function and located it in the Sources group. The Minimal Program project window is shown in figure 5-3 with the `main.cpp` file highlighted.

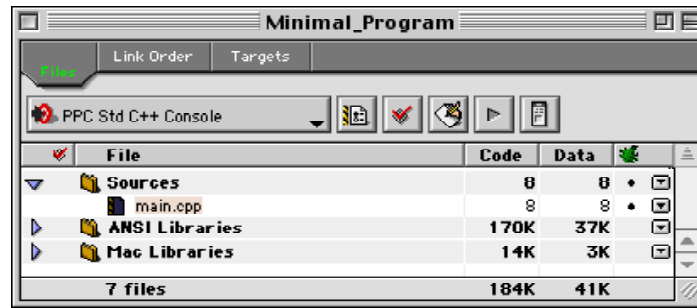


Figure 5-3. Minimal_Program Project Window

Next choose Disassemble from the Project menu as shown in figure 5-4

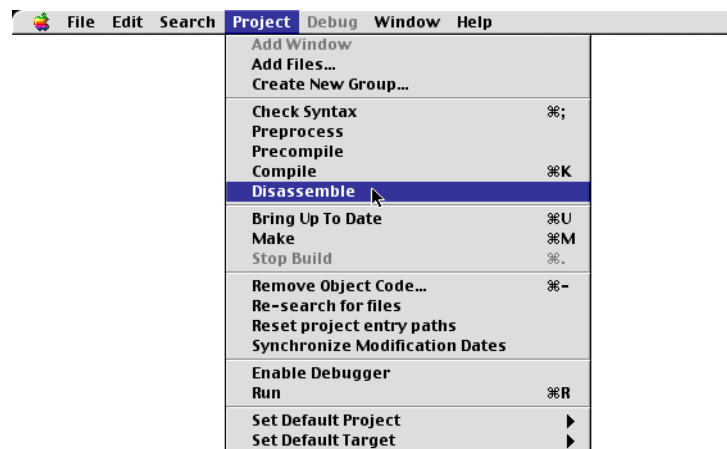


Figure 5-4. Selecting Disassemble from the Project Menu

Disassembling the main.cpp file results in the output shown in example 5-1 above. That's all there is to it! Use the disassemble feature to explore your code to get a better idea of how it works on the inside.

ANOTHER C++ PROGRAM

Example 5.2 gives the source code for another short C++ program. Although short, it has a little more meat to it than the minimal program discussed above.

```

1  /*****
2      Simple C++ Program
3  *****/
4
5  #include <iostream>
6
7  using namespace std;
8
9  int main(){
10     const int const_val = 100;
11     int i;
12     i = 10;
13
14     cout<<"This is a simple C++ program!"<<endl;
15     cout<<"The value of i is: "<<i<<endl;
16     cout<<"The value of const_val is: "<<const_val<<endl;
17
18     return 0;
19 }
```

5.2 Another C++ Program

PARTS OF THE PROGRAM

Let us discuss example 5.2 line by line. The numbers 1 through 19 in *italics* to the left of the listing are line numbers and are not part of the source code.

COMMENTS

Line 1 begins with the `/*` characters and is the start of a C-style comment. The comment proceeds to the end of line 3 which ends with the `*/` characters. Everything appearing between the `/*` and `*/` characters is ignored by the compiler. If you were to disassemble this code the comments would be omitted.

PREPROCESSOR DIRECTIVE

Line 5 begins with the preprocessor directive `#include` and names a library header file called `iostream`. The library header file named `iostream` is enclosed in the `< >` character pair telling the preprocessor how it should conduct its search for the `iostream` file. If you are using an older development environment you may have to use the filename `iostream.h` to access the `iostream` library.

LIBRARIES

Libraries, like `iostream`, are code modules that implement some type of functionality and can be incorporated into your programs. To gain access to a library the compiler must know where to find it and its header file must be included in the source code via an `#include` preprocessor directive.

USING DIRECTIVE

Line 7 contains a `using` directive indicating that the namespace `std` (the standard namespace) is being used. This is required to gain access to various `iostream` objects. One `iostream` object, `cout`, is being used on lines 14, 15, and 16 to send stream output to the standard output device. The standard output device in this case is the computer screen.

MAIN() FUNCTION

Line 9 contains the start of the `main()` function. The body of the `main()` is comprised of everything appearing between the opening left brace `{` at the end of line 9, and closing right brace `}` on line 19.

CONSTANTS

Line 10 declares an integer constant named `const_val` and defines its value as 100. A constant is an object whose value is to remain unchanged during its lifetime. The keyword `const` is also used to prevent dumb programming mistakes or to enforce good program design.

VARIABLES

Line 11 declares an integer variable named `i`. On line 12 `i` is assigned the value 10. Unlike a constant, a variable is an object whose value is allowed to change during its lifetime.

STATEMENTS AND EXPRESSIONS

Lines 10 through 12, lines 14 through 16, and line 18 each contain a statement. Line 12 is an example of a statement that is also an expression.

KEYWORDS

C++ programs, like example 5-2, are constructed using certain identifiers that are reserved for use by the C++ compiler. These reserved words are known as keywords. You can name your variables and constants anything you like so long as you avoid using keyword names. The keywords are listed in the first four columns of the C++ keyword list below. The fifth column lists identifier names that are reserved to implement alternative representations of certain operators and punctuators. The alternative representation identifiers may or may not be implemented by the compiler you are using. Regardless, you are advised not to create user-defined types using these reserved names because doing so may break your program sometime in the future.

asm	else	new	template	
auto	enum	operator	this	
bool	explicit	private	throw	
break	export	protected	true	
case	extern	public	try	
catch	false	register	typedef	and
char	float	reinterpret_ca	typeid	and_eq
class	for	st	typename	bitand
const	friend	return	union	bitor
const_cast	goto	short	unsigned	compl
continue	if	signed	using	not
default	inline	sizeof	virtualvoid	not_eq
delete	int	static	volatile	or
do	long	static_cast	wchar_t	or_eq
double	mutable	struct	while	xor
dynamic_cast	namespace	switch		xor_eq

C++ Keyword List

FUNDAMENTAL TYPES

C++ comes with several built-in data types ready for immediate use. Each of the character, integer, and floating point types has a corresponding range of values they should represent. I use the word “should” because each compiler implementation can choose to increase the value range of a particular type so long as they meet certain American National Standards Institute (ANSI) requirements as specified in the C++ standard.

Your compiler defines its fundamental data type value ranges in the following header files: `climits.h`, `limits.h`, and `cfloat.h`. The value ranges of each type is largely hardware dependent. For example, a 32-bit processor will have a smaller maximum long integer value than a 64-bit processor, assuming the compiler is keeping pace with the hardware it is running on.

Table 5-1 gives the type value ranges for the Metrowerks CodeWarrior C++ compiler version 5.0.

Type	Minimum Value	Maximum Value
bool	false (0)	true (1)
signed char	-128	127
char	-128	127

Table 5-1: Fundamental Types and their Value Ranges

Type	Minimum Value	Maximum Value
unsigned char	0	255
wchar_t	0	65,535
signed short int	-32,768	32,767
short int	-32,768	32,767
unsigned short int	0	65,535
signed int	-2,147,483,648	2,147,483,647
int	-2,147,483,648	2,147,483,647
unsigned int	0	4,294,967,295
signed long int	-2,147,483,648	2,147,483,647
long int	-2,147,483,648	2,147,483,647
unsigned long int	0	4,294,967,295
signed long long int	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
long long int	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long int	0	18,446,744,073,709,551,615
float	1.17549e ⁻³⁸	3.40282e ⁺³⁸
double	2022507e ⁻³⁰⁸	1.079769e ⁺³⁰⁸
long double	2022507e ⁻³⁰⁸	1.079769e ⁺³⁰⁸

Note: These ranges are valid for Metrowerks CodeWarrior C++ version 5.0 for Macintosh. Your ranges may look similar or somewhat different for some of the larger types depending on your compiler implementation.

Table 5-1: Fundamental Types and their Value Ranges

DETERMINING YOUR DATA TYPE RANGES

There are a couple of ways to determine your data type ranges. The first is by inspecting the `climits.h` and `cfloat.h` header files. In them you will find the valid ranges listed for each fundamental type. If you are not used to reading commercial grade header files this is an excellent exercise. Another benefit to this approach is that it lets you get better acquainted with your development environment.

The second way to determine your fundamental type ranges is by calculation. The `limits.h` file in newer implementations of C++ defines the `numeric_limits` template class. Example 5.3 gives a short program showing you how to use the `numeric_limits` class to calculate a type's range. If you don't feel comfortable using template classes just yet don't worry. The inspection method is all you need for now.

The line numbers to the left of example 5.3 are not part of the source code.

```

1  #include <iostream>
2  #include <limits>
3
4  using namespace std;
5
6  int main(){
7      numeric_limits<int> _i;
8      cout<<"Integer Range: "<<_i.min()<<" "<<_i.max()<<endl;
9      return 0;
10 }
```

*5.3 Using numeric_limits
Template Class to Calculate
Type Ranges*

Running this short program produces the output shown in Figure 5-5.

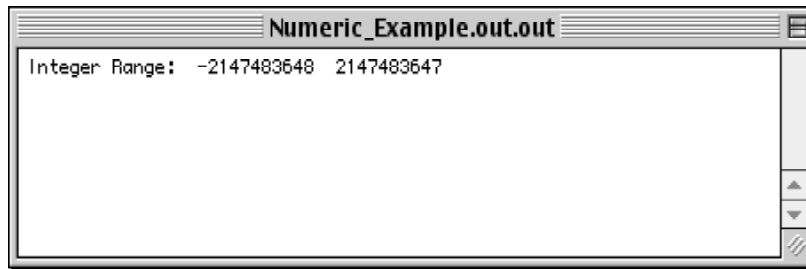


Figure 5-5: Results of Running Example 5.3

I'd like to take a moment to explain what's happening here. On line 7 a `numeric_limits<int>` object named `_i` (pronounced underscore i) is declared. The `numeric_limits<int>` type is an example of how to create a new type with a template class. The `numeric_limits< >` class has several methods a programmer can use to get implementation specific information regarding fundamental data types. Two of these methods are used on line 8. The first is `min()`, and the second is `max()`. Calling the `min()` and `max()` functions on the `_i` object results in the output of an integer's minimum value and its maximum value.

The `numeric_limits<>` class can be used on all the C++ data types. For example, if you need information regarding the `char` type, just use `char` instead of the `int` type used in example 5.3 when creating the `numeric_limits` object. Expanding example 5.3 to print out all the data type ranges for your compiler implementation is left as an exercise.

DETERMINING DATA TYPE SIZE WITH THE `sizeof` OPERATOR

The size of the data type directly determines the range of values it can effectively store. The bigger the type, the more bits it has available to represent data. If you need to know how large a particular data type is in bytes you can use the `sizeof` operator. To use the `sizeof` operator first declare an object of a particular type then use the `sizeof` operator on the object. Example 5.4 shows a short program that declares an integer object and then uses the `sizeof` operator to report its size:

```
#include <iostream>
using namespace std;
```

```
int main(){
    int i = 0;
    cout<<sizeof i<<endl;
    return 0;
}
```

5.4 Using the `sizeof` Operator

← Declare and initialize object

← Use `sizeof` to report size of object in bytes.

LITERALS

You often need to represent data “literally” in your program. The following statement gives an example:

```
int i = 25;
```

The integer value 25 is being assigned to the newly declared integer variable named `i`. When numbers, characters, and strings of characters appear directly in source code, like the 25 does in this example, they are called literals. Here, the value 25 is known as a *decimal integer literal*.

INTEGER LITERALS

There are three types of integer literals: decimal, octal, and hexadecimal.

Decimal

You have already seen an example of a decimal integer literal above. Decimal literals must start with a non-zero digit. Digits following the first can be zero or non-zero. Decimal literals can have a suffix of L or l, or U or u, attached marking them as long or unsigned. You can combine the suffixes to indicate a decimal literal is both unsigned and long. Decimal literals have no decimal point. Here are a few more examples of decimal literals:

```
123L
 34
2887934567u1
 281
```

Octal

Octal literals are formed beginning with a zero digit prefix followed by octal digits. The octal digits include the integers 0 - 7. They can have the same suffixes attached as their decimal counterparts. Examine the following octal literal examples:

```
0123L
 034
 010
0673467UL
```

Hexadecimal

Hexadecimal literals are formed beginning with the zero digit and the letter x prefix. The x can be lowercase or uppercase. The prefix is followed by hexadecimal digits. The hexadecimal digits include the digits 0-9 and the letters a-f or A-F. Hexadecimal literals can be suffixed like their octal and decimal counterparts. Here are a few examples:

```
0x123L
0X1a2b3c4dL
0x2887934567u1
0x281
```

A WORD OF CAUTION

Always be aware of the size of your integer literal and how it is being used. If you are assigning it to a variable make sure the variable's type is large enough to hold the literal's complete value. For example, the following statement attempts to assign a large literal to a variable whose type isn't big enough to sufficiently store it:

```
unsigned short small_fry = 0xffffffff;
```

In this case, `small_fry` is an unsigned short which is only two bytes long. The literal being assigned is four bytes long. Assigning big literals to objects with insufficient storage capacity will result in truncation of the literal value and no compiler warning. The following statement will work fine:

```
unsigned int tough_guy = 0xffffffff;
```

In this case `tough_guy` is an unsigned integer type that's four bytes long.

CHARACTER LITERALS

Character literals are formed by enclosing characters between single quotes. Essentially four things can appear between the single quotes: a single character or multiple characters, a simple escape sequence, an octal escape sequence, or a hexadecimal escape sequence. Escape sequences provide a way of representing special characters or characters not otherwise represented in the implementation character set.

Single Character Literals

Single character literals are formed by enclosing a single character between single quotes. Single character literals are of type `char`. Here are a few examples of single character literals:

```
'a'
```

```
'1'
```

```
't'
```

```
'?'
```

Multiple Character Literals

Multiple character literals are formed by enclosing more than one character between single quotes. Multiple character literals have the type and the numeric value they contain is implementation dependent. You normally don't use multiple character literals. For example, the following statement attempts to assign a multiple character literal to a `char` variable:

```
char c = 'Help';
```

In this example the multiple character literal consists of four chars. Each char takes up a byte of storage. The `char` variable is one byte. There's just not enough space. After making this assignment, printing the variable `c` results in the character `p` being printed to the screen.

What's happening here is that value of the characters between the single quotes is being assigned to the `char` variable. If you change the type from `char` to `int` there will be no loss of data in the conversion. The following source code gives an example:

```
int c = 'Help';
```

Now when the assignment is made there is no loss of data. But what value will the variable `c` contain? It is not the four characters 'Help', rather, it is the value of the 32-bit word that contains the ASCII value of 'H' in the most significant byte, followed by the ASCII value of 'e' in the next byte, followed by the ASCII value of 'l' in the third byte, followed lastly by the ASCII value of 'p' in the least significant byte.

Figure 5-6 illustrates how the integer value of 'Help' can be manually calculated. Determine the hexadecimal value for each character from an ASCII table. The conversion from hexadecimal to binary is then a straightforward operation. The binary positional values are then calculated and added yielding a total of 1,214,606,448.

The important thing to remember about multiple character literals is that they are not strings of characters although a quick glance misleads the uninitiated to believe otherwise.

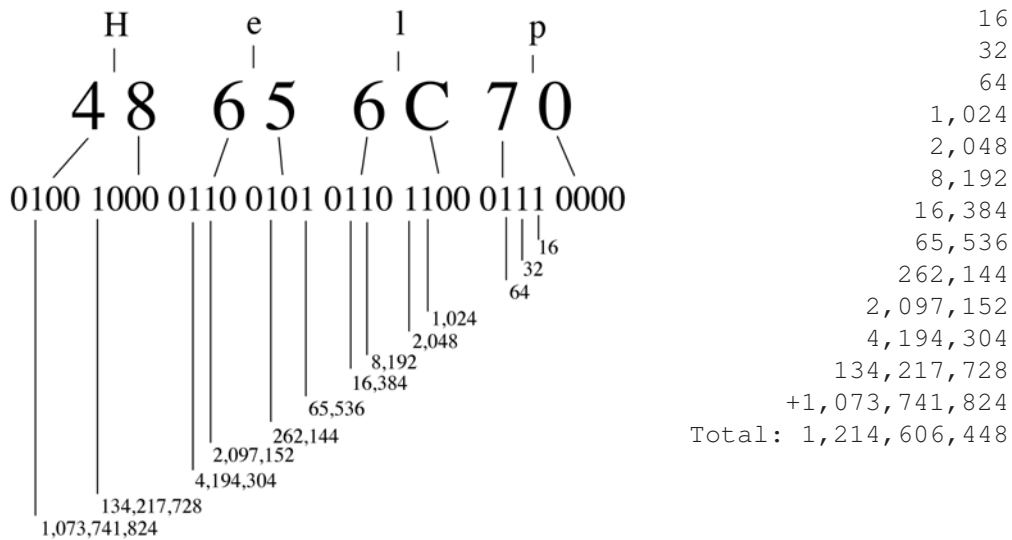


Figure 5-6: Integer Value of Character Literal 'Help'

ESCAPE SEQUENCES

Character literals can also be represented as escape sequences. An escape sequence begins with the backslash character '\'. Three different things can follow the backslash character: a character that together with the backslash results in a simple escape, a series of octal digits resulting in an octal escape, or a series of hexadecimal digits resulting in a hexadecimal escape.

SIMPLE ESCAPE SEQUENCES

Table 5-2 lists the simple escapes.

Escape Sequence	Meaning
\n	Newline
\t	Horizontal Tab
\v	Vertical Tab
\b	Backspace
\r	Carriage Return
\f	Form Feed
\a	Alert
\\	Backslash
\?	Question Mark
\'	Single Quote
\"	Double Quote

Table 5-2: Simple Escape Sequences

Simple escape sequences must be enclosed in single quotes. The following statement gives an example of a simple escape in use:

```
char c = '\'';
```

The character being assigned to the variable `c` is the single quote.

OCTAL ESCAPE SEQUENCES

Octal escape sequences are formed with the backslash character followed by up to three octal digits. Octal escape sequences must be enclosed in single quotes. The following statement gives an example of an octal escape:

```
char c = '\230';
```

Printing the variable `c` with this octal value results in the character ò being printed to the screen on a Macintosh.

HEXADECIMAL ESCAPE SEQUENCES

Hexadecimal escape sequences are formed with the backslash character followed by the character `x`, then a sequence of hexadecimal digits. Hexadecimal escape sequences must be enclosed in single quotes. Here's an example:

```
char c = '\xC0';
```

Printing the variable `c` with this hexadecimal value results in the character ù being printed to the screen.

FLOATING POINT LITERALS

Figure 5-7 dissects a floating point literal.

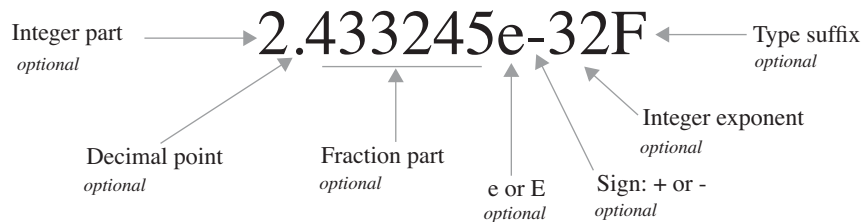


Figure 5-7: Parts of a Floating Point Literal

Well...everything's not optional at the same time. There are some rules. The integer part or the fraction part can be omitted but not both at the same time. The (decimal point) or (the letter `e` and the exponent) can be omitted. I added the parenthesis for clarification of grouping.

The natural type for a floating point literal is double. It can be changed to float or long double by adding the suffix `F` or `L` respectively. Suffixes can be in upper or lower case. Here are a few examples of floating point literals:

```
2e+3
-2e3f
.357E5
5.0L
```

Notice that you can sign the integer part.

STRING LITERALS

String literals are sequences of characters enclosed in double quotes. String literals are automatically terminated with a null character ‘\0’. This is important in that what differentiates strings from ordinary arrays of characters is the presence of the null terminator at the end of each string. The null terminator is used by string processing programs to determine the end of the string. Here’s another very important characteristic of string literals to keep in mind: A string literal is an “lvalue” as opposed to all the other types of literals which are “rvalues”. Let us take a closer look at string literals.

The following statement assigns a string literal to an array:

```
char char_array[] = "Hello World!";
```

The next statement prints the contents of `char_array` to the screen:

```
cout<<char_array<<endl;
```

The following statement will print the size of `char_array` to the screen:

```
cout<<sizeof char_array<<endl;
```

The value printed by the previous statement is 13. This leads to an important property of strings: The size of a string is the number of characters the string contains plus the null terminator.

To embed double quotes in a string literal use an escape sequence. The following statement gives an example:

```
cout<<"The man yelled, \"Run Jimmy, run!\""<<endl
```

BOOLEAN LITERALS

The boolean values 1 and 0 are represented by the boolean literals *true* and *false*. The following statement gives an example of their use:

```
bool keep_going = true;
```

In this example, a boolean variable named `keep_going` is declared and assigned the value 1 or `true`. The following statement prints the value of the variable `keep_going` to the screen:

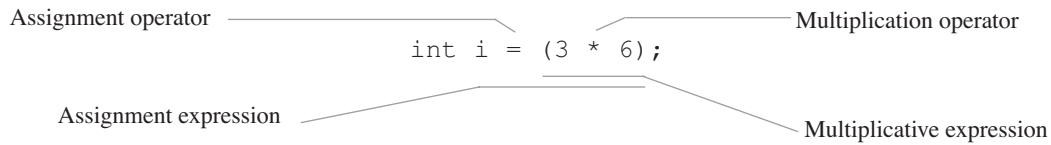
```
cout<<keep_going<<endl;
```

The following statement will print the values of the boolean literals `true` and `false` to the screen:

```
cout<<true<<" "<<false<<endl;
```

EXPRESSIONS

Expressions are built using operators and operands. The operators and operands in the expression specify a computation from which a value may result. The following statement offers several examples:



In this example, the two integer literals 3 and 6 are the operands to the multiplication operator. The resulting value from the multiplicative expression becomes one of the operands to the assignment operator. The other operand is the integer variable named `i` which is declared in the same statement.

There are many types of expressions and, in my opinion, they are best learned gradually rather than all at once. However, it will be helpful to get a feel for the different expression forms you will encounter as you learn C++. Table 5-3 lists different expression forms along with an example.

Expression Form	Examples
Primary	“Literals are examples of primary expressions!”
Postfix - subscripting - function call - explicit type conversion(functional notation) - pseudo destructor call - class member access - increment - decrement - dynamic cast - type identification - static cast - reinterpret cast - const cast	<code>my_array[3]</code> <code>printScreen()</code> <code>float(int_val)</code> <code>my_class.~my_class()</code> <code>my_class.printScreen()</code> <code>count++</code> <code>count--</code> <code>dynamic_cast<float>(int_val)</code> <code>typeid(float).name()</code> <code>static_cast<char>(a + b)</code> <code>reinterpret_cast<int>(long_ptr)</code> <code>const_cast<A*>(&ra2)</code>
Unary - increment - decrement - sizeof - new - delete	<code>++i</code> or <code>i++</code> <code>--i</code> or <code>i--</code> <code>sizeof i</code> <code>int * int_ptr = new int(7)</code> <code>delete int_ptr</code>
Explicit type conversion(cast notation)	<code>short s = (short) (3.5 + 2.6);</code>
Pointer-to-member operators	<pre>class foo{ public: foo(){i=3;} int i; }; foo f1; int foo::* ip = &foo::i; cout<<f1.*ip<<endl;</pre>
Multiplicative operators	<code>i * j</code> <code>i / j</code> <code>i % j</code>
Additive operators	<code>i + j</code> <code>i - j</code>

Table 5-3: Expression Forms

Expression Form	Examples
Shift operators	<code>i << 2</code> <code>i >> 2</code>
Relational operators	<code>i < j</code> <code>i > j</code> <code>i <= j</code> <code>i >= j</code>
Equality operators	<code>i == j</code> <code>i != j</code>
Bitwise AND operator	<code>i & j</code>
Bitwise exclusive OR operator	<code>i ^ j</code>
Bitwise inclusive OR operator	<code>i j</code>
Logical AND operator	<code>i && j</code>
Logical OR operator	<code>i j</code>
Conditional operator	<code>(i < j) ? i = 3 : i = 7</code>
Assignment operators	<code>i = j</code> <code>i *= j</code> <code>i /= j</code> <code>i %= j</code> <code>i += j</code> <code>i -= j</code> <code>i >>= j</code> <code>i <<= j</code> <code>i &= j</code> <code>i ^= j</code> <code>i = j</code>
Comma operator	<code>int i = 3, j = 2, k = 8</code>
Constant expressions	<code>const int MAX_VALUE = 500;</code> <code>int i = MAX_VALUE;</code>

Table 5-3: Expression Forms

No doubt some of the examples shown in table 5-3 will seem confusing to you if you are new to C++. Have no fear. Your understanding of complex expressions will grow as you progress in your C++ studies. And, as I said earlier, you don't have to learn all the expression types at once.

You will, however, need to know some of the common operators and issues surrounding their use to be productive with C++ right away so I will discuss a few of these below.

OPERATORS

Most of the expressions listed in table 5-3 involve the use of multiplicative, additive, relational, conditional, and assignment operators. You may already have a fundamental understanding of how these operators work, especially the multiplicative, additive, and assignment. I will discuss these and a few others in this section. Any operator I fail to discuss here will be introduced to you later in the book when you are ready to learn its use.

Before I talk about each operator I want to talk briefly about operator precedence.

OPERATOR PRECEDENCE

Operators in C++ have a precedence associated with their use. Table 5-4 lists C++ operators in order of their precedence, from highest to lowest, along with their associativity. The use of parentheses is covered in the next section. Use them and you will have fewer bugs in your code and fewer headaches.

Operator	Description	Associates
++	Post-increment	Left to right
--	Post-decrement	Left to right
()	Function call	Left to right
[]	Array element	Left to right
->	Pointer to structure member	Left to right
.	Structure or union member	Left to right
++	Pre-increment	Right to left
--	Pre-decrement	Right to left
!	Logical NOT	Right to left
~	Bitwise NOT	Right to left
-	Unary minus	Right to left
+	Unary plus	Right to left
&	Address	Right to left
*	Indirection	Right to left
sizeof	Size in bytes	Right to left
new	Allocate program memory	Left to right
delete	Deallocate program memory	Left to right
(type)	Type cast (includes all C++ cast operators)	Left to right
.*	Pointer to member (objects)	Left to right
->*	Pointer to member (pointers)	Left to right
*	Multiply	Left to right
/	Divide	Left to right
%	Modulo or Remainder	Left to right
+	Add	Left to right
-	Subtract	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right

Table 5-4: C++ Operators, Precedence, and Associativity

Operator	Description	Associates
<=	Less than or equal to	Left to right
>	Greater than	Left to right
>=	Greater than or equal to	Left to right
==	Equal to	Left to right
!=	Not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
	Bitwise OR	Left to right
&&	Logical AND	Left to right
	Logical OR	Left to right
?:	Conditional	Right to left
=	Assignment	Right to left
*=, /=, %=, +=, -= <<=, >>=, &=, ^=, =	Compound assignment	Right to Left
,	Comma	Left to right

Table 5-4: C++ Operators, Precedence, and Associativity

USE PARENTHESES

Using table 5-4 as a guide, can you determine what value will be printed to the screen when this statement executes?

```
cout<<7 * 3 + 1 - 201 % 20<<endl; //version 1
```

How about the next statement?

```
cout<<(7 * (3 + 1)) - (201 % 20)<<endl; //version 2
```

And this one?

```
cout<<(((7 * 3) + 1) - 201) % 20<<endl; //version 3
```

As you might guess, each version of the expression results in a different value. Version 1 results in a value derived from performing the computations using each operator's native precedence. Versions 2 and 3 result in different values because the parentheses force a different order of computation.

From a human perspective, versions 2 and 3 are easier to understand. Version 1 takes a little effort unless you are already familiar with the precedence of the operators used. The next version produces the same result as version 1:

```
cout<<(((7 * 3) + 1) - (201 % 20))<<endl; //version 4
```

The parentheses make the expression easier to read and understand. Now you are faced with a dilemma; try and memorize the precedence of every operator, or, use parentheses and simplify your life! Choose wisely grasshopper!

Multiplicative OPERATORS

Table 5-5 lists the three multiplicative operators.

Operator	Description
*	multiplication
/	division
%	modulus

Table 5-5: Multiplicative Operators

The multiplication and division operators are overloaded to work on all the arithmetic types such as float, double, and integer, and enumerations. The modulus operator works on integral type and enumerations only.

Multiplication OPERATOR

The asterisk is used as the multiplication operator. The following code gives an example of its use:

```
int i=0, j=10, k=10000;
i = j*k;
cout<<i<<endl;
```

Be careful when using the multiplication operator. You can easily calculate a value that is too big to fit into a small integer variable. The following code looks like the previous example with one exception. Can you spot the difference?

```
int j=10, k=10000;
short i = j*k;
cout<<i<<endl;
```

The variable named *i* is now declared a short, which holds half as much as a regular integer. This causes a truncation of the larger value to a size that will fit into the smaller data type. These types of errors are easy to make and hard to detect because the compiler offers no warning.

The asterisk, like other symbols in C++, is overloaded to perform more than just multiplication in C++. It is also used to declare and dereference pointers. (*See chapter 8*) As you gain experience reading and writing C++ code you will become comfortable recognizing the context in which operators are used, but, until that happens, you will be a little confused to see what you think is the multiplication operator being used for something other than multiplication.

Division OPERATOR

The division operator works as you would expect although there are a few issues to keep in mind when you use it. The following statement shows the division operator in use:

```
float f = 3.5f / 1.5f;
```

In this example, a float variable named *f* is declared and initialized to the value of 2.33333. In this case, each of the numeric literals are of type float and the size of the result fits into the variable *f*. You will run into trouble when you attempt to store the results of floating point division into an integer variable. The following statement shows an example:

```
int result = 3.5f/1.5f;
```

Integer types are not designed to represent the decimal portion of floating point values so you will lose the .33333 portion of the result. The variable `result` will be initialized with the value 2 and you will not be warned about the loss of numeric precision.

You can perform division on complex expressions by using parentheses. The following statement gives an example:

```
float f = (3.5 + 1.5) / 2.5;
```

You should also be aware that an attempt to divide a number by zero will result in a compiler warning.

Modulus OPERATOR

The modulus or remainder operator works like the division operator but returns the remainder and discards the quotient. The following statement shows the modulus operator in use:

```
int remainder = 215 % 20;
```

This statement declares an integer variable named `remainder` and initializes its value to 15.

The primary thing to remember with the modulus operator is that it is to be used on integral types only. Using it on floating point literals or variables results in a compiler error.

Additive OPERATORS

Table 5.6 lists the additive operators.

Operator	Description
+	Addition
-	Subtraction

Table 5-6: Additive Operators

Addition OPERATOR

The addition operator performs arithmetic addition on arithmetic, enumeration, or pointer types. The following code shows an example of the addition operator being used with an enumeration type:

```
enum set_one {up, down, left, right};
...
int where = 1 + down;
```

In this example, an enumerated type named `set_one` is declared with the four enumerations `up`, `down`, `left`, `right`. The value of `up` is 0, the value of `down` 1, the value of `left` 2, and the value of `right` 3. (*Enums are covered in more detail in chapter 10*) The integer variable `where` is declared and initialized to the value of the integer literal 1 plus the enumeration value of `down`.

The following code gives an example of the addition operator being used with a pointer operand:

```
int int_array[] = {1, 2, 3, 4, 5};
int int_val = *(int_array + 3);
```

In this example an array of integers named `int_array` is declared and initialized with five integer values. On the next line an integer variable named `int_val` is declared and initialized to the value that resides at the 4th element of the array. In this case the value stored in the 4th element of the array is 4. Arrays are covered in excruciating detail in chapter 8 but here's a quick explanation of what's going on here. The array name `int_array` points to the start of the

integer array. This means the name of the array is a pointer, which means it contains a memory address. Since the addition operator is being applied to a memory address that points to an array of integer elements, the value 3 means 3 integer storage units. The addition will be the memory address of the array + (3 times the size of an integer in bytes) or *array address* + (3 x 4) or *array address* + 12.

The value resulting from the addition is a memory address or pointer. The overloaded asterisk symbol, in this case used as the pointer dereferencing operator, must be applied to the result to obtain the actual integer object residing in the 4th element of the array. It is this value that is ultimately assigned to the integer variable named `int_val`.

Don't worry if you don't fully understand all this pointer stuff right now. After you read chapters 7 and 8 you will be an expert!

Subtraction Operator

Besides being used for traditional arithmetic subtraction operations on arithmetic data types, the subtraction operator can be used on pointers as well. The following code demonstrates the use of the subtraction operator on pointer types:

```
int_val = *(&int_array[4] - 3);
```

Using the integer array from the previous example, the subtraction operator is being used to subtract 3 integer storage units from the address of the 5th element of `int_array`. This will result in the value 12 being subtracted from the address of the 5th array element which will yield the address of the 2nd element. The result of the subtraction is a memory address and must be dereferenced to access the integer object stored at that address. When this statement is executed the integer value 2 will be assigned to the variable named `int_val`.

Shift Operators

Shift operators let you perform bit shifting operations on integral objects. Table 5-7 lists the shift operators.

Operator	Description
<<	Left Shift
>>	Right Shift

Table 5-7: Shift Operators

Left Shift Operator

The following code shows the left shift operator in use.

```
unsigned shift_val = 1;
shift_val = shift_val << 1;
```

The first statement declares an unsigned integer variable named `shift_val` and initializes its value to 1. In a computer with 32 bit registers the value 1 looks like this in binary:

```
00000000000000000000000000000001
```

The second statement shifts the bits of `shift_val` to the left by one bit and assigns the result of the shift operation back to the `shift_val` variable. As the bits are shifted to the left, the right-hand replacement bits are set to zero. Figure 5-8 shows what happens to `shift_val` when its bits are shifted to the left four times.

The effect of left shifting a bit value by one bit is the same as multiplying the value by two. But you need to be careful and not left shift it too far. You also have to pay attention to the type of bit pattern you are shifting. For instance, left shifting a signed type will result in values going from positive to negative depending on the value of the bit that moves into the sign bit position.

```

      shift_val starts with the value 1
00000000000000000000000000000001 ←———— shift_val == 1
      shift_val = shift_val << 1;
00000000000000000000000000000010 ←———— shift_val == 2
      shift_val = shift_val << 1;
00000000000000000000000000000100 ←———— shift_val == 4
      shift_val = shift_val << 1;
0000000000000000000000000001000 ←———— shift_val == 8
      shift_val = shift_val << 1;
000000000000000000000000010000 ←———— shift_val == 16

```

Figure 5-8: Left Shifting `shift_val`***Right Shift OPERATOR***

The right shift operator works similar to the left shift operator. If the value being shifted is an unsigned type or a signed type with a positive value then the effect of shifting the bits to the right by one bit will be the same as dividing the value by 2. If the value being shifted is a negative number then the result of shifting right is implementation dependent, meaning the operator's behavior in this regard is left to the discretion of the compiler manufacturer. The following code shows the right shift operator in use on a negative number:

```

int shift_val = 0xFFFFFFFF;
shift_val = shift_val >> 1;

```

The first statement declares the integer variable `shift_val` and initializes it to the hexadecimal value `FFFFFFFF`. This is the bit pattern for `-1`. When statement two is executed the bits are shifted to the right by one bit. On Metrowerks CodeWarrior the value remains `-1`. This is due to the bits coming in from the left being set to 1, which keeps the sign bit set.

It will be helpful to see another example. The following code initializes `shift_val` to the maximum negative number an integer can hold then shifts it to the right by one bit.

```

int shift_val = 0x80000000;
shift_val = shift_val >> 1;

```

In the first statement, `shift_val` is initialized to the value `-2,147,483,648`. After the execution of the second statement its value will be `-1,073,741,824`. Shifting the negative number the right by one bit has the effect of dividing the number by 2. However, this will only work until you have shifted all the way to the right, at which time the value will be `-1`. Further right shifting will have no effect. Figure 5-9 shows what happens to the variable `shift_val` when the right shift operator is applied 4 times.

You can see in figure 5-9, with a negative number, the bits shifted into the bit pattern in the most significant bit position are set to 1. This keeps the value negative.

If you start with a positive number in a signed integer type the number will remain positive as you shift bits to the right. This means the bit being shifted into the most significant bit position is set to 0.

```

shift_val starts with the value -2,147,483,648
10000000000000000000000000000000 ← shift_val == -2,147,483,648
    shift_val = shift_val >> 1;
11000000000000000000000000000000 ← shift_val == -1,073,741,824
    shift_val = shift_val >> 1;
11100000000000000000000000000000 ← shift_val == -536,870,912
    shift_val = shift_val >> 1;
11110000000000000000000000000000 ← shift_val == -268,435,456
    shift_val = shift_val >> 1;
11111000000000000000000000000000 ← shift_val == -134,217,728

```

Figure 5-9: Right Shifting shift_val

RELATIONAL OPERATORS

Table 5-8 lists the relational operators.

Operator	Description
<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To

Table 5-8: Relational Operators

Relational operators are used to compare the value of arithmetic, enumeration, or pointer objects. A relational operator returns a boolean value which is either true or false.

LESS THAN OPERATOR

The less than operator takes two operands and returns true if the left operand is less than the right operand. The following statement illustrates the use of the less than operator:

```
bool result = 3 < 5;
```

When this statement is executed the boolean variable named result will be initialized to the value true or 1.

GREATER THAN OPERATOR

The greater than operator takes two operands and returns true if the left operand is greater than the right operand. The following statement illustrates the use of the greater than operator:

```
bool result = 3 > 5;
```

In this case, the variable result will be initialized to false or 0.

LESS THAN OR EQUAL TO OPERATOR

The less than or equal to operator takes two operands and returns true if the left operand is of lesser value or equal to the right operand. The following statement illustrates the use of the less than or equal to operator:

```
bool result = 3 <= 5;
```

When this statement is executed the boolean variable result will be initialized to true or 1.

GREATER THAN OR EQUAL TO OPERATOR

The greater than or equal to operator takes two operands and returns true if the left operand is of greater value or equal to the right operand. The following statement illustrates the use of the greater than or equal to operator:

```
bool result = 3 >= 5;
```

When this statement executes result will be initialized to the value false or 0.

You will use relational operators heavily to make decisions in your source code in order to figure out what to do next. You will see more about relational operators covered in chapter 6 when I discuss how to control the flow of program execution.

EQUALITY OPERATORS

Table 5-9 lists the equality operators. Equality operators can compare arithmetic, enumeration, and pointer values just like the relational operators but have a lower precedence.

Operator	Description
==	Equal To
!=	Not Equal To

Table 5-9: Equality Operators

EQUAL TO OPERATOR

The equal to operator compares two operands for equality and returns a boolean value of true or false based on the result of the comparison. The following statement shows the equality operator in action:

```
bool result = 3 == 5;
```

This statement will initialize the boolean variable result to false.

NOT EQUAL TO OPERATOR

The not equal to operator compares two operands and returns true if they are not equal. The following code illustrates the use of the not equal to operator:

```
bool result = 3 != 5
```

This statement will initialize the variable result to true or 1.

BITWISE AND OPERATOR - &

The bitwise AND operator takes two operands, ANDs them, and returns the result. What the heck is an AND operation you ask?

An AND operation compares two bits. The result of the bit comparison is either 1 or 0 depending on the state of the two bits compared. An AND on two bits will result in a true or 1 output only when both bits being compared are true or 1. Figure 5-10 gives the truth table for an AND operation. The following statement shows the bitwise AND operator in use:

```
int and_result = 0x00000001 & 0xFFFFFFFF;
```

In this example, the variable `and_result` will be initialized to the result of the AND operation which, in this case, is 1. Take a look at this operation again in slow motion:

```
00000000000000000000000000000001
11111111111111111111111111111111
00000000000000000000000000000001
```

When the hexadecimal is converted to binary you can easily see the only bit comparison that results in a true is the two least significant bits. The rest of the bits are set to false or 0 because only one bit in each comparison is on or true. I use the terms on, true, and 1 interchangeably here as well as off, false, or 0.

Bitwise Exclusive OR OPERATOR - ^

Bitwise exclusive OR operations take place according to the truth table shown in figure 5-11. The following statement illustrates the use of the exclusive OR operator in action:

```
int ex_or_result = 0x00000001 ^ 0xFFFFFFFF;
```

When this statement is executed the variable `ex_or_result` will be initialized to `0xFFFFFFFFE`. Take a look at the binary version:

```
00000000000000000000000000000001
11111111111111111111111111111111
11111111111111111111111111111110
```

Bitwise Inclusive OR OPERATOR - |

The bitwise inclusive OR takes two operands and returns the result of an inclusive OR comparison between each bit. Figure 5-12 gives the truth table for an inclusive OR operation. The following statement show the bitwise inclusive OR operator in action:

```
int in_or_result = 0x00000001 | 0xFFFFFFFF;
```

The variable `in_or_result` will be set to the value `0xFFFFFFFF`. Examine the binary version:

```
00000000000000000000000000000001
11111111111111111111111111111111
11111111111111111111111111111111
```

In this case all bits are set to 1 as the truth table for the inclusive OR operation would suggest.

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5-10: AND Truth Table

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5-11: Exclusive OR Truth Table

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Figure 5-12: Inclusive OR Truth Table

LOGICAL AND OPERATOR - &&

The logical AND operator takes two boolean expressions as operands and returns true or false based on the truth table given in figure 5-10. This is different from its bitwise counterpart in that it is not comparing bits, rather, it is comparing the result of one expression to that of another, and returning a result base on the result of the comparison. The following code gives an example of the logical AND operator in use:

```
int a = 0;
boolean and_result = (a<5) && (true);
```

On the first line an integer variable named `a` is declared and initialized to 0. On the second line the boolean variable `and_result` is declared and initialized to the result of the logical AND expression. The logical AND operator will compare the results of `(a<5)`, which is true, to the `(true)`, which is always true.

LOGICAL OR OPERATOR - ||

The logical OR operator makes comparisons of two boolean expressions according to the truth table shown in Figure 5-12. The following code gives an example of the logical OR in use:

```
int a = 0;
boolean or_result = (a<5) || (true);
```

In this example, the boolean variable `or_result` is initialized to the result of the OR comparison between `(3<5)` which is true, and `(true)`, which is still true!

CONDITIONAL OPERATOR - ? :

This is a cool operator but until you get used to it you will look a little cross-eyed at it when you see it in source code. To discuss the conditional operator I have to get ahead of my story a little bit. The conditional operator is a shorthand way of writing an if statement. If statements are covered in detail in chapter 6.

The conditional operator will evaluate a boolean expression and offer two possible alternatives, depending on the result of the evaluation. The expression to be evaluated comes before the question mark. The first alternative, or statement you want to execute if the expression is true is placed to the right of the question mark and to the left of the colon. The statement you want to execute if the expression evaluates to false is placed to the right of the colon. Use this handy map to the conditional operator if you get lost while trying to impress your friends by using it:



Figure 5-13: Conditional Operator Map

The following statement shows the conditional operator in use:

```
(3>5) ? cout<<"True statement"<<endl : cout<<"False statement"<<endl;
```

When this statement is executed the expression `(3>5)` will be evaluated and result in a boolean value of false. This will cause the false statement to be executed. In this case the text `False statement` will be printed to the screen. The following if statement will do the same thing:

```
if (3>5)
    cout<<"True statement"<<endl;
```



```
else cout<<"False statement"<<end;
```

ASSIGNMENT OPERATORS

Table 5-10 lists the assignment operators.

Operator	Description
=	Assignment
*=	Compound Multiplication Assignment
/=	Compound Division Assignment
%=	Compound Modulus Assignment
+=	Compound Addition Assignment
-=	Compound Subtraction Assignment
>>=	Compound Right Shift Assignment
<<=	Compound Left Shift Assignment
&=	Compound Bitwise AND Assignment
^=	Compound Bitwise Exclusive OR Assignment
=	Compound Inclusive OR Assignment

Table 5-10: Assignment Operators

You have seen the assignment operator, =, in action many times in this chapter. The important thing to remember when using the assignment operator is that it is not the Equal To operator, ==, which is an equality operator vice an assignment operator. To better understand the use of the assignment operator it is helpful to know the difference between an “lvalue” and an “rvalue”.

lvalue vs. rvalue

If you are new to C++ programming you have probably seen the compiler error, “Not an lvalue...”, and wondered what it meant.

When making an assignment using an assignment operator, what you are trying to do is assign the result, or value, of some expression to a memory location. The assignment expression, when viewed in this context, looks like this:

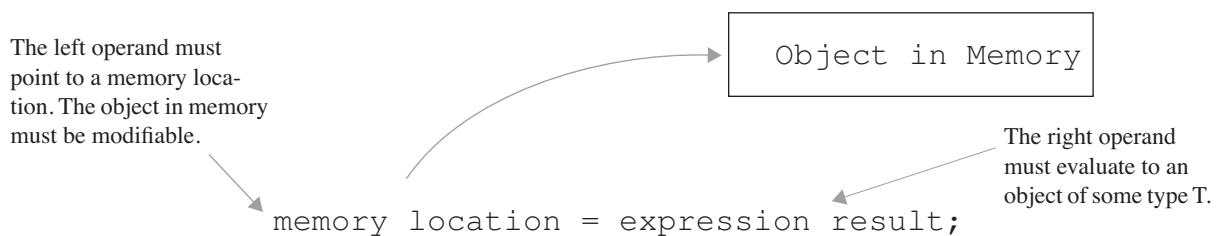


Figure 5-14: Assignment Operator Operands

An assignment is an expression and therefore returns a result. The type of the result is the type that was stored in the memory location pointed to by the left operand. The following example might help clarify this concept:

```
int a = 0, b = 0;
a = (b = 5);
```

In this example, two integer variables named `a` and `b` were declared and initialized to 0 using the assignment operator. The second statement assigns to the variable `b` the value 5. The result of the assignment is 5 and is assigned to the variable `a`.

Compound Assignment Operators

The rest of the assignment operators are known as compound assignment operators and are used as a shorthand way of getting things done in C++. The following code shows the compound multiplication assignment operator in action:

```
int a = 3;
a *= 3;
```

The first statement declares the integer variable named `a` and assigns it the value 3. The next statement multiplies `a` by 3 and assigns the result back to the variable `a`. The second statement is equivalent to the following longer version that does the same thing:

```
a = a * 3;
```

The rest of the compound operators work the same way.

Comma Operator - ,

The comma operator can be used to separate expressions. The following code shows the comma operator in use:

```
int a = 0, b = 0, c = 0;
a = (c = 10), (b = 8);
(3 < 5), (a = 0);
```

The first statement shows the comma operator being used to separate variable declarations and assignments on the same line. The second statement shows two assignment expressions being separated by the comma operator. The third statement is something not often seen but reiterates the use of the comma operator's ability to separate any expression, not just assignment expressions.

Increment and Decrement Operators (++, --)

There are many times during the course of programming that you need to increment or decrement a variable by 1. You could increment the old fashioned way...

```
i = i + 1; or i += 1;
```

...in this example the variable `i` is being set to the value it contains plus 1, or use the `++` operator:

```
i++; or ++i;
```

There are two versions of the increment and decrement operators: prefix and postfix. Consider the following example:

```
int a = 0, i = 1;
```

```
a = i++;
```

This is an example of the postfix version of the increment operator in use. When the second statement executes `a` will be assigned the value of 1, which is the value of `i` before the increment operator expression is evaluated. The following example demonstrates the use of the prefix version of the increment operator and will result in the variable `a` being assigned the value 2:

```
int a = 0, i = 1;
a = ++i;
```

The decrement operator works the same way. You will most likely see the increment and decrement operators used in for statements to do array processing. The following code gives an example:

```
for(int i=0; i<ARRAY_SIZE, i++){
    //array processing statements here
}
```

Chapter 6 discusses program control flow statements like the for statement, above, in detail.

IDENTIFIERS

Identifiers are names given to various objects in a program. You have already seen several identifiers declared and used in this chapter; they were given short names like `i`, `a`, `b`, and represented simple data types. Identifiers are also used to name functions, labels, and other user defined data types. The C++ reserved keywords, listed in the Keywords section of this chapter, are examples of identifiers you cannot use to name your objects because they are reserved by the compiler.

Identifiers are formed using letters and digits, however, an identifier must start with a letter or underscore “_” character. The following are valid identifiers:

```
count
_count
I_count_to_10
_9to5
getCount ()
printScreen ()
Label1
```

IDENTIFIER NAMING CONVENTIONS

In chapter 1 I discussed the importance of adopting a naming convention and sticking with it. I also proposed a simple naming convention you could use in your programs. However, other naming conventions exist.

HUNGARIAN NOTATION

Some naming conventions are famous, such as Hungarian notation, formulated by Dr. Charles Simonyi of Microsoft. The concept of Hungarian notation goes something like this: Variable names are prefixed with an abbreviation indicating the variable’s type or class. Table 5-11 lists possible type prefixes.

Prefix	Description
c	signed character

Table 5-11: Possible Hungarian Notation Prefixes

Prefix	Description
uc	unsigned character
i	integer
ui	unsigned integer
si	short integer
li	long integer
n	an integer number where the actual size is irrelevant
f	float
d	double
s	string of characters
sz	string of characters, terminated by a null character
b	an integer or character being used as a boolean value
by	single byte
ct	an integer being used as a counter or tally
p	pointer to a structure or general void pointer
pfs	file stream pointer
pfn	pointer to a function
px	pointer to a variable of class x, e.g. pc, pf, pSubmarine
v	void type

Table 5-11: Possible Hungarian Notation Prefixes

To derive identifiers combine the prefixes with names that describe the use of the identifier. Capitalize the first letter of each word in the identifier name. Here are a few examples:

`pf_grade_average` — *pointer to float type*
`li_number_of_cars` — *long integer type*

You could also apply this naming scheme to functions. The purpose of prefixing the function name would be to indicate what type, if any, the function returned. Here's a couple examples:

`vprintScreen()` — *returns nothing*
`igetCount()` — *returns integer*

Again, the important thing about naming conventions is not necessarily which one you choose, but that you use it consistently.

CONSTANTS

A constant is an object in your program that you intend to remain unchanged during its lifetime. The use of constants can vastly improve the readability and maintainability of your code.

You can declare constants in C++ using the keyword *const*. Constants have to be defined at the point of declaration. The following statement gives an example of a constant declaration and definition:

```
const int MAX_COUNT = 25;
```

There is an exception to the define-at-the-point-of-declaration rule, and that is when you are declaring constants for use in classes or structures. In chapter 11 I will show you how to use constructors to define class constant members.

Once you have defined the constant, any attempt to change its value will be met with disapproval from the compiler.

VARIABLES

A variable is an object whose value will likely change during the execution of a program. Variables have a storage class, and a visibility or scope

DECLARING

You have seen several variables declared and used in this chapter. You precede the identifier with the variable's type as in the following example:

```
char* f_name;
```

In this case the type of the variable named `f_name` is pointer to char. Here's another example:

```
float account_balance;
```

Variables can be defined or initialized at the point of declaration or later if necessary. As a rule, though, it is a good idea to initialize the variable to some known value so you don't try to use a variable that contains a garbage value. Here's an example:

```
long defunct_dot_coms = 0;  
...  
defunct_dot_coms = 32345;
```

Here the variable named `defunct_dot_coms` is declared and initialized to 0. You can also initialize a variable using "constructor" notation. Check this out:

```
int cool_variable(40);
```

SCOPE

Variables have a scope or authorized area of usage within a program. Sorting out all the scoping rules can be overwhelming and they're best mastered on-the-fly so I will only discuss two basic scoping rules here and leave the others to the chapter to which they more aptly belong.

Scoping is also related to the topic of linkage, so, to understand each completely, you must understand both as they relate to each other. The best way to learn them both is by studying examples so I will go light on the verse and heavy on the code.

Local Scope

A variable is available for use after its point of declaration. Let us look at an example:

```
int a;
a = 3;
cout<<a<<endl;
```

The variable `a` is available for use just after the its declaration on the first line, just before the semicolon indicating the end of the first statement. From that point forward `a` is in scope and can be used as shown. The braces `{ }` can be used to introduce local blocks of scope into a sequence of statements as shown in the following example:

```
1  int a = 1, b = 2;
2  {
3      cout<<a<<endl;
4      int a = 3;
5      cout<<a<<endl;
6  }
7  cout<<a<<" "<<b<<endl;
```

In this example, two integer variables, `a`, and `b`, are declared and initialized on line 1. On the line 2 a new scope is introduced with the left brace and proceeds up to the closing right brace on line 6. Within this scope a new integer variable named `a` is declared and initialized on line 4. Its scope is up to the closing right brace. Figure 5-15 gives a graphical view of what's happening:

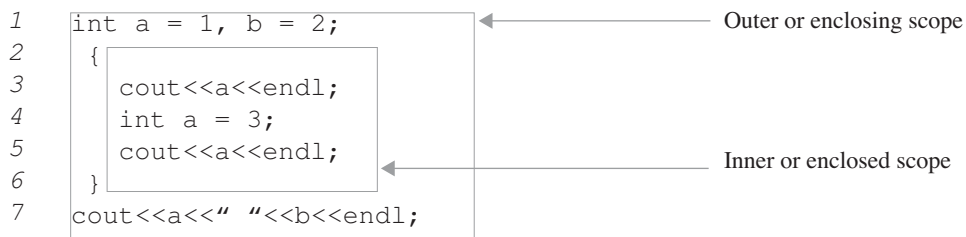


Figure 5-15: Creating Local Scope Blocks with Braces

When a local scope is created within another scope, the inner scope is said to be the enclosed scope, while the outer scope is referred to as the enclosing scope. If you declare a variable in an enclosed scope using the name of a variable already in scope in the outer scope it causes the outer scope variable to be hidden or masked from the point of declaration in the enclosed scope.

Looking at figure 5-15, the variable `a` declared in the outer scope is visible up to where the second variable `a` is declared in the inner scope. The statement on line 3 will print the outer scope `a` value to the screen. The statement on line 5 will print the inner scope's `a` value to the screen, and finally, the last statement will print the values of both outer scope variables `a` and `b`.

FUNCTION SCOPE

Functions will be discussed in detail in chapter 9 so I will keep the examples here simple. Variables declared within a function have scope within that function. Examine the following code:

```

1  int a = 1, b = 2;
2
3  void someFunction(){
4      cout<<"In someFunction(): outer scope a == "<<a<<endl;
5      int a = 3;
6      cout<<"In someFunction(): inner scope a == "<<a<<endl;
7  }
```

In this example, the integer variable `a` declared on line 1 exists in the outer scope and is visible within `someFunction()` up to the point of declaration of `someFunction()`'s local variable `a`. Example 5.5 illustrates the scoping issues discussed thus far:

5.5 Source Code Showing Local, Function, and File Scoping

```

1  #include <iostream>
2
3  using namespace std; //introduces namespace std
4
5  void someFunction();
6
7  int a = 1, b = 2;
8
9  void someFunction(){
10     cout<<"In someFunction(): outer scope a == "<<a<<endl;
11     int a = 3;
12     cout<<"In someFunction(): inner scope a == "<<a<<endl;
13 }
14
15 int main(){
16     someFunction();
17     {
18         cout<<a<<endl;
19         int a = 3;
20         cout<<a<<endl;
21     }
22     cout<<a<<" "<<b<<endl;
23     return 0;
24 }
```

Referring to example 5.5, line 5 gives the function declaration of `someFunction()`. On line 7 appears the declaration and definition of integer variables `a` and `b`. Since these two variables are declared outside of any function they have file scope. Variables with file scope are often referred to as global variables because they are visible to all functions defined within the file in which they appear and, as you will see below, to functions appearing in other files as well.

File Scope

As mentioned above, variables declared outside of any function within a file have file scope. These are referred to as global variables because they can be seen globally throughout the program.

Keep the use of global variables to the absolute minimum required to reduce coupling in your program. Remember from chapter 1 that intermodule coupling is a bad thing! Minimize coupling — maximize cohesion.

Multifile Variable Usage

If you must use global variables and are programming multifile projects you will need to know how to control your variables across many files. In some cases you will want to make your global variables visible to all the files in your program; in others, you will want to limit the visibility of file scope variables to one file. Here's how you do it...

SHARING FILE SCOPE VARIABLES ACROSS MULTIPLE FILES

File scope variables by default have external linkage meaning they are visible across all the files in your project. Examine the two source files shown in example 5.6:

5.6 file scope linkage

```
#include <iostream>
using namespace std;
int a = 1;
void someFunction();
void someFunction() {
    cout<<a<<endl;
}
```

file1.cpp

```
#include <iostream>
using namespace std;
void someFunction();
extern int a;
int main() {
    a = 3;
    someFunction();
    cout<<a<<endl;
    return 0;
}
```

file2.cpp

file1.cpp and file2.cpp represent two files belonging to the same project. file1.cpp contains the declaration and definition of someFunction() along with a global variable a which is assigned the value of 1. file2.cpp declares someFunction() again before calling it in the main() function. file2.cpp also declares the variable a but prefaces it with the keyword *extern* to tell the linker that the variable a has been declared and defined in another file and that you intend to use that variable in this file too.

When a is assigned the value 3 in the body of the main function, that's the value printed to the screen when someFunction() is called.

If the keyword *extern* were to be left off the declaration of a in file2.cpp the following link error would occur when you tried to compile and run the program:

```
Link Error      : multiply-defined 'a' (data)
Defined in file1.cpp
Defined in file2.cpp
```

LIMITING FILE SCOPE VARIABLE VISIBILITY TO ONE FILE

Example 5.7 shows the same two files after being modified slightly: The keyword *static* now prefaces the declaration of a in file1.cpp, limiting its visibility to file1.cpp. With file1.cpp's variable a safely limited to file scope file2.cpp can declare its own variable a and use it as it sees fit. The results of calling someFunction() and executing the cout statement now produce different results, namely, the values 1 and 3 are printed to the screen.


```
#include <iostream>

using namespace std;

static int a = 1;

void someFunction();

void someFunction(){
    cout<<a<<endl;
}
```

file1.cpp

```
#include <iostream>

using namespace std;

void someFunction();

int a;

int main(){
    a = 3;
    someFunction();
    cout<<a<<endl;
    return 0;
}
```

file2.cpp

THE MAIN() FUNCTION

Every C++ program contains a main() function. You may write library routines that do not contain a main function but the program that ultimately uses them will have a main() function. This section will briefly discuss the purpose of the main() function and the two forms it can take.

THE PURPOSE OF THE MAIN() FUNCTION

The main() function marks the start of program execution. A program may be small and contain only a main() function that itself only contains a few simple statements, or it may be huge and be comprised of many different objects and stand-alone functions that are defined across thousands of files and millions of lines of source code. What ever form it takes, it all starts with main().

TWO FORMS OF MAIN()

The main() function can be written two ways. You have seen the first, and most simple, form of writing main() in this chapter already:

```
int main(){
    // some statements here
    return 0;
}
```

The second way to write main() looks like this:

```
int main(int argc, char* argv[]){
    //process command line input
    //along with other statements here
    return 0;
}
```

This form of `main()` is used to write programs that process command line arguments. If you have used DOS or UNIX you have probably used utility commands that required command line arguments to run properly.

Exiting `main()`

There are several ways to exit the `main()` function. You seen one way used throughout this chapter and that is by using the keyword `return` followed by a `0`. This is actually the lazy way of doing things, mainly because having the value `0` in a statement gives no clue to what `0` means.

It just so happens that the value `0`, when returned from `main()`, means the program executed successfully. In fact there is a constant already declared for your use called `EXIT_SUCCESS`. You will find it and some other cool stuff in the `cstdlib` header file. (`stdlib.h` on older C or C++ environments) Here is an example of using `EXIT_SUCCESS`:

```
int main() {
    // amazingly clever code goes here...
    return EXIT_SUCCESS;
}
```

Instead of using the `return` keyword you can use the function `exit(int status)`:

```
int main() {
    //code gone bad
    exit(EXIT_FAILURE);
}
```

In chapter 7 you will see other ways to use the `exit()` function and what effects it has on objects in your program.

Calling Functions Upon Exiting `main()`

Use the `atexit(void (*func)(void))` to register a function you want called when the `exit()` function is called. Example 5-8 gives the code:

```
#include <iostream>
#include <cstdlib>

using namespace std;

void goodBye();

void goodBye() {
    cout<<"Goodbye cruel world!"<<endl;
}

int main() {
    atexit(&goodBye);
    exit(EXIT_SUCCESS);
}
```

5-8 Registering Functions with `atexit()`

Register the `goodBye()` function by calling `atexit()` with the `goodBye()` function's address. Use only the name `goodBye` leaving off the parenthesis.

Simple Input and Output

C++ provides character stream input and output in the form of the `cin` and `cout` objects. You have already seen the stream insertion operator in use to send variable values and strings to the `cout` object. My objective in this book is to

limit the use of `cin` and `cout` to the absolute minimum required to facilitate program interaction. I do this because there exist several good `iostream` books on the market that go into much greater detail than I wish to do so in this book.

My second reason for limiting my coverage is that should you decide to specialize in a particular operating system you will need to learn how to use the graphical user interface components like windows, text boxes, text fields, etc., supported by that operating system.

`cin`

The `iostream` header file declares the `cin` object for your console stream input use. The `cin` object takes input from the standard input device, which is usually the keyboard, and directs it to designated variables. Example 5.9 shows the `cin` object being used to read in several integer values and direct the input to the integer variables `a`, `b`, and `c`:

```

1  #include <iostream>
2
3  using namespace std; //introduces namespace std
4
5  int main(){
6      int a = 0, b = 0, c = 0;
7
8      cout<<"Enter values for a, b, and c: ";
9      cin>>a>>b>>c;
10     cout<<a<<" "<<b<<" "<<c<<endl;
11
12     return 0;
13 }
```

5.9 Using cin Object to Read Integer Values from Keyboard

Notice the way the `>>`'s point when using the `cin` object vs. the `cout` object. The `>>` is called the stream extraction operator and the `<<` is called the stream insertion operator.

TRAPPING BAD INPUT

A problem arises when using the `cin` object to read input from the keyboard. If you are expecting a certain type, say, an integer, but the user enters a character or string of characters, it causes the extraction operation to fail and `cin`'s internal fail bit will be set. What the user sees when this happens is a screen scrolling wildly out of control; the program just crashed.

A simple solution, until you learn more robust error trapping techniques, is to test for the success of the `cin` stream extraction. Example 5.10 offers an example. Example 5.10 shows the `cin` object being tested for success or

5.10 Testing for Valid Input

```

1  int d;
2
3  cout<<"Enter an integer value: "<<flush;
4  cin>>d; ←————— Should extract an integer value
5
6  while(!cin){ ←————— Test for cin failure
7      cout<<"Input was bad! Try Again..."<<endl;
8      cin.clear(); ←————— Reset cin object to good state
9      cin.ignore(INT_MAX, '\n'); ←————— Get rid of garbage characters
10     cout<<"Enter an integer value: "<<flush;
11     cin>>d; ←————— Try it again...
12 }
```

failure in the expression section of a while loop. If the extraction operation was successful, which in most cases means the correct type was extracted from the input stream and stored in the designated variable, then the while loop is skipped. If the extraction operation failed then the body of the while loop is entered and will loop until the extraction operation is a success.

COUT

The cout object sends stream output to the standard output device, usually the screen. You have seen the cout object in action throughout this chapter. As noted above “<<” is called the stream insertion operator. It is overloaded to properly handle all the native C++ types.

The most common mistake made when using the cout object is forgetting to put quotation marks around strings, but practice makes perfect.

LEARNING MORE ABOUT COUT AND CIN

My favorite book on C++ iostreams is the *C++ IOSTreams Handbook* by Steve Teale, Addison Wesley, ISBN 0-201-59641-5.

SUMMARY

All C++ programs require a main() function and a minimum, well-formed C++ program may have nothing but a main() function, although it wouldn't be very useful. You can learn a lot about the language and the host computer by disassembling programs. The Metrowerks CodeWarrior disassembler can be configured to display output in several different formats.

Keywords are identifiers reserved for use by the compiler. You build C++ programs from declarations, statements, and expressions constructed from keywords, identifiers, fundamental data types, and operators.

C++ contains several fundamental data types such as char, int, float, and double, etc. Fundamental data types can represent a certain range of values depending on their size. Learn the size, in bytes, of data types by using the sizeof operator. Determine the range of data types by using the numeric_limits template class.

Numbers, characters, and strings of characters that appear directly in programs are known as literals. Integer literals can be expressed in decimal, octal, or hexadecimal format. Multiple character literals have a value equal to the integer value of their bit representation. Special characters can be represented by escape sequences. Floating point literals have type double unless specified as float with the f or F suffix. String literals are terminated with the null character '\0'. Boolean literals true and false represent the values 1 and 0.

Expressions, constructed from operators, specify a computation from which a value may result. Operators have precedence; use parenthesis to explicitly define operator precedence in an expression and make code easier to read and understand.

Identifiers are names given to objects in a program; adopt a naming convention and stick with it. A constant is an object whose value will remain unchanged during its lifetime. A constant must be defined at the point of declaration except in class member constants which require initialization in the class constructor. (*see chapter 11*)

A variable's value will change during its lifetime. Variables have an area of authorized usage within a program known as scope. Redeclaring a variable in an enclosed scope can mask or hide a variable of the same name in outer or enclosing scopes. File scope variables have external linkage by default. Use the static keyword to limit file scope variable visibility to the file in which it is declared.

The main() function takes two forms. The cstdlib header defines the constants EXIT_SUCCESS and EXIT_FAILURE for use in the return statement or exit() function. Use atexit() function to call a function of your choosing upon exiting a program.

Stream input and output is provided by the cin and cout objects.

Skill Building Exercises

- Disassembly:** Write a short program that adds two integer literals together and assigns the result to an integer variable. Disassemble the program and study the resulting output.
- Fundamental Type Sizes:** Write a program demonstrating the use of the sizeof operator and calculate the sizes of all the fundamental data types for your development environment.
- Fundamental Type Value Ranges:** Write a program demonstrating the use of the numeric_limits template class and calculate the range of values of all the fundamental data types for your development environment.
- Hex To Binary Conversion:** Convert the following hexadecimal numbers to binary:
 - 0xD3F45C88
 - 0x864EE701
 - 0xAAFFAAFF
 - 0x37808978
 - 0xA1E5A1E5
- Decimal to Binary Conversion:** Convert the following decimal numbers to binary. Indicate the smallest C++ data type necessary to represent the value:
 - 34 (ASCII)
 - 246
 - 32,746
 - 124,256
 - 4,294,967,295
- Multiple Character Literals:** What integer value does the character literal ‘Stop’ represent?
- Operator Usage:** Write a program that demonstrates the use of each of the following operators: postfix ++ and prefix ++, postfix and prefix --, *, /, %, +, -, <<, >>, <, >, <=, >=, ==, !=, &, ^, |, &&, ||, !, ?:, *=, /=, %=, +=, -=, >>=, <<=, &=, ^=, |=.
- Exiting main():** Write three short programs demonstrating how to exit the main() function using 1) a return statement, 2) the exit() function, and 3) the exit() function with a clean-up function registered with the atexit() function.
- Variable Scoping and Linkage:** Write a short, two-file program demonstrating the following aspects of variable scoping and linkage and answer the associated questions:
 - Declare an integer variable named shared_global in one file and redeclare it with the extern keyword and use it in the other file. What effect does using the variable in each file have on the value of the variable?
 - Declare an integer variable named file_global in both files using the static keyword. What effect does using this variable in each file have on the value of the variable?
 - In the body of the main() function declare and initialize an integer variable named var1. Print its value to the screen using the cout object.
 - Below the cout statement declare a local block using the braces { }, and print the variable again using the cout object. Did its value change? Explain why or why not.
 - In the local block, below the cout statement, redeclare and initialize to a different value from the original, the variable var1. Print var1 to the screen using the cout object. Did its value change? Explain the results.
 - Is there a way to access the original var1 variable from within the local block?
- IOStream Input and Output:** Write a short program demonstrating simple iostream input and output using the cin and cout objects. Declare a few variables of various types and set their values via the keyboard using the cin object. Use the technique shown in this chapter to catch bad input. Print the contents of each variable to the screen

using the cout object.

SUGGESTED PROJECTS

1. **Calculate Averages:** Write a program that calculates the average age of the members of your family. If you have a large family limit the number of ages you use to ten. Use the cout object to prompt for the entry of each age. Use the cin object to read the input from the keyboard and store it in a variable. Use the technique shown in the chapter to trap bad input.
2. **Calculate Area:** Write a program that calculates the area of a square given the length of one side. Use the cout object to prompt for input and the cin object to read the input from the keyboard and assign it to a variable for computation.
3. **Calculate Area:** Write a program that calculates the area of a rectangle given its length and width. Use the cout object to prompt for input and the cin object to read the input from the keyboard. Use two variables of the appropriate type to store the values of length and width. Print the values of the rectangle's length and width, and its area, using cout.
4. **Experiment:** Write a program that prints out all the escape sequences listed in table 5-2. You may want to add characters to the output string when printing the horizontal tab and vertical tab so you can gauge the effect.
5. **Bitwise AND:** Write a program that takes two hexadecimal numbers as input, performs a bitwise AND on them, and displays the result.
6. **Bitwise Exclusive OR:** Write a program that takes two hexadecimal numbers as input, performs a bitwise exclusive OR on them, and displays the result.
7. **Bitwise Inclusive OR:** Write a program that takes two hexadecimal numbers as input, performs a bitwise inclusive OR on them, and displays the result.
8. **Comparison Operators:** Write a program that takes two decimal integers as input, compares their values with the <, >, <=, and >= operators, and displays the results.
9. **Calculate Formula:** Write a program that performs the following computation:

$$x = \frac{(a \cdot b)/c}{a^2}$$

Enter the values for a, b, and c from the keyboard. Display the value of x using cout.

10. **Left Shift, Right Shift:** Write a program that reads a decimal value from the keyboard, stores it in a variable, left shifts it 4 bits, prints the value, then right shifts it 8 bits and prints the value.

SELF TEST QUESTIONS

1. What's the purpose of the `main()` function? Why does every C++ program need one?
2. Describe how disassembling your programs and studying the results can improve your understanding of the C++ language and your computing platform.
3. What is the purpose of the preprocessor directive `#include`?
4. (True/False) You can use the C++ reserved keywords to name your own program objects.
5. The `sizeof` operator will return the size of data types in what unit of measure?
6. You can either memorize C++ operator precedence or use _____ to force precedence and make source code easier to read at the same time.
7. Why can an unsigned data type represent a larger positive value than a signed data type?
8. When negative numbers are shifted to the right using the `>>` operator what is the effect on the sign bit?
9. How are string literals terminated?
10. Study the following code and answer the following questions:

```
1  #include <iostream>
2  using namespace std;
3
4  int val1 = 1;
5
6  int main() {
7      {
8          cout<<val1++<<endl;
9          int val1 = ::val1;
10         cout<<val1<<endl;
11     }
12     cout<<val1<<endl;
13     return 0;
14 }
```

- What value will the new variable named `val1` be initialized to on line 9?
- Which `val1` will be printed to the screen on line 10?
- What is the value of `val1` printed to the screen on line 12?
- Describe the effect of using the unary scope resolution operator `::` on line 9. Why is its use necessary?

REFERENCES

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Metrowerks CodeWarrior Version 5.5 Reference Documentation for Windows 95/98/NT and Apple Macintosh.

Steve Teale. *C++ IOStreams Handbook*, Addison-Wesley, Reading Massachusetts, 1993, ISBN 0-201-59641-5.

Paul J. Lucas. *The C++ Programmer's Handbook*, Prentice Hall P T R, Englewood Cliffs, New Jersey, 1992, ISBN 0-13-118233-1.

NOTES

CHAPTER 6



Twisted Trees

CONTROLLING THE FLOW OF PROGRAM EXECUTION

LEARNING OBJECTIVES

- *Control the flow of program execution with C++ control flow statements*
- *State the purpose and use of the if statement*
- *Explain the purpose of a null statement*
- *Utilize blocks to create local variable scopes in control statements*
- *State the purpose and use of nested if statements*
- *State the purpose and use of the for statement*
- *State the purpose and use of nested for statements*
- *State the purpose and use of the keywords break and continue*
- *State the purpose and use of the while statement*
- *State the purpose and use of the do statement*
- *State the purpose and use of the switch statement*
- *Explain the importance of using break to exit case statements properly*
- *Explain the importance of a default case*
- *Demonstrate your ability write effective, self-commenting expressions utilizing sound identifier naming techniques*

INTRODUCTION

Control flow statements are a critical component of the C++ programming language, for without them processing would proceed from beginning to end with no detours and how boring would that be?

In this chapter I will show you how to use the selection statements `if`, and `switch`. Selection statements allow you to evaluate an expression and continue processing in a direction based on the result of an expression.

The selection statements will be followed by the iteration statements `while`, `do`, and `for`. Iteration statements are used to do repetitive processing controlled by the evaluation of an expression. I will also cover the use of labeled statements, `goto` statements, `break` statements, `continue` statements, and `null` statements.

Whatever you do, don't forget the material covered in chapter 5. Most of the operators introduced there can be used to build the expressions used in the selection and iteration statements.

Along the way I will show you the idiomatic way of writing each type of statement. Idioms are important in C++ and other languages because knowing the accepted way of writing a `while` loop, `for` loop, etc., increases the readability of your code while reducing the opportunity to introduce syntax errors or logic flaws into your programs.

STATEMENTS, NULL STATEMENTS, AND COMPOUND STATEMENTS

Before talking about `if` statements, `for` statements, `do` statements, and the like, it will be a big help to you to understand statements, `null` statements, and compound statements. If all this talk about statements is making you dizzy just hang on, it will soon make perfect sense.

STATEMENTS

You have seen various statement forms used in the preceding chapters. For instance, expressions, discussed in chapter 5, are a form of statement known as expression-statements. Statements are terminated by a semicolon “;”. A statement will usually result in something happening, as in the case of expression statements that perform a calculation, or assignment expressions where a value is being assigned to some memory location. (remember lvalues and rvalues?) Here are a few examples of statements:

```
int a;
a = 1;
for(int i=0; i<some_val; i++)
    cout<<i<<endl;
```

The first statement is a declaration statement; the variable name `a` is being introduced. The second statement is an assignment expression, a.k.a., expression-statement. The third example is a `for` statement, which will be discussed below. For now, however, see where the semicolon occurs in the `for` statement. It could have been written like so...

```
for(int i=0; i<some_val; i++) cout<<i<<endl;
```

...and you will no doubt see short `for` statements written in this format. The semicolon also serves as a sequence point meaning the statement it terminates and all side effects associated with the statement will be fully executed before the next statement executes.

Statements are executed sequentially, that is, one right after the other, unless the flow of program control is changed with a selection statement, iteration statement, or `goto` statement.

NULL STATEMENTS

Sometimes you need a statement that does nothing. A semicolon appearing by itself serves as a `null` statement. The following is a `null` statement:

;

The null statement doesn't have to be on a line by itself but it often appears that way; it could immediately follow another statement. The following code shows an expression statement followed by a null statement:

```
int a = 25; ;
```

It does look weird doesn't it? Regardless, keep the null statement in mind. You will see it again soon being put to good use.

COMPOUND STATEMENTS

Throughout the rest of this chapter you will see blocks like this:

statement

Where you can use a statement, you can also use a compound statement, otherwise known as a block statement. Compound statements are formed using the opening and closing braces { }. The following code is an example of a compound or block statement:

```
{
  int i = 3;
  cout<<i<<endl;
}
```

You will rarely see compound statements used in the middle of nowhere. They are more often employed in if statements, for statements, while statements, etc. If the body of a selection statement or iteration statement requires more than one statement a compound statement can be used. You will see compound statements used heavily throughout the rest of this chapter.

SELECTION STATEMENTS

Selection statements change the flow of program execution by evaluating the result of an expression, referred to as a condition. The if and switch statements are selection statements.

if STATEMENT

An if statement evaluates a condition and executes its associated statement if the condition is true. Otherwise it skips the statement and processing continues on to the next statement following the if.

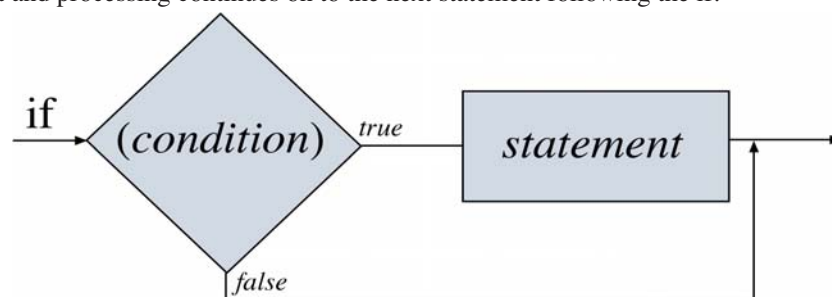


Figure 6-1: if Statement Diagram

Figure 6-1 illustrates the if statement. The expression that formulates the condition must evaluate to either true or false. True is any non-zero integer value. The following code gives an example of a simple if statement:

```
1 int a = 0, b = 1;
2 if(a++ > b)
3     cout<<"a is greater than b"<<endl;
```

6.1 if statement

In this example the postfix ++ operator will increment a after the greater than comparison is made comparing a's value with b's value. This will result in the expression evaluating to false. However, after the if statement executes, a will be 1.

The expressions that form the condition can be simple or complex. I should warn you that the most common mistake novice programmers make with regards to selection statement conditions is using an assignment operator, =, where they really meant to use an equality operator, ==. The following example is perfectly legal in C++:

```
1 int a = 0;
2 if(a = 3)
3     cout<<"Assignment expressions can be conditions too!"<<endl;
```

6.2 assignment

The condition on line 2 evaluates to true because the result of the assignment is the value assigned to a, which is 3, which is a non-zero integer value and therefore considered true. The following example looks similar but is completely different from example 6.2:

```
1 int a = 0;
2 if(a == 3)
3     cout<<"Assignment expressions can be conditions too!"<<endl;
```

6.3 equality

In this example the condition on line 2 will evaluate to false because a is not equal to 3. This leads me to my first good piece of advice regarding selection statements:

*Don't use the = operator
when you really mean to use
the == operator!*

Declarations can be part of the condition, in which case the scope of the declared variable are the statements controlled by the selection statement. Examine the following example:

```
1 if(int a = 3)
2     cout<<"Declarations are allowed in the condition!"<<endl;
```

6.4 declaration in condition

The scope of variable a would be up to the end of line 2.

if STATEMENTS AND COMPOUND STATEMENTS

Compound statements can be used with if statements. Examine the following example:

```

1 int a = 0, b = 1;
2
3 if(++a == b){
4     int c = 2;
5     cout<<"The value of a is: "<<a++<<endl;
6     cout<<"The value of b is: "<<b<<endl;
7     cout<<"The value of c is: "<<c<<endl;
8     cout<<"The value of a is: "<<a<<endl;
9 }

```

6.5 compound statements

All the statements appearing between the opening and closing braces will be executed if the condition on line 3 evaluates to true which it will. The variable `c` declared on line 4 has block scope within the braces from its point of declaration up to the closing brace on line 9.

if-else STATEMENT

The if-else statement works like the if statement with one major difference. If the condition evaluates to false the statement following the else keyword will execute. A diagram of the if-else statement is shown in Figure 6-2.

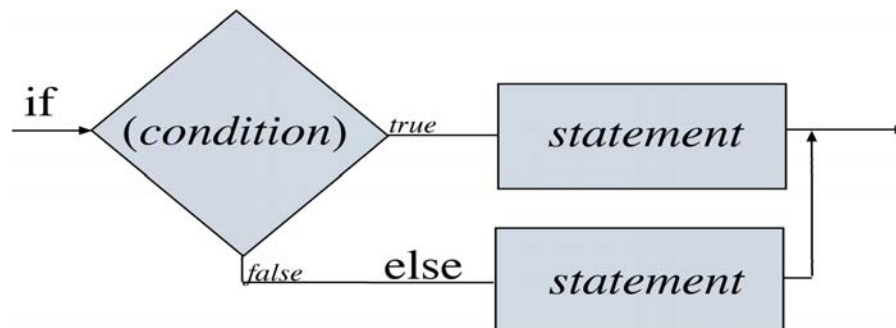


Figure 6-2: if-else Statement Diagram

The following code gives an example of an if-else statement in action:

```

1 int a = 0;
2
3 if(a)
4     cout<<"True statement."<<endl;
5     else
6         cout<<"False statement."<<endl;

```

6.6 if-else

In this example the variable `a` is evaluated resulting in false since `a` is zero. Line 4 will be skipped and the statement on line 6 will execute.

The `!` operator can be used to negate an expression. Examine the following example:

```

1 int a = 0;
2
3 if(!a)
4     cout<<"True statement."<<endl;
5     else
6         cout<<"False statement."<<endl;

```

6.7 ! operator

The statement is read, "if not `a`...". Since `a` is zero, `!a` will evaluate to true causing the statement on line 4 to execute.

Compound statements can be used with if-else statements as illustrated by the following example:

```

1  int a = 1;
2
3  if(!a){
4      int b = 1;
5      cout<<"The value of a: "<<a++<<endl;
6      cout<<"The value of b: "<<b<<endl;
7  }
8  else{
9      int b = 2;
10     cout<<"The value of a: "<<a<<endl;
11     cout<<"The value of b: "<<b<<endl;
12 }

```

*6.8 compound statements
with if-else*

On line 1 the variable `a` is initialized to 1 causing `!a` to evaluate to false. The variable `a` is in scope of the entire if-else statement while each variable `b` has only block scope within each compound statement.

NESTING if-ELSE STATEMENTS

Nesting if-else statements refers to putting one if-else statement within another or chaining them together. One method of nesting if-else statements is to follow the `else` keyword with another if-else statement. The following short program reads a character from the keyboard and prints a message to the screen based on the character entered:

```

1  #include <iostream>
2  using namespace std;
3
4  int main(){
5
6  cout<<"Enter the character a, b, or c: ";
7  char input;
8  cin>>input;
9
10 if(input == 'a')
11     cout<<"You entered a."<<endl;
12 else if(input == 'b')
13     cout<<"You entered b."<<endl;
14     else if(input == 'c')
15         cout<<"You entered c."<<endl;
16         else
17             cout<<"You entered the wrong character!"<<endl;
18
19 return EXIT_SUCCESS;
20 }

```

6.9 nesting if-else

In this complete example a character is read from the input stream and assigned to the variable named `input`. The variable `input` is then compared to three different values in each if-else statement starting on line 10. If the character entered matches the compared value then the expression evaluates to true, otherwise it is false. If the input character is not an 'a', 'b', or 'c' then the final else is executed.

What is helpful to keep in mind is that almost anywhere a statement can be used, a selection statement can be used there as well. Examine the following complete program:

```

1  #include <iostream>
2  using namespace std;
3
4  int main(){
5
6  cout<<"Enter the character a, b, or c: ";
7  char input1;
8  cin>>input1;
9
10 cout<<"Enter the character u or p: ";
11 char input2;
12 cin>>input2;
13
14 if(input1 == 'a'){
15     cout<<"You entered a."<<endl;
16     if(input2 == 'u')
17         cout<<"You entered u."<<endl;
18     else if(input2 == 'p')
19         cout<<"You entered p."<<endl;
20     else cout<<"You didn't enter u or p!"<<endl;
21 }else if(input1 == 'b'){
22     cout<<"You entered b."<<endl;
23     if(input2 == 'u')
24         cout<<"You entered u."<<endl;
25     else if(input2 == 'p')
26         cout<<"You entered p."<<endl;
27     else cout<<"You didn't enter u or p!"<<endl;
28 }else if(input1 == 'c'){
29     cout<<"You entered c."<<endl;
30     if(input2 == 'u')
31         cout<<"You entered u."<<endl;
32     else if(input2 == 'p')
33         cout<<"You entered p."<<endl;
34     else cout<<"You didn't enter u or p!"<<endl;
35 }else {
36     cout<<"You didn't enter a, b, or c!"<<endl;
37     if(input2 == 'u')
38         cout<<"You entered u."<<endl;
39     else if(input2 == 'p')
40         cout<<"You entered p."<<endl;
41     else cout<<"You didn't enter u or p!"<<endl;
42 }
43     return EXIT_SUCCESS;
44 }

```

6.10 use of selection statements

In this example two character inputs are read from the keyboard and compared to character literals in the if statements. This example gives you a good idea of how messy nested if-else statements can become even if liberal indenting is used to improve readability. Luckily, there is an alternative to the nested if-else — the switch statement.

switch STATEMENT

A switch statement evaluates a condition much like nested if-else statements. The condition must evaluate to either an integral value or enumeration value. Enumerations will be covered formally in chapter 10 but I will give you an example of their use with switch statements in this section. Figure 6-3 shows a diagram of the switch statement.

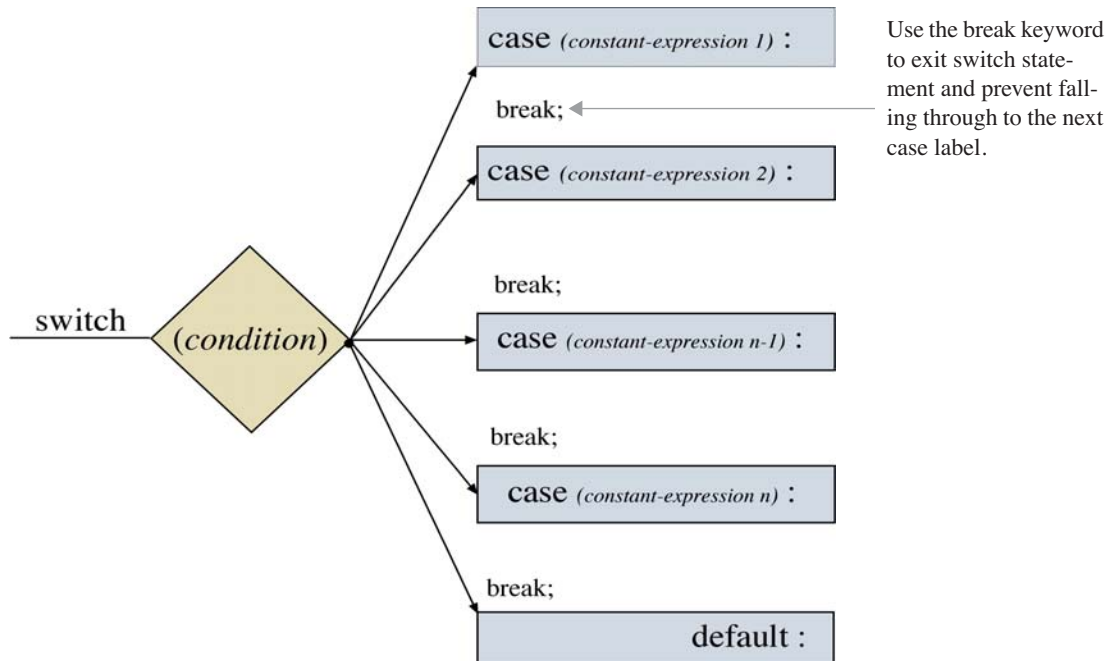


Figure 6-3: switch Statement Diagram

Execution will transfer to the statement whose case label constant expression matches the result of the condition evaluation.

Important!!! Execution of a case statement will fall through to the next case statement unless the keyword `break` is used to exit the switch statement. (Sometimes you want a case statement to fall through to the next case — and sometimes you do not) If a default label is present, and the result of the condition fails to match any of the cases, the default case will execute. Keep the following advice in mind when using switch statements:

Use break to exit the switch and prevent case statement fall-through; always have a default case!

Example 6.11 gives the switch statement version of the nested if-else statement shown in example 6.10:

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5
6 cout<<"Enter the character a, b, or c: ";
7 char input1;
8 cin>>input1;
9
10 cout<<"Enter the character u or p: ";
11 char input2;
12 cin>>input2;
13
14 switch(input1){
15     case 'a':{
16         cout<<"You entered a."<<endl;

```

6.11 switch statement

```

17         switch(input2) {
18             case 'u': cout<<"You entered u."<<endl;
19                 break;
20             case 'p' : cout<<"You entered p."<<endl;
21                 break;
22             default : cout<<"You didn't enter u or p!"<<endl;
23         }
24     break;
25 }
26 case 'b':{
27     cout<<"You entered b."<<endl;
28     switch(input2) {
29         case 'u': cout<<"You entered u."<<endl;
30             break;
31         case 'p' : cout<<"You entered p."<<endl;
32             break;
33         default : cout<<"You didn't enter u or p!"<<endl;
34     }
35     break;
36 }
37 case 'c': {
38     cout<<"You entered c."<<endl;
39     switch(input2) {
40         case 'u': cout<<"You entered u."<<endl;
41             break;
42         case 'p' : cout<<"You entered p."<<endl;
43             break;
44         default : cout<<"You didn't enter u or p!"<<endl;
45     }
46     break;
47 }
48
49 default: {
50     cout<<"You didn't enter a, b, or c!"<<endl;
51     switch(input2) {
52         case 'u': cout<<"You entered u."<<endl;
53             break;
54         case 'p' : cout<<"You entered p."<<endl;
55             break;
56         default : cout<<"You didn't enter u or p!"<<endl;
57     }
58 }
59 } //end switch
60
61 return EXIT_SUCCESS;
62
63 } //end main()

```

6.11 (continued)

The switch statement begins on line 14 and evaluates the character variable `input1`. There are four possible cases that can be executed. If the `input1` evaluates to 'a' then the statements associated with case 'a': on line 15 execute. Case 'a' contains all the statements between the opening brace on line 15 to the closing brace on line 25. Notice that case 'a' also contains a nested switch statement that evaluates the variable `input2`. Case 'b' and case 'c' execute in similar fashion. The break statements in each case will exit the switch and prevent execution falling through to the next case. If none of the cases match the input then the default case is executed. The use of braces to enclose statements associated with a case label are optional.

BREAK AND THE DEFAULT CASE

As long as the default case is the last labeled statement in the switch statement then a break is not required; execution will fall through to the end of the switch. However, if another case label appears beneath the default label then use break to make sure you exit the switch normally. Here is another piece of advice to keep in mind when using a switch statement:

Make the default case the last case in a switch statement!

ITERATION STATEMENTS

Selection statements like if and switch are required to give your programs decision making capability, however, once through the selection statement that is it! Sometimes you want to do things over and over again either forever or at least until something happens to break the loop. This is where iteration statements come into the picture.

In this section I will discuss the while, do, and for statements. Each statement gives you the ability to perform one or more operations repeatedly. Your choice of which iterative statement to use depends on what needs to be done.

***while* STATEMENT**

Figure 6-4 shows a diagram of a while statement. The while statement will evaluate the condition before executing the associated statement. As indicated in figure 6-4, the statement may never execute if the condition is always false.

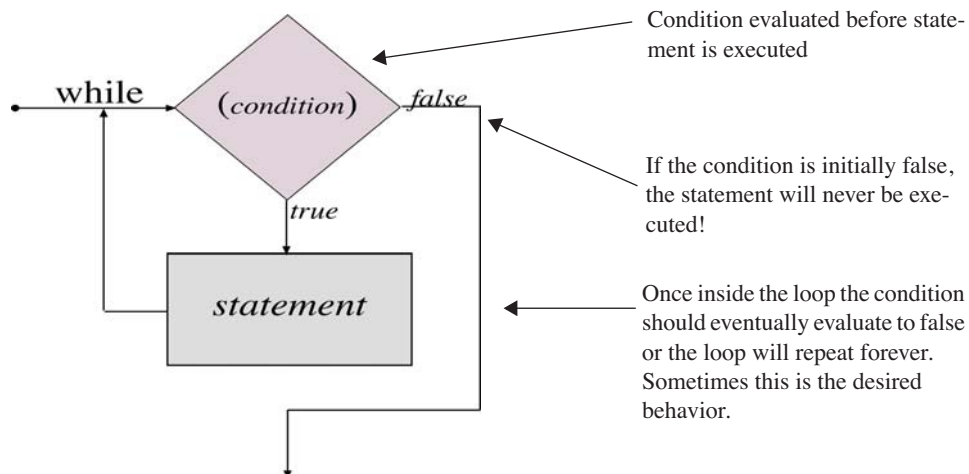


Figure 6-4: while Statement Diagram

Once inside the body of the while loop the condition must eventually evaluate to false or the loop will execute forever. In many programming situations you want to loop forever until the program exits from within the body of the loop. An example of this would be a menu-processing while loop. I will show you an example of this below, but for now, take a look at a simple while statement:

```

1 int count = 0;
2
3 while(count++ < 5){
4     cout<<"Count = : "<<count<<endl;
5     }

```

6.12 while statement

One line 1 the integer variable count is declared and initialized. The condition expression involves the use of two operators. First, the less than comparison is made followed by the postfix increment. When first compared, the variable count is zero, the body of the while loop is entered, and the statement on line 4 executes. What values of count will print to the screen?

In this case, the postfix increment is evaluated prior to entry into the body of the loop. The first value to be printed to the screen is one. This while loop will execute five times. The following while loop behaves somewhat differently:

```

1 int count = 0;
2
3 while(++count < 5){
4     cout<<"Count = : "<<count<<endl;
5     }

```

6.13 while statement

In this example the prefix operator increments the value of count before the less than expression is evaluated. The first value of count to print to the screen is still one but the loop executes only four times. The lesson here is clear: Be sure you understand the effects operators have on your expressions.

CONTROLLING WHILE STATEMENTS WITH SENTINEL VALUES

In most instances you won't know how many times a while loop should execute. In these situations you can employ sentinel values. A sentinel value is simply a value, in most cases arbitrary, of your choosing, that when used in the condition expression of the while loop forces the loop to exit. Sentinel values can be set based upon some condition in the body of the while loop, or, they can be entered via the keyboard. Examine the following code:

```

1 float total = 0, average = 0, input = 0;
2 int count = 0;
3
4 while(input >= 0){
5
6     cout<<"Please enter a positive number to average "
7         <<" or negative number to exit: ";
8     cin>>input;
9
10    if(!cin){
11        cin.clear();
12        cin.ignore(INT_MAX, '\n');
13    }else if(input >= 0){
14        total += input;
15        average = (total/(++count));
16        cout<<"The average is: "<<average<<endl;
17    }
18 }//end while

```

6.14 use of sentinel value

In this example the while loop condition will evaluate to true so long as the number entered is positive. Any negative number will cause the while loop condition to evaluate to false thus exiting the loop. This code also shows an if-else statement being used in the body of a while loop. On line 10 the cin object is examined for failure. If the cin

object fails it means the wrong input type was entered. If the cin object was successful the else statement on line 13 is executed. The else statement includes another if statement that checks to make sure a positive number was entered. If so the averaging calculation is performed, otherwise, the calculation is skipped.

In this case the sentinel value is any negative number. So long as the number entered is greater than or equal to zero the while loop will repeat. A negative number will terminate execution of the while loop.

NESTING WHILE STATEMENTS

While statements can be nested just like their if-else cousins. The following code offers an example:

```

1 bool done = false;
2 int inner_count = 0, outer_count = 0;
3
4 while(!done){
5     while(inner_count < 3){
6         cout<<"Inner count is: "<<inner_count++<<endl;
7     }
8     if(outer_count++ > 3){
9         done = true;
10    }
11
12 inner_count = 0;
13 }
```

6.15 nested while statements

In this example, the outer loop is controlled by a boolean sentinel value which is set to false prior to entry into the loop and subsequently set to true in the body of the outer loop via the if statement on line 8. The inner while loop will execute until the variable inner_loop becomes greater than three, at which time the inner loop terminates. The outer_count variable is compared to three and then incremented. If outer_count is less than three the variable done remains false, inner_count is reset to zero, and the outer while loop executes again. When outer_count is greater than three done is set to true causing the outer loop to terminate.

DOING SOMETHING FOREVER

While statements are often used without sentinel values to indefinitely repeat an operation, leaving the proper exiting of the program to some statement within the body of the while loop. The idiomatic way of writing a while loop so that it repeats forever is shown in the following example:

```

1 while(true){
2     //repeat what's in here forever
3 }
```

6.16 looping forever

The condition of the while loop is simply the boolean literal value true. Alternatively the statement can be written in the following manner...

```

1 while(1){
2     //repeat this stuff too!
3 }
```

6.17 looping forever

...although I don't recommending doing so!

EXITING WHILE LOOPS WITH THE BREAK STATEMENT

The important thing to remember when using forever-repeating while loops is they must be explicitly exited by the programmer. One way to exit a while loop is with the break statement. Examine the following source code:

```

1  int inner_loop = 0, outer_loop = 0;
2
3  while(outer_loop++ < 3){
4
5  cout<<"Outer loop = "<<outer_loop<<endl;
6      while(true){
7          if(inner_loop++ < 3)
8              cout<<"Inner loop = "<<inner_loop<<endl;
9              else break;
10         }
11     }

```

6.18 nested while loop

This example shows nested while loops. The inner while loop, starting on line 6, will repeat forever until the if statement's condition on line 7 evaluates to false, at which time the break statement will be executed exiting the inner loop. Since the variable `inner_loop` is not reset to zero the inner loop executes only once.

A `break` exits its immediate enclosing iteration statement. In example 6.18 above, the `break` statement is part of an if-else statement, which itself is contained within a while loop. Since the inner while loop contains the `break` it is the one exited when the `break` statement is executed.

You can also exit the entire program from a while loop using the `exit()` function. The following extended example shows how a forever-repeating while loop can be used to continuously execute a switch statement that processes keyboard input. The program terminates when the letter `q` or `Q` is entered.

```

1  char input = ' ';
2
3  while(true){
4      cout<<"Enter a character: ";
5      cin>>input;
6
7      switch(input){
8
9          case 'a':
10         case 'A': cout<<"You entered "<<input<<"!"<<endl;
11                 break;
12
13         case 'b':
14         case 'B': cout<<"You entered "<<input<<"!"<<endl;
15                 break;
16
17         case 'c':
18         case 'C':cout<<"You entered "<<input<<"!"<<endl;
19                 break;
20
21         case 'd':
22         case 'D':cout<<"You entered "<<input<<"!"<<endl;
23                 break;
24
25         case 'e':
26         case 'E':cout<<"You entered "<<input<<"!"<<endl;
27                 break;
28
29         case 'f':
30         case 'F':cout<<"You entered "<<input<<"!"<<endl;
31                 break;
32
33         case 'g':
34         case 'G':cout<<"You entered "<<input<<"!"<<endl;

```

6.19 switch inside of while loop

```
35         break;
36
37     case 'h':
38     case 'H':cout<<"You entered "<<input<<"!"<<endl;
39         break;
40
41     case 'i':
42     case 'I':cout<<"You entered "<<input<<"!"<<endl;
43         break;
44
45     case 'j':
46     case 'J':cout<<"You entered "<<input<<"!"<<endl;
47         break;
48
49     case 'k':
50     case 'K':cout<<"You entered "<<input<<"!"<<endl;
51         break;
52
53     case 'l':
54     case 'L':cout<<"You entered "<<input<<"!"<<endl;
55         break;
56
57     case 'm':
58     case 'M':cout<<"You entered "<<input<<"!"<<endl;
59         break;
60
61     case 'n':
62     case 'N':cout<<"You entered "<<input<<"!"<<endl;
63         break;
64
65     case 'o':
66     case 'O':cout<<"You entered "<<input<<"!"<<endl;
67         break;
68
69     case 'p':
70     case 'P':cout<<"You entered "<<input<<"!"<<endl;
71         break;
72
73     case 'q':
74     case 'Q':cout<<"You entered "<<input<<"!"<<endl;
75         cout<<"Goodbye!";
76         exit(EXIT_SUCCESS);
77
78     case 'r':
79     case 'R':cout<<"You entered "<<input<<"!"<<endl;
80         break;
81
82     case 's':
83     case 'S':cout<<"You entered "<<input<<"!"<<endl;
84         break;
85
86     case 't':
87     case 'T':cout<<"You entered "<<input<<"!"<<endl;
88         break;
89
90     case 'u':
91     case 'U':cout<<"You entered "<<input<<"!"<<endl;
92         break;
93
```

```

94     case 'v':
95     case 'V':cout<<"You entered "<<input<<"!"<<endl;
96         break;
97
98     case 'w':
99     case 'W':cout<<"You entered "<<input<<"!"<<endl;
100        break;
101
102     case 'x':
103     case 'X':cout<<"You entered "<<input<<"!"<<endl;
104        break;
105
106     case 'y':
107     case 'Y':cout<<"You entered "<<input<<"!"<<endl;
108        break;
109
110     case 'z':
111     case 'Z':cout<<"You entered "<<input<<"!"<<endl;
112        break;
113
114     default: cout<<"Character not part of alphabet!"<<endl;
115             break;
116 } //end switch
117 }// end while

```

6.19
continued

do STATEMENT

The do statement differs from the while statement in the position of the condition test. In the while statement the condition test took place before the loop was executed. This resulted in a possibility of the while loop never executing the body of the loop. The do statement tests the condition after the body of the loop. This results in the body of the do loop executing at least once.

Figure 6-5 shows a diagram of the do statement. The following source code shows a do statement in action:

```

1 int count = 0;           6.20
2
3 do {
4   cout<<"Count ="<<count<<endl;
5 }while(count++ < 3);

```

NESTING do STATEMENTS

Do statements can be nested in exactly the same way as while statements.

for STATEMENT

There is often a need in programming to repeat a series of steps a known amount of times. You can do this with the while statement or the do statement and you have seen it done several times already in this chapter. To make a while or do statement iterate for a specified number of loops you must perform the following general steps:

Step 1 - Declare an integer variable with which to keep count of the number of loops,

Step 2 - Initialize the counting variable declared in step one (step one and two can be combined into one statement),

Step 3 - Test the condition,

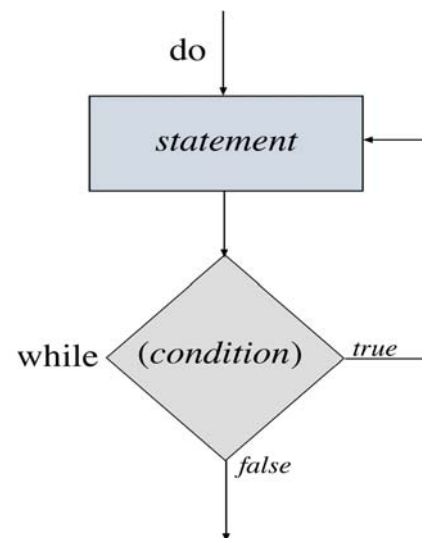


Figure 6-5: do Statement Diagram

Step 4 - Perform the statement or statements in the body of the statement if condition is true,

Step 5 - Increment the loop counting variable,

Step 6 - Repeat steps 3 through 5 until the condition evaluates to false.

The following example shows a while statement with steps 1 through 5 labeled:

6.21 while loop behaving like for loop

```

1 int count = 0;
2
3 while(count <= 3){
4     cout<<"Loop executed ";
5     cout<<count<<" times!"<<endl;
6     count++;
7 }

```

Steps 1 & 2: count variable declared and initialized

Step 3: Condition tested

Step 4: Body executed

Step 5: count variable incremented

The for statement provides a convenient format for performing steps 1, 2, 3, and 5, as shown in figure 6-6.

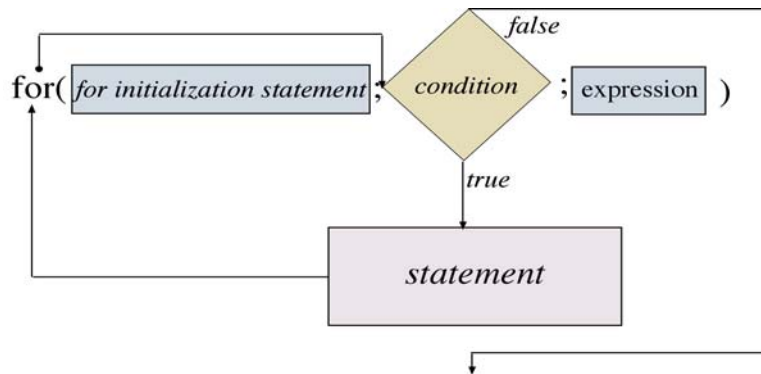


Figure 6-6: for Statement Diagram

The following source code shows a for statement in action:

```

1 for(int i = 0; i<3; i++){
2     cout<<"I equals = "<<i<<endl;
3 }

```

6.22 for statement

On line one, the variable *i* is being declared and initialized for the purpose of loop counting. The condition is tested and, if true, the body of the for statement is performed, and *i* is incremented. The scope of the variable *i* is local to the body of the for statement and is visible to any enclosing scopes. Examine the following code:

```

1 for(int i = 0; i<3; i++){
2     cout<<"I equals = "<<i<<endl;
3 }
4 int i = 0;

```

6.23 for loop scope

On line four another variable *i* is being declared and defined. This is legal because the *i* declared in the for statement initialization section on line one is visible only in the body of the for statement.

I'd like to point out that example 6.22 illustrates the idiomatic way of writing a for statement. Writing the for statement in the idiomatic way makes your source code easier to read and comprehend.

For statements are an ideal way to implement summations in source code. Example 6.24 implements the following summation:

$$\sum_{i=1}^{100} i$$

```

1 for(int i = 1, total = 0; i<=100; i++){
2     total += i;
3     cout<<"i = "<<i<<" ";
4     cout<<"Total = "<<total<<endl;
5 }
```

6.24 implementing summation

Note on line one how the variable total is declared and initialized in the for statement initialization section as well as the variable i.

NESTING FOR STATEMENTS

For statements can be nested like the while and do-while statements. The following code gives an example:

```

1 for(int i = 0; i<3; i++){
2     cout<<"i = "<<i<<endl;
3     for(int j = 0; j<3; j++){
4         cout<<"j = "<<j<<endl;
5     }
6 }
```

6.25 nesting for statements

When nesting for loops you have to be especially aware of variable scoping within each for loop's body block. In this example, the variable i is in scope for the outer loop as well as the inner loop. Most times, however, a separate counting variable is required to keep track of enclosed nested loop iterations. The variable j is declared in the for loop on line 3 for this purpose.

Nesting for loops in this fashion results in the inner loop being executed each time the outer loop is executed.

BREAK

The break statement can be used to terminate for loops. The break statement exits its enclosing loop. Examine the following source code:

```

1 for(int i = 0; i<3; i++){
2     cout<<"i = "<<i<<endl;
3     for(int j = 0; j<3; j++){
4         cout<<"j = "<<j<<endl;
5         if(j == 1)
6             break;
7     }
8 }
```

6.26 break statement

This code is similar to example 6.25 with the addition of the if statement on line 5. If the variable j, which is visible only in the body of the inner for statement, is equal to 1, then the inner loop will terminate and control will be passed to the outer for statement.

DOING SOMETHING FOREVER WITH A FOR STATEMENT

A for statement can also be used to perform an operation repeatedly forever. The following code shows how this

is done:

```
1 for(;;){
2     //do this forever
3 }
```

6.27 looping forever

CONTINUE

The continue keyword can be used with while, do, and for statements to pass control to the end of the loop. Here's an example:

```
1 char ch = ' ';
2 int count = 0;
3
4 cout<<"Enter characters or numbers: "<<endl;
5
6 while(cin>>ch){
7     if((ch >= '0') && (ch <= '9')){
8         continue;
9     }
10
11     cout<<"Count is "<<++count<<" and the character is ";
12     cout<<ch<<endl;
13
14     if(ch == 'q'){
15         cout<<"Goodbye!"<<endl;
16         break;
17     }
18 }
```

6.28 continue statement

This while loop will repeat as long as characters are input via the keyboard or until the q character is entered. If a 0 through 9 is entered the continue statement on line 8 causes any statements appearing on lines 9 through 17 to be skipped.

Avoiding BREAK AND CONTINUE

Use break and continue statements sparingly. Their overuse leads to hard-to-understand code and dang-near-impossible-to-detect programming errors. Break is used mostly in switch statements, otherwise, there is usually a more elegant way of rewriting code that avoids break and continue. The ability to write elegant code comes from experience.

Writing ELEGANT Code

I'd like to expand on the last sentence of the previous paragraph. Writing code is closely related to writing a story or a novel. You will find that as you are happily programming along you will write whatever code pops into your mind as you type along. As with ordinary writing, rarely will you write code that can't be improved to a certain degree, either in function or appearance, from editing or a complete rewrite. Experience brings with it the ability to edit on the fly.

However, don't try too hard to write "beautiful" code. Rather, ensure you have chosen, or developed, the optimum algorithm with which to solve a particular problem. Implement the algorithm with your best possible effort and let the optimizing compiler do its magic.

Labeled Statements

There are three types of labeled statements: case, default, and identifier. You have seen the case and default labeled statements used in the switch statement. In this section I will discuss the identifier labeled statement.

A statement can be preceded with an identifier label which consists of the label's name and a colon as shown in figure 6-7. The only use of such labeled statements is with the goto statement

goto STATEMENT

The goto statement performs an unconditional jump either forward or backward in source code to a labeled statement. The range of the jump is within the current function. In other words, you can't jump out of a function into another function, or back to the calling function, using a goto statement. But then you wouldn't want to do that anyway! The following example shows a goto statement in action:

```

1 int count = 0;
2
3 start: cout<<"This is the start!"<<endl;
4
5 if(count++ < 3)
6     goto start;
7
8 cout<<"All done!"<<endl;

```

6.29 labeled statement

In this example line 3 is labeled with the identifier start. The if statement will evaluate count and, if it is less than three, will execute the goto statement on line 6. This code can be rewritten without a goto statement by using a while loop as in the following example:

```

1 int count = 0;
2
3 while(count++ < 3)
4     cout<<"This is the start!"<<endl;
5
6 cout<<"All done!"<<endl;

```

6.30 gotoless code

Advice On Using Goto

If you must use a goto statement keep the jump within eye sight on the same page of code.

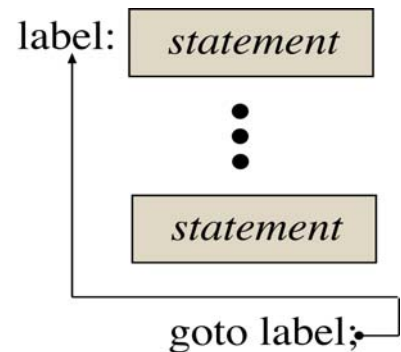


Figure 6-7: goto Statement Diagram

CONTROL STATEMENT USAGE GUIDE

Table 6-1 will come in handy when deciding when to use a particular C++ control statement:

Control Statement	Use
<i>if</i>	...for simple decision
<i>if-else</i>	...for decision with alternative action
<i>switch</i>	...in place of nested if-else statements where the condition is an integral or enumeration type
<i>while</i>	...when condition must be tested first
<i>do</i>	...when body must be executed at least once
<i>for</i>	...to loop for a known amount of times
<i>break</i>	...in case statement to prevent case statement fall-through; ...to terminate while, do-while, and for loops.
<i>continue</i>	...to skip to the end of a while, do-while, or for loop.
<i>goto</i>	...to perform unconditional jump to a labeled statement

Table 6-1: Control Statement Usage Guide

SUMMARY

Program flow can be controlled using selection, iteration, and goto statements. The selection statements include the if and switch statements; the iteration statements include the while, do, and for statements.

Use the if statement to perform a simple decision operation. Be on the watch for conditions that always evaluate to false. Use the if-else statement to provide a path of execution should the condition evaluate to false. The switch statement should be used in place of nested if-else statements where the condition is an integral or enumeration type. Place a break statement at the end of every case to prevent case fall-through. Every switch should have a default case. If the default case is placed last in the switch statement a break is not necessary, but a good idea nonetheless.

Be careful not to use the assignment operator in a condition expression when you really mean to use the equal to operator. This error alone causes more heartache for novice students and professional programmers than any other I know.

The while statement places the condition test before the body statements are executed; the do places the condition test after the body statements ensuring they are executed at least once. Use a for statement when the number of loop repetitions is known. It provides a convenient format for statement variable initialization, condition testing, and expression evaluation.

If you use while, do, or for statements to perform indefinite looping either use sentinel values or provide some other way to terminate the loop.

Break and continue statements give you an extra measure of iteration statement control but use them sparingly.

Goto statements provide a way to perform unconditional jumps within your program. If you use them keep the jump within eye sight.

Skill Building Exercises

1. **if statements:** Write a program that declares two integer variables, a and b, initialized to values of your choice. In this program write ten if statements that use the following conditions:

<code>(a <= b)</code>
<code>!(b == a)</code>
<code>(3 < a) && (3 < b)</code>
<code>(a b)</code>
<code>((++a) == (--b)) b)</code>
<code>(a ^ b)</code>
<code>((a && b && (!0)) true)</code>
<code>(b == 10)</code>
<code>(int c = b)</code>
<code>(a (!b))</code>

Disassemble your source code and examine how each if statement and its condition is implemented in assembly.

2. **if-else statement:** Write a program that declares two char variables, a and b, and initialize them both to `' '`. Write ten if-else statements using the following condition expressions:

<code>(a = 'y')</code>
<code>(a == 'Y')</code>
<code>(!(b = 'n'))</code>
<code>(b == 'n')</code>
<code>('c' <= b)</code>
<code>(('A' <= a) && (a <= 'Z'))</code>
<code>(a == b)</code>
<code>(a != b)</code>
<code>((b >= a) (a <= '1'))</code>
<code>(a && b && '0')</code>

3. **nested if-else statements:** Write a program using nested if-else statements that reads character input from the keyboard one character at a time and performs the following operations:
- if the input is '1' through '5', add 1 to the character value and print the resulting character to the screen,
 - if the input is '6' through '9', add 5 to the character value and print the resulting character to the screen,
 - if the input is 'a' through 'z', convert to upper case and print to the screen,
 - if the input is 'A' through 'Z', convert to lower case and print to the screen.
4. **switch statement:** Rewrite exercise three above using a switch statement.
5. **while statement:** Add a while statement to the code you wrote in exercise four to repeatedly process keyboard input. Use a sentinel value to terminate the loop.

6. **do-while statement:** Use do-while statements to calculate the following: Prompt the user for the value of n.

$\sum_{i=1}^n i^2$
$n!$
$\sum_{i=1}^n i + 2$
$\left(\sum_{i=1}^n i \right) \left(\sum_{j=1}^n j \right)$

7. **for statement:** Rewrite exercise 6 using for statements.

SUGGESTED PROJECTS

1. **Game Program:** Write a game program that generates a random number between 0 and 100. Prompt the user to guess what number was generated. Keep track of the number of guesses the user makes and print the statistics to the screen at the end of each game.
2. **Adder:** Write a program that continuously adds numbers entered via the keyboard until zero is entered. Print the running total to the screen after the entry of each number.
3. **Calculator:** Write a calculator program. Have the program alternately prompt the user for numbers, then operators. Implement the following operators: +, -, *, /, %, and C for clear. Print the results of each operation.
4. **Converter:** Write a program that calculates the integer value of characters input via the keyboard.
5. **Create Multiplication Tables:** Write a program that prints multiplication tables. Prompt the user for the multiplication table to print. Calculate the table up to the 12th factor.
6. **Weight Guesser:** Write a program that tries to guess the weight of the user. Prompt the user for their sex, age, height, and level of physical activity. (hi, moderate, or low) Use their input as a starting point for the computer's guess. If a guess is high, have the user enter 'h'; if the guess is low have the user enter 'l'. If the guess is right on have the user enter 'y'. Keep statistics on the number of high, low, and total guesses the computer makes. Print the statistics at the end of each game. Ask the user if they want to play again or quit.
7. **Calculate Area:** Write a program that calculates the area of a circle given its radius. Prompt the user for the radius. After the calculation is complete print the results to the screen and prompt the user to continue or quit.

8. **Calculate Sin, Cosine, and Tangent:** Write a program that calculates and prints the sin, cos, and tangent tables for triangles given the values of their side, angle, and side.
9. **Temperature Converter:** Write a program that converts fahrenheit to centigrade.
10. **Course Made Good:** Write a program that calculates the sum of distance vectors movement 360 degrees. Prompt the user to enter a direction in degrees and a distance in units. Calculate the direction and distance made good.

Self Test Questions

1. What characters are used to begin and end compound statements?
2. How does the if-else statement differ from the if statement?
3. What control statement can be used in place of nested if-else statements?
4. Why should a break statement be added to the end of each case of a switch statement?
5. What's the purpose of a default case in a switch statement?
6. What's the difference between a while and a do-while statement? When would you use a do-while instead of a while statement?
7. Convert the following while statement to a for statement:

```
int i = 0;
while (i < 10) {
    //do something
    i++;
}
```

8. What will happen when the following code executes:

```
int i = 0;
while (i++ < 3) {
    cout << "i = " << i << endl;
    break;
}
```

9. What happens when the following code executes:

```
int i = 0;
while (i < 3) {
    cout << "i = " << i << endl;
    continue;
    i++;
}
```


10. True/False: Goto statements can be used to jump outside of functions.

REFERENCES

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Metrowerks CodeWarrior Version 5.5 Reference Documentation for Windows 95/98/NT and Apple Macintosh.

Steve Teale. *C++ IOStreams Handbook*, Addison-Wesley, Reading Massachusetts, 1993, ISBN 0-201-59641-5.

Paul J. Lucas. *The C++ Programmer's Handbook*, Prentice Hall P T R, Englewood Cliffs, New Jersey, 1992, ISBN 0-13-118233-1.

NOTES

CHAPTER 7



Balboa Hall

POINTERS AND REFERENCES

LEARNING OBJECTIVES

- *STATE THE PURPOSE AND USE OF POINTERS AND REFERENCES IN C++*
- *STATE THE DEFINITION OF AN OBJECT*
- *EXPLAIN HOW TO DETERMINE AN OBJECT'S ADDRESS USING THE & OPERATOR*
- *EXPLAIN HOW TO DECLARE POINTERS USING THE POINTER DECLARATOR **
- *EXPLAIN HOW TO DEREFERENCE A POINTER USING THE * OPERATOR*
- *DESCRIBE THE CONCEPT OF DYNAMIC MEMORY ALLOCATION*
- *EXPLAIN HOW TO DYNAMICALLY CREATE OBJECTS USING THE NEW OPERATOR*
- *EXPLAIN HOW TO DESTROY OBJECTS USING THE DELETE OPERATOR*
- *EXPLAIN HOW TO DECLARE REFERENCES USING THE REFERENCE DECLARATOR &*
- *EXPLAIN WHY REFERENCES MUST BE DEFINED AT THE POINT OF DECLARATION*
- *DESCRIBE THE BENEFITS OF USING REFERENCES VS. POINTERS*
- *UTILIZE POINTERS AND REFERENCES TO CREATE POWERFUL C++ PROGRAMS*

INTRODUCTION

Mastering the concept and use of pointers is crucial to understanding just about all other aspects of the C++ language. An understanding of pointers is required to really understand what's going on in arrays, dynamic object creation, function argument passing by reference, iterators, and a whole host of other language topics.

Pointers are easy to learn but can be confusing at first. I think the reason for this is that C++ uses the same operator, like the asterisk `*` for instance, for several purposes. The purpose you're most familiar with at this point is multiplication. But the asterisk is also used to declare and dereference pointer variables. The concept of using an operator for more than one purpose in C++ is referred to as operator overloading and is done to keep the language small and manageable. The compiler knows how to deal with overloaded operators based upon what context the operator appears. This is convenient for the compiler and the compiler writer but confuses the novice to no end. You'll learn more about overloaded operators in chapter 14.

The question students ask when they first encounter pointers is, "OK...but what are they good for?" That's a fair question. Pointers let you write leaner, meaner code. Pointers, combined with dynamic object instantiation, (creating objects at runtime using the `new` operator) let you conserve memory space, creating objects only when needed in a program. Pointers let you do things faster. An example of this can be found in sorting operations where the objects being compared are large. The objects themselves can remain in their original memory locations and only their addresses need be manipulated during the sort operation. Pointers let you pass the address of a large object to a function for processing without moving the object itself. Additionally, an understanding of pointers is key to understanding more complex data structures like linked lists and trees.

The purpose of this chapter, then, is to give you a solid understanding of pointers, what they are, how to declare them, how to dereference them, how to dynamically create objects in memory using the `new` and `delete` operators, and how to determine the address of objects in memory. I do not cover all the different ways to use pointers in this chapter because doing so would require discussing aspects of the language to which you have not yet been exposed. An example of this would be the use of pointers with functions which is left to chapter 9.

BUT FIRST, A SHORT STORY

Perplexed One sat listening to the professor drone on about C++ pointers. He tried hard to stay awake but couldn't and his head dropped to the desk with a dull thud. He was soon fast asleep.



Droning Professor



Perplexed One



Fast Asleep!

Poor Perplexed One. It looked as if pointers were about to get the best of him, but alas, he was roused from his peaceful state by the sound of cats screeching and dogs howling! Could it be? Yes, it's C++ Man!



C++ Man

Perplexed One looked up to see C++ Man hovering where the droning professor once stood.

“Tell me Perplexed One, why are pointers putting you to sleep?” C++ Man asked.

“I just don’t get ’em...” Perplexed One answered, nervously looking around the room wondering where all the other students had disappeared to.

“Don’t let pointers get you down, they’re really quite simple. Tell you what, write down all your questions about pointers and I’ll answer each one.

Perplexed One agreed and before long he handed C++ Man the following list:

*What is an object?
 What is a memory address?
 How do you determine an object’s memory address?
 What is a pointer?
 How do you declare a pointer?
 How do you access the object a pointer points to?
 How do you create objects dynamically with the new operator?
 How do you delete dynamically created objects with the delete operator?
 What’s the difference between a pointer and a reference?
 How do you declare and use references?*

“These are great questions Perplexed One” C++ Man said as he poured over the list. “Let’s get started right away beginning with the first question!”

Perplexed One agreed excitedly and took out a piece of paper to take notes.

WHAT IS AN OBJECT?

According to the ANSI C++ standard an object is a region of storage. An object can be a fundamental data type like an integer, char, etc. In the case of fundamental data types the regions of memory occupied by each data type are set by the environment. For instance, on a 32-bit computer with a 32-bit wide memory organization, an integer will occupy four contiguous bytes. Thus, an integer object occupies a region of memory four bytes wide.

Objects can be user-defined or abstract data types as well. Abstract data types are types you create by combining fundamental data types or other abstract data types in order to model the problem you are trying to solve on the computer. Most abstract data types will be either structures or classes. The region of memory occupied by an abstract data type is dictated by its composition. Abstract data types are discussed in detail in chapter 10.

An object can be created in three ways:

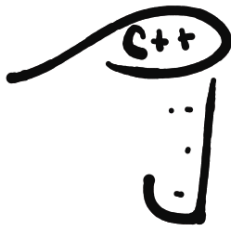
1. by definition,
2. by using the new or new[] operator,
3. or by the compiler when required.

C++ Man waved his hand at the board causing the following code to appear:

```
1 int a = 1, b = 2;
2
3 cout<<"a = "<<a<<endl;
4 cout<<"b = "<<b<<endl;
```

7.1 integer objects

In this example, Perplexed One, there are two integer objects being created via definition. The variable names `a` and `b` are NOT the objects, but the identifiers that become bound to a region of memory at compile time. The process of associating an identifier with a memory location is called binding. In this case the size of the memory region associated with each of the variables `a` and `b` is four bytes which is the size of an integer object.



“So you see Perplexed One, an object occupies a region of memory. Objects can be simple data types and occupy only a small amount of memory or they can be complex data types and occupy large amounts of storage. While your computer is running, memory is filled with many different kinds of objects of all different sizes.”

“Remember, identifier names appearing in your source code are bound to their object locations during the compilation process.”

“Thanks C++ Man! I think I finally understand the concept of objects. I always used to confuse the identifier names with the objects they represented.”

Perplexed One scribbled a few more notes then looked up at C++ Man. “OK C++ Man, I’m ready to discuss the next question on the list!”

“Great, Perplexed One! Let’s do it.”



WHAT IS A MEMORY ADDRESS?

C++ Man waved his hand at the board causing the following diagram to appear:

“Here’s a simple diagram showing the arrangement of computer memory Perplexed One. Memory is essentially an array of byte addressable elements in which binary values can be stored. Memory addresses start at zero and go up to however much memory you have in your computer. For instance, a computer with 256 megabytes of main memory has more storage, and therefore more byte addressable storage elements, than a computer with only 128 megabytes of memory.”

“If you recall from chapter 4, the hardware architecture of the computer system dictates how physical memory is arranged and accessed, but here we will assume that the word size is 32 bits. Looking at figure 7-1, memory starting at address 0 contains the four bytes with addresses 0, 1, 2, and 3 for a total of 32 bits. The 32-bit memory words are aligned on addresses divisible by 4, which is why the addresses 0, 4, 8, 12, etc., are going up the left side of memory in the diagram.”

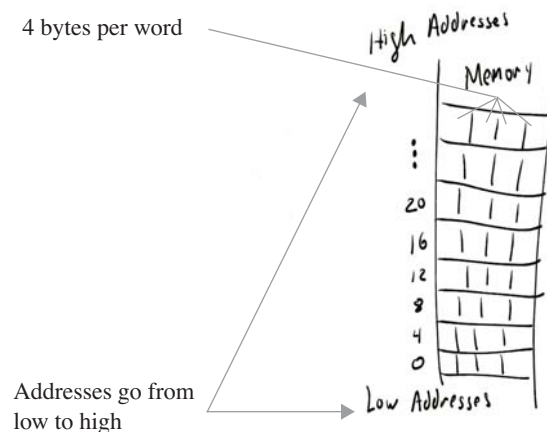


Figure 7-1: Memory

“Figure 7-2 shows another way of representing memory that reinforces the idea that it is a contiguous array of elements.

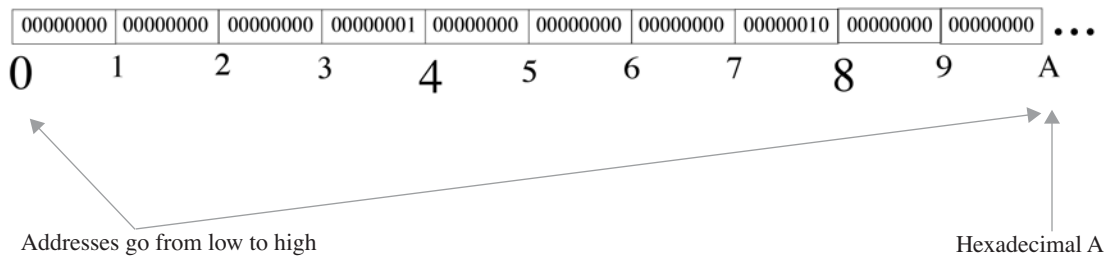


Figure 7-2: Another Way To Represent Memory

Figure 7-2 also shows the memory initialized with two integer values. The integer object beginning at address 0 has a value of 1. The second integer object begins at address 4 and contains the value 2. This way of thinking of memory will come in handy when you learn about arrays in chapter 8.”

Perplexed One studied the two diagrams for a while. “Why did you use hexadecimal A in figure 7-2 to represent the address 10?”

“Good question Perplexed One!” C++ Man answered. “You’ll find it helpful in your studies of C++, and computers in general, to become accustomed to using hexadecimal. It’s easier to work with than large decimal numbers. Figure 7-3 is another diagram of memory using hexadecimal addresses.”

C++ Man lifted his pinky finger and figure 7-3 appeared on the board.



“So you see, Perplexed One, all objects in memory are accessible via an address. The addresses go from low to high and are organized according to the hardware architecture of the computer. A large object may span several addresses but have only one starting address.”

Perplexed One looked up from his notes. “This is cool. I was totally lost but now I understand the whole addressing thing.”

C++ Man beamed with pride. “Are you ready to talk about the next question Perplexed One?” he asked.

“Ready! Perplexed One replied.”



...				
3C	00000000	00000000	00000000	00000000
38	00000000	00000000	00000000	00000000
34	00000000	00000000	00000000	00000000
30	00000000	00000000	00000000	00000000
2C	00000000	00000000	00000000	00000000
28	00000000	00000000	00000000	00000000
24	00000000	00000000	00000000	00000000
20	00000000	00000000	00000000	00000000
1C	00000000	00000000	00000000	00000000
18	00000000	00000000	00000000	00000000
14	00000000	00000000	00000000	00000000
10	00000000	00000000	00000000	00000000
C	00000000	00000000	00000000	00000000
8	00000000	00000000	00000000	00000000
4	00000000	00000000	00000000	00000000
0	00000000	00000000	00000000	00000000

Figure 7-3: Hexadecimal Addressing

How do you DETERMINE AN OBJECT’S MEMORY ADDRESS?

“An object’s memory location can be determined by using the & operator” C++ Man said. Perplexed One looked even more perplexed.

“What do you call & operator?” he asked, pencil ready to record C++ Man’s answer.

“I call it the ‘address-of’ operator” C++ Man replied, waving his hand in the direction of the board. “Let’s see how it works.”

```

1 int a = 3;
2
3 cout<<"          a = "<<a<<endl;
4 cout<<"Address of a = "<<&a<<endl;

```

“In this example, an integer variable named `a` is declared and defined. Remember, definition is one way objects are created in memory. On line 3 the value of `a` is being printed while on line 4 the `&` operator is applied to the variable `a` in order to determine the address of the object to which the identifier `a` is bound to. Let’s disassemble this code and study it before running the program. The listing has been edited to make it easier to read!”

```

Hunk:Kind=HUNK_GLOBAL_CODE  Align=4  Class=PR  Name=".main"(22)  Size=136
00000000: 7C0802A6  mflr      r0
00000004: 90010008  stw       r0,8(SP)
00000008: 9421FFC0  stwu     SP,-64(SP) ← Set stack register (SP) to our stack
0000000C: 38000003  li        r0,3 ← Load 3 into r0 register
00000010: 90010038  stw       r0,56(SP) ← Store 3 at offset 56 from stack pointer
00000014: 80620000  lwz      r3,cout__3std(RTOC)
00000018: 80820000  lwz      r4,@661(RTOC)
0000001C: 48000001  bl       .__ls<Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream ← Load character a
<c,Q23std14char_traits<c>>PCC and print
00000020: 60000000  nop
00000024: 80810038  lwz      r4,56(SP) ← Load 3 into register r4...
00000028: 48000001  bl       .__ls_Q23std39basic_ostream<c,Q23std14char_traits<c>>Fi ← ...then print
0000002C: 60000000  nop
00000030: 80820000  lwz
00000034: 48000001  bl
00000038: 80620000  lwz      r4,endl<c,Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream
<c,Q23std14char_traits<c>>(RTOC)
0000003C: 80820000  lwz      r3,cout__3std(RTOC)
00000040: 48000001  bl       .__ls_Q23std39basic_ostream<c,Q23std14char_traits<c>>FPFRQ23std39basic_ostream
<c,Q23std14char_traits<c>>_RQ23std39basic_ostream<c,Q23std14char_traits<c>>
00000044: 60000000  nop
00000048: 38810038  addi     r4,SP,56 ← Address of a = value in SP + 56
0000004C: 48000001  bl       .__ls_Q23std39basic_ostream<c,Q23std14char_traits<c>>FPCv ← print
00000050: 60000000  nop
00000054: 80820000  lwz
00000058: 48000001  bl
0000005C: 38600000  li        r3,0
00000060: 80010048  lwz      r0,72(SP) ← endl, print, housekeep, reset SP, and return
00000064: 38210040  addi     SP,SP,64
00000068: 7C0803A6  mtlr     r0
0000006C: 4E800020  bblr
00000070: 00000000  dc.l     $00000000 ; traceback table
00000074: 00092041  dc.l     $00092041
00000078: 80000000  dc.l     $80000000
0000007C: 00000070  dc.l     $00000070
00000080: 00052E6D  dc.l     $00052E6D
00000084: 61696E00  dc.l     $61696E00

```

Listing 7.1: Example 7.2 Disassembled

“Running example 7.2 can produce different results, depending on what other programs are running on the computer at the same time. Figure 7-4 shows the results of running the program from the CodeWarrior programming environment.”

“Notice here, Perplexed One, that the address of `a` is shown to be `0x18da0078`. Shutting down all other programs and then running example 7.2 again produced the results shown in figure 7-5.”

“Notice in figure 7-5 that the address of `a` has changed to `0x19f72078`. If you try this experiment on your computer at home, Perplexed One, you will get different results from those shown here.”

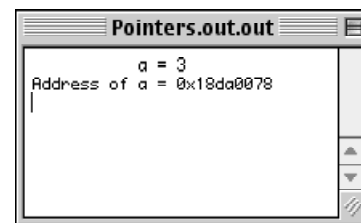


Figure 7-4: Running Example 7.2 with CodeWarrior

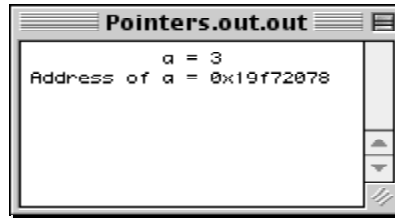
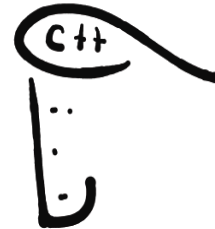


Figure 7-5: Running Example 7.2 Alone

“The disassembled code shown in listing 7-1 shows that the integer object of example 7.2 will always be located at an offset 56 bytes from the value of the stack. This could be anywhere in main memory depending on where the program loads.”

Luckily, you don’t have to worry about the actual addresses. That’s the job of the operating system. The important thing to remember is that the & operator will return the address of an object’s memory location, whatever that address may be.”



Perplexed One quickly summarized what he had learned so far. “OK, an object is a region of memory, and all objects have an address that can be determined by using the & operator.”

“That’s great Perplexed One!” C++ Man said smiling. “Now that you understand the foundation material it’s time to talk about pointers for real. Are you ready?”

“Ready!” Perplexed One said, taking out another piece of paper.

WHAT IS A POINTER?

C++ Man sneezed, causing the following diagram to appear on the board:

“I’ll begin with a definition of a pointer and then explain in greater detail” C++ Man said floating from one end of the board to the other.

“A pointer is a variable that holds a memory address. The important part of this definition is the word variable. You can assign an object’s address to a pointer variable and later, change the contents of the pointer to point to some other object.”

“When a pointer contains the address of an object located somewhere in memory, the pointer is said to point to that object, hence the term pointer.”

“Pointer variables are all the same size, which is the size of the address bus. If the machine you are using has a 32-bit address bus, then a pointer will be 4 bytes long. However, a pointer can only contain addresses to objects of its declared type. For instance, a pointer to float objects can’t have the address of integer objects assigned to it. The reason for this is that the compiler needs to know what size objects a pointer is pointing to.”

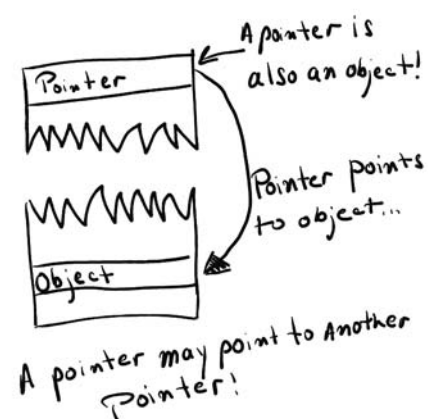


Figure 7-6: Pointer

“Pointers can point to other pointers, that is, a pointer can contain the address of a pointer.”



“Remember, Perplexed One, that a pointer is simply a variable that can hold an object’s address. Because it’s a variable, the address a pointer contains can be changed, just like any other variable.”

“I see.” said Perplexed One, staring at the diagram on the board. “But how do you tell a pointer what size objects it can point to?”

“That’s an excellent question!” C++ Man said, “And it’s the next one on your list!”



How do you DECLARE A POINTER?

“A pointer is declared with the help of the asterisk *. Let’s look at an example.” C++ Man waved his hand at the board and the following code appeared:

```
1 int a = 3;
2 int* int_ptr = &a;
3
4 cout<<"          a = "<<a<<endl;
5 cout<<"Address of a = "<<&a<<endl;
6 cout<<"      int_ptr = "<<int_ptr<<endl;
```

7.3 Using * operator

“Notice how the asterisk is used on line 2. The variable `int_ptr` has the type pointer to integer or pointer to int. The address of the variable `a` is assigned to `int_ptr` with the help of the `&` operator. When this code is compiled and run it produces the results shown in figure 7-7. Remember, Perplexed One, if you run this program at home you will most likely see a different address than the one shown here. The actual value of the address depends on where in memory the program is loaded, which depends in turn on how much memory your computer has installed and what programs are running at the same time.”

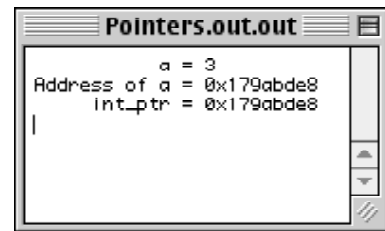


Figure 7-7: Contents of `int_ptr`

C++ Man let Perplexed One think about what had been discussed so far for a moment and then continued.

“The placement of the asterisk is a personal matter. Some programmers will do it differently than others. Line 2 of example 7.3 could be written in any of the following ways with the same results:”

```
int* int_ptr = &a;
int * int_ptr = &a;
int *int_ptr = &a;
```

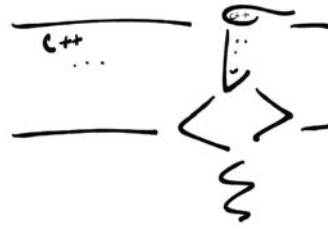
“I like the first method” Perplexed One said, busily scratching notes with his pencil. “It makes it clear to me what’s being done” he added.

C++ Man continued. “It’s also a good idea to choose a name for a pointer variable that gives a hint of its purpose. For instance, the variable `int_ptr` does a good job of indicating that the identifier is a pointer and what type of objects it can point to.”

Perplexed One was writing as fast as he could. He was sure this stuff would be on a test!

“Remember, use the asterisk to declare pointer variables. You can declare a pointer to any type of object, even user-defined objects. You’ll learn how to do that in chapter 10.

You can use the & operator to get the address of an object so it can be assigned to a pointer. Later, I’ll show you how to use the new operator to create objects out of thin air!”



C++ Man continued. “I can see from your hair that you’re beginning to understand pointers much better than before!”

Perplexed One wondered what C++ Man was talking about but was too busy writing notes to check his hair.

“I’m ready to discuss the next question C++ Man” Perplexed One said, pulling out another piece of paper.

How do you ACCESS THE OBJECT A POINTER POINTS TO?

“This is where things get a little tricky” C++ Man started. Perplexed One looked nervous. “Don’t worry, Perplexed One, it will all make perfect sense when we’re done. Remember, a pointer will contain an object’s memory address. You can get to the object itself with the help of the * operator.” Perplexed One’s eyes glazed over.

“This is confusing” Perplexed One said defiantly. “The asterisk is already used to declare pointers.

“You’re right,” C++ Man said, “but I’ll show you how you can easily remember the difference between using the asterisk for declaring pointers and dereferencing them.”

“OK, show me” Perplexed One said, sitting ready to copy notes.

“Examine the following code” C++ Man said as he waived his hand at the board.

```
1 int a = 3;
2 int* int_ptr = &a;
3
4 cout<<"          a = "<<a<<endl;
5 cout<<"Address of a = "<<&a<<endl;
6 cout<<"      int_ptr = "<<int_ptr<<endl;
7 cout<<"          a = "<<*int_ptr<<endl;
```

7.4 dereferencing pointers

“One line 2, the asterisk appears to the right of the data type when you declare a pointer variable. When you use the * operator to dereference a pointer the asterisk appears to the left of the pointer as shown on line 7.”

Perplexed One’s eyes brightened immediately. “That’s all there is to it?”

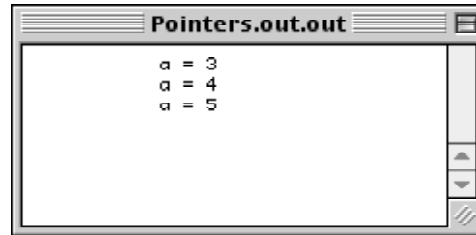
“Yep, that’s about it.” Here’s another example:

```
1 int a = 3;
2 int* int_ptr = &a;
3 cout<<"          a = "<<a<<endl;
4 *int_ptr = 4;
5 cout<<"          a = "<<a<<endl;
6 (*int_ptr)++;
7 cout<<"          a = "<<a<<endl;
```

7.5 dereferencing pointers

Perplexed One studied the code on the board.

“As you can see,” C++ Man began, “the variable `a` is initialized to 3 on line 1. On line 2 the variable `int_ptr` is declared and initialized to the address of `a` using the `&` operator. From that point on, all modifications to variable `a` can be effected via the pointer with the help of the `*` operator. Notice how the `*` operator appears on the left side of the pointer variable. On line 4 the pointer is used to change the value of `a` from 3 to 4. Line 6 shows how parentheses can be used along with the `*` operator to help force operator precedence. Running example 7.5 produces the results shown in figure 7-8.”



```

Pointers.out.out
a = 3
a = 4
a = 5

```

Figure 7-8: Running Example 7.5



When the asterisk appears to the right of a data type it is being used to declare a pointer variable. When the asterisk is applied to the left side of a pointer it is called dereferencing and is the way you access the object the pointer is pointing to.

“That seems too easy, there must be a catch.” Perplexed One stated confidently.
 “Nope,” replied C++ Man. “there’s no catch. It’s just spine-tingling isn’t it?”
 “Isn’t what?” Perplexed One asked, looking more perplexed.
 “Why, all the fun we’re having with pointers! Prepare for the next topic.”



How do you DYNAMICALLY CREATE AND DELETE objects?

“Now you’re really going to learn some cool stuff!” C++ Man began. Perplexed One was ready and felt confident he’d understood everything up to this point, and they were over half way through the list of questions. “So far so good” he thought to himself, pencil poised to scribble with wild abandon. C++ Man began again.

“Up to this point you’ve seen how the address of a statically allocated object can be determined with the `&` operator. You’ve seen how pointer variables can be declared and used to modify the object they point to. The real power of pointers comes from being able to dynamically create objects when a program is running, also referred to as run time or dynamic object allocation. Before I talk about the use of the `new` operator I want to discuss the two different kinds of memory a program has access to, namely, the stack and the heap.”

“You’ve seen stack memory in action in listing 7-1. The program in example 7.2 used stack memory to store the value of the variable `a`. Stack memory works well in this case because all the program’s storage needs were determined at compile time.”

“When a program does not know up front how much storage it needs it must dynamically allocate memory during run time from an area of memory known as the heap. The heap is managed by the operating system. A program makes requests to the operating system for memory space and, if there is enough space available on the heap, the memory is reserved and an address is returned to the program.”

“Figure 7-9 shows where stack and heap memory are located with respect to one another in a computer’s memory space.”

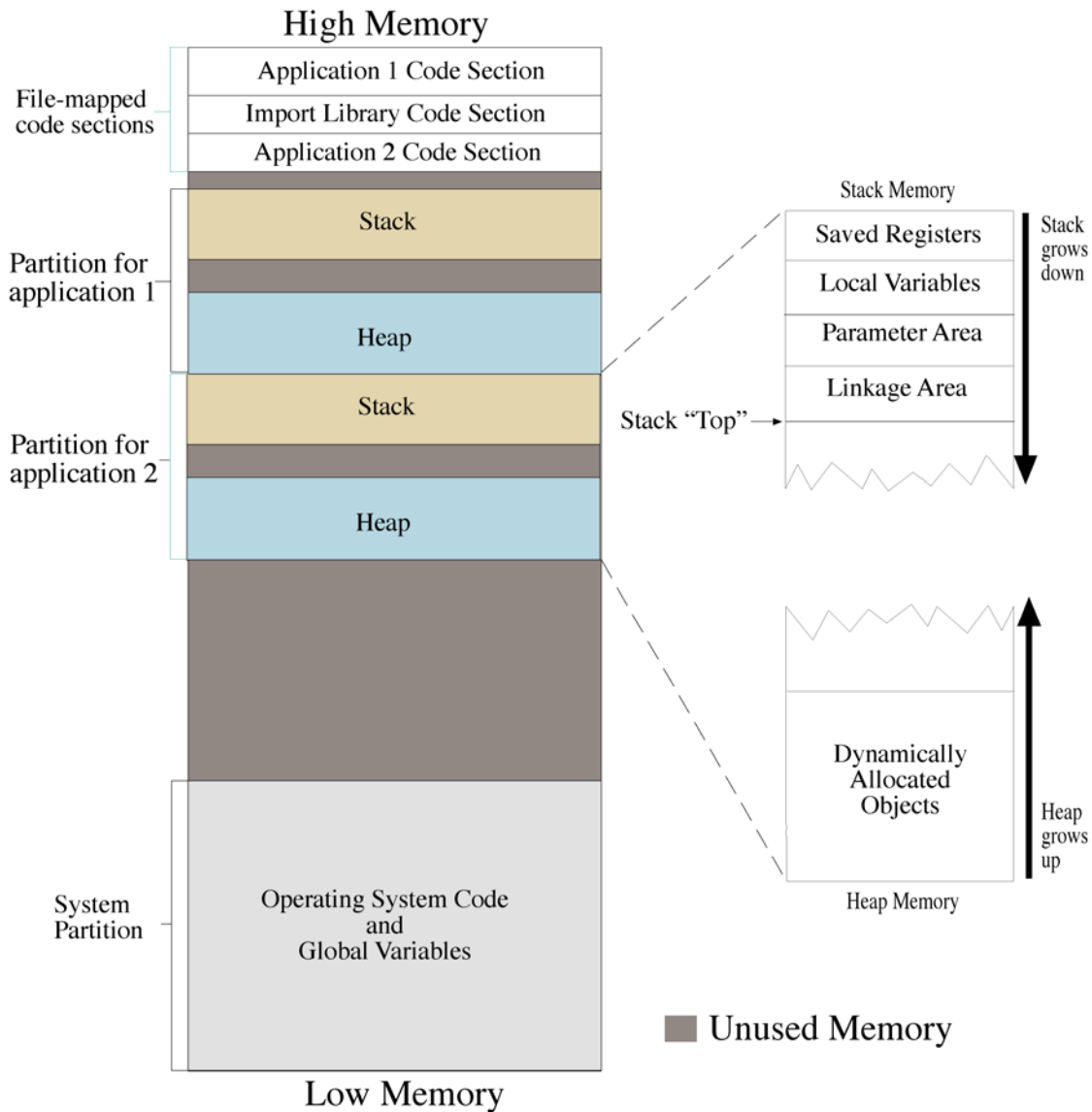


Figure 7-9: Application Stack and Heap Relationship

When the operating system loads a program it sets up an application partition. The partition will be organized into two sections, namely, the stack and the heap. During program execution the stack will grow downwards as functions are called and upwards as functions return to the calling routines. The heap will grow upwards and downwards as dynamically allocated objects are created and destroyed. When an application requests dynamically allocated memory it is carved out of its application partition’s heap section.”

THE NEW OPERATOR

“The new operator is used to dynamically allocate objects during program runtime. The new operator makes a request for heap memory space to the operating system and, if memory is available, will return the newly created object’s memory address. In the case of complex objects like structures and classes, the address returned will be the start of the object’s memory region.”

DIFFERENCE BETWEEN NEW AND NEW[]

“There are two types of new operator: new, and new[]. The new operator is used to create singular object instances. The new[] operator is used to create arrays of objects.” (*The new[] operator is discussed in detail in chapter 8.*)

FORMAT OF THE NEW OPERATOR

“The new operator takes the following format:”

Returns memory address
of newly created object

→ **new type;** ←

Type can be native
or user-defined

USING THE NEW OPERATOR

C++ Man waived his hand at the board and the following source code appeared:

```

1 int* int_ptr;
2 int_ptr = new int;
3 *int_ptr = 0;
4
5 cout<<"The reserved address is:      "<<int_ptr<<endl;
6 cout<<"The value of the int object is: "<<*int_ptr<<endl;
7
8 delete int_ptr;
```

7.6 dynamic memory
allocation

Perplexed One studied the code intensely and quickly copied it into his notebook. C++ Man began his explanation. “On line 1 I declared a pointer variable named int_ptr to hold the address returned by the new operator. On line 2 the new operator is used to create an object of type int. The address returned is assigned to int_ptr, which is then used on line 3 to access the integer object itself.”



C++ Man summarized, “Dynamic object allocation is a way to create objects in memory at program runtime. The new operator will request memory from the operating system and return a memory address you can use in your program. Dynamic objects are created in an application’s heap memory.”

“What does line 8 in example 7.6 do?” Perplexed One asked, looking up from his notes.

“Good question Perplexed One” he began. “When you’re finished with a dynamically allocated object you must remember to release the memory back to the operating system. Failure to do so will result in a memory leak.”

Perplexed One liked the sound of the term memory leak. He thought it was cool.



The Difference Between delete and delete[]

C++ Man pointed once again to the code on line 8 in example 7.6. “There are two types of delete operators” he began. “The delete operator, shown here, is used with the new operator, and is used to release the memory assigned to singular objects back to the operating system.”

“The delete[] operator is used to delete memory dynamically assigned to an array via the new[] operator. A detailed discussion on the delete[] appears in chapter 8.”

A NEAT TRICK: CALLING OBJECT CONSTRUCTORS

“I want to show you something you’ll find helpful when using the new operator” C++ Man told Perplexed One. Perplexed One looked at the board just as C++ Man was waving his hand, causing the following code to appear:

```

1 int* int_ptr;
2 int_ptr = new int(0);    //Constructor call
3
4 cout<<"The reserved address is:      "<<int_ptr<<endl;
5 cout<<"The value of the int object is: "<<*int_ptr<<endl;
6
7 delete int_ptr;

```

7.7 calling object constructor

“Study the difference between example 7.6 and 7.7. In example 7.6 the object was initialized via the pointer after it was created in memory. In example 7.7 the object is initialized when it is created via a constructor function call. A constructor is a special function whose purpose is to properly initialize objects when they are created in memory. The integer constructor is invoked by enclosing an integer value in parentheses after the type name int when used with the new operator as shown on line 2 above. Other fundamental data types can be initialized in similar fashion.” (Constructors are formally covered in chapters 10 and 11.)

“Here’s another example Perplexed One” C++ Man said, looking toward the board.

```

1 int* int_ptr = new int(7);
2 char* char_ptr = new char('e');
3
4 cout<<" The int value is: "<<*int_ptr<<endl;
5 cout<<"The char value is: "<<*char_ptr<<endl;
6
7 delete int_ptr;
8 delete char_ptr;

```

7.8 calling object constructor

WHAT’S THE DIFFERENCE BETWEEN A POINTER AND A REFERENCE?

Perplexed One was eager to continue the lesson and started a new sheet of notes. C++ Man waived his hand at the board and removed all the code so there would be nothing to confuse Perplexed One.

“The basic thing to remember about a pointer is that it is a variable. Because a pointer is a variable, it can be changed to point to different objects of the same type.”

“A reference is different from a pointer in that once a reference is initialized it cannot be changed. Another good way of thinking about a reference is that it is a nickname for the object it references. A nickname is just another name for the same object. Pointers must be dereferenced to gain access to the object they point to. Not so with references.”

C++ Man waived his hand at the board and table 7-1 appeared. Perplexed One took notes furiously.

Characteristic	Pointers	References
Variable	Yes	No
Must be initialized at the point of declaration	No	Yes
Must be dereferenced with the * operator	Yes	No
Can be thought of as another name for the referenced object	No	Yes
Can be used in place of pointers to reduce errors and simplify code	No	Yes

Table 7-1: Pointers VS. References

How do you DECLARE AND USE REFERENCES?

Perplexed One looked worried. C++ Man asked him what was the matter. “This is where things get tricky” Perplexed One said, fidgeting at his desk.

“Fear not, Perplexed One,” C++ man said assuringly, “all you have to remember is when to use the & operator. Can you tell me, from what you’ve learned so far, when you would use the & operator?”

Perplexed One studied his notes and answered timidly. “When you want to find an object’s memory address?”

“That’s exactly right!” C++ Man said, flying a victory circle in front of the room. “Now there’s another use for the &. Tell me Perplexed One, how do you declare a pointer variable?”

“You have to use the asterisk between the data type and the identifier.” Perplexed One answered, this time with more confidence.

“Right again!” C++ Man said, waving his hand at the board. “Take a look at this code:”

```

1 int a = 3;
2 int& ref_a = a;
3
4 cout<<"    a = "<<a<<endl;
5 cout<<"ref_a = "<<ref_a<<endl;
```

7.9 using a reference

“Notice how the & is used on line 2 to declare a reference type. The identifier ref_a can now be used in place of the identifier a without any dereferencing, as is shown on line 5.”

“References come in handy when you’re passing function parameters by reference or returning objects by reference from functions. Functions will be discussed in detail in chapter 9.”

Perplexed One felt relieved. “That doesn’t seem so hard!” he said, finishing his notes.

C++ Man hovered in front of the class. “Your basic understanding of pointers and references will serve you well as you progress through your studies of C++ Perplexed One. Why don’t you summarize all your pointer knowledge before I have to fly and help another perplexed student.”

SUMMARY

Perplexed One gathered his notes and walked to the front of the class. C++ Man took a seat in the front row and listened attentively as Perplexed One reviewed what he had learned.

“An object is a region of memory. An object can be a fundamental data type like a character or integer, or a complex user-defined type. An object can be created in three ways: 1) by definition, 2) by using the new or new[] operator, or 3) as required by the compiler.”

“All objects in memory can be accessed via a memory address. To determine the memory address of an object use the & operator.”

“A pointer is a variable that holds a memory address. Because a pointer is a variable, the address it contains can

be changed, causing it to point to something else. Use the asterisk to declare a pointer and the asterisk to access the object a pointer points to. This is also referred to as dereferencing the pointer.”

“Dynamic object allocation allows programs to create objects at program runtime. Dynamically allocated objects are created on the application heap. Use the new operator to create dynamic objects, and don’t forget to deallocate objects when you no longer need them with the delete operator. Constructors can be used during dynamic object allocation to initialize objects in memory.”

“References differ from pointers in several ways. References must be initialized when they are declared because they are not variables. A reference is considered another name for a particular object. Use the & to declare references. The cool thing about references is that they don’t need to be dereferenced.”

Skill Building Exercises

1. **Research Memory Organization:** Research the architecture of your computer. Write a brief description of how the hardware and operating system work together to manage memory resources.
2. **Determine Object Address Using & Operator:** Write a program that creates three variables with different data types and assign each a value. Use the & operator to determine the address of each variable. Print the value of each variable and its memory address to the screen.
3. **Dissemble Code:** Disassemble the program you wrote in exercise 2 above and study the output. Examine the listing to determine how and where the variables are being stored in memory.
4. **Declare and Initialize Pointers:** Write a program that declares a char, int, float, and double variable. Initialize each variable to a value of your choice. Next, declare four pointers, one for each data type char, int, float, and double. Assign the address of each object to the proper corresponding pointer variable using the & operator. Print the value of each object to the screen using the variable name, and the dereferenced pointer. (Remember to use the * operator to dereference each pointer.)
5. **Modify Objects Via Pointer:** Using the code you wrote in exercise 4, change the value of each object via the pointer. Print the new values to the screen.
6. **Dynamically Create and Destroy Objects:** Write a program that declares four pointer variables, one for each data type char, int, float, and double. Use the new operator to dynamically create an object of each type and assign its address to the corresponding pointer variable. Use the pointer to initialize each object and print their values to the screen. Use the delete operator to release the dynamically allocated memory back to the operating system.
7. **References:** Write a program and declare and initialize three variables of any type and value your choose. Declare three references of the required type and initialize them using the three variables previously declared. Access and modify each of the three objects via the references. Print the object values to the screen using the references.
8. **Thinking:** Describe in your own words the differences between pointers and references.

Suggested Projects

1. **Dynamic Object Creation:** Write a program that dynamically creates integer objects. Declare five integer pointers. Prompt the user to enter integer values via the keyboard. Read the integer values and use them to initialize the integer objects. Print the object values to the screen via the pointers.

SELF TEST QUESTIONS

1. What is an object? What is the difference between an object and the identifier name used to reference the object?
2. Describe in general terms how computer memory is typically organized.
3. What operator do you use to determine an object's memory address? Give an example of its use.
4. What is a pointer?
5. What character do you use to declare a pointer? Give several examples of its use.
6. What operator do you use to access an object via a pointer? Give an example of its use.
7. What character is used to declare a reference?
8. What are the primary differences between a pointer and a reference?
9. How do you access the object referred to by a reference? How is this different from accessing an object via a pointer?
10. (T/F) A pointer is a variable.

REFERENCES

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Robert J. Traister. *Conquering C++ Pointers*. Academic Press Professional, Inc., Harcourt Brace & Company, Publishers, Boston, Massachusetts, 1994. ISBN: 0-12-697420-9

Paul J. Lucas. *The C++ Programmer's Handbook*. Prentice Hall PTR, Englewood Cliffs, New Jersey, 1992. ISBN: 0-13-118233-1

NOTES

CHAPTER 8



RACE DAY Waikiki BEACH

ARRAYS

LEARNING OBJECTIVES

- Describe the concept of an array
- State the purpose and use of single- and multi-dimensional arrays
- Describe how to declare and initialize single- and multi-dimensional arrays
- Explain how the compiler uses the array's declared type to calculate offset addresses into an array
- Explain how to access array elements using array subscript notation and pointer notation
- List and describe the similarities between an array name and a pointer
- Explain how to use pointers to dynamically allocate memory for an array with the `new[]` operator
- Explain how to release dynamically allocated array memory with the `delete[]` operator
- Explain how to idiomatically process an array using a for loop
- Explain how to process multi-dimensional arrays using nested for loops
- Explain how strings are implemented in C++
- Utilize single- and multi-dimensional arrays in your C++ programming projects

INTRODUCTION

Arrays are the workhorses of data structures. An understanding of arrays, their declaration, their use, and their manipulation, presents a wealth of programming possibilities. No longer will you be limited to modeling data as individual objects; with arrays you will have the power to manipulate collections of objects.

The primary focus of this chapter is to give you a thorough understanding of array basics. You will learn what arrays are, how they are represented in memory, how to declare single- and multi-dimensional arrays, and how to manipulate them. The material presented here builds on what you learned in chapters 5 through 7. You will continue to expand your knowledge of pointers by learning how to create arrays dynamically using the `new[]` operator. The secondary focus of this chapter is to give you a good understanding of when to use an array — and when not to.

Since a formal discussion of user-defined data types occurs later in chapter 10, the examples in this chapter will use arrays of fundamental data types such as `char`, `int`, or `float` only. I took this approach to keep the discussion focused on the topic of arrays, however, the principles learned here will apply to arrays of user-defined data types as well.

WHAT IS AN ARRAY?

An array is a contiguous allocation of memory to homogeneous objects. Contiguous means the objects are stored in memory one after the other. Homogeneous means the objects in the array are of the same type. Figure 8-1 shows an array of 4 integer objects.

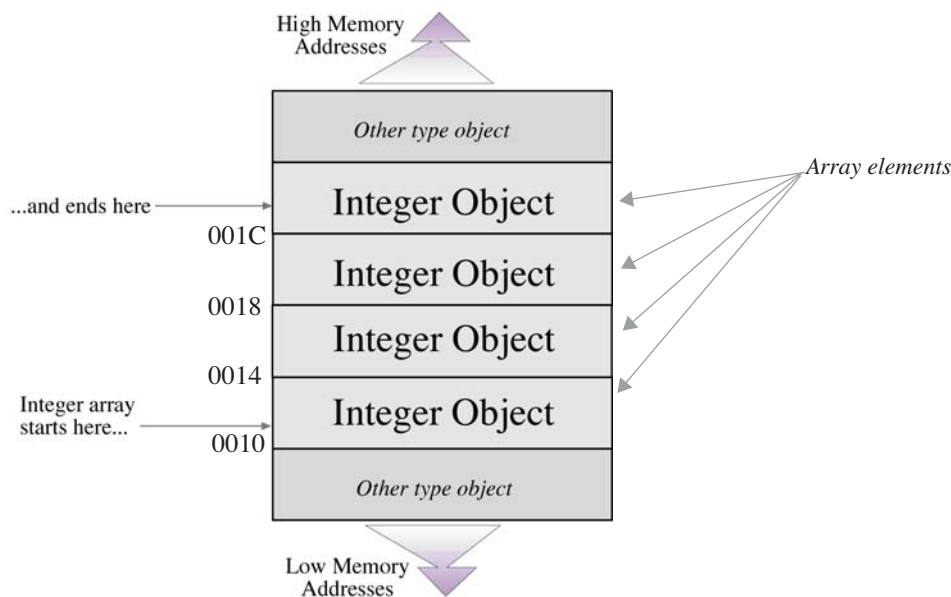


Figure 8-1: Array of Four Integer Objects

The array begins at a certain memory address. Each integer object in the array follows the one before it and is located at a memory address one allocation unit higher than the previous object in the array. In the case of integers, an allocation unit is equal to four bytes on a 32-bit machine, since an integer occupies four bytes of memory. As an example, if the integer array shown in figure 8-1 starts at memory address 0010, then the next integer object will be located at memory address 0014, the next at memory address 0018, and the last at memory address 001C.

LOCATING ARRAY ELEMENTS

Each object in an array is referred to as an array element. Individual array elements are accessed by combining the starting address of the array with a memory address offset that points to the beginning of the particular array element. The offset address is calculated by multiplying the size of an array object in bytes by the number of the element to be accessed minus 1. For instance, to access the third array element in the integer array shown in figure 8-1, the address of the beginning of the array, *0010*, is added to the offset, $(4 \text{ bytes} \times (3 - 1)) = 8$, to yield an element address of 0018. Thus, the third element of the array is located at memory address 0018.

You may be wondering why you need to subtract one from the element number. Since the first element of the array is located at the start of the array, no offset is required to access the first element. In other words, the address of the start of the array points to the first element of the array, the second element of the array is located one allocation unit from the start of the array, the third element is located two allocation units from the start of the array, and so on. Remember, the size of the allocation unit used to calculate array element offsets depends on the type of objects the array is declared to contain.

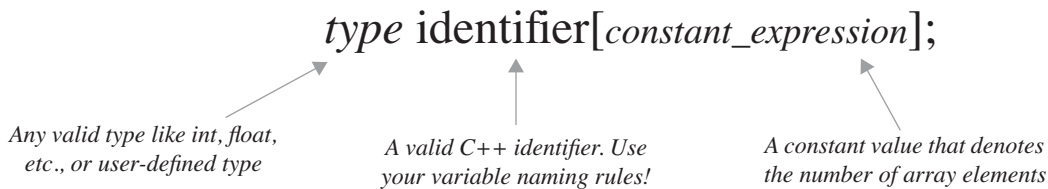
Fortunately, C++ provides a convenient notation for accessing array elements that is discussed in the next section.

DECLARING & DEFINING STATICALLY ALLOCATED ARRAYS

Statically allocated arrays, like statically allocated variables, exist in stack memory as opposed to dynamically allocated arrays which are located in memory allocated from the application heap. This section will discuss the declaration and initialization of static arrays of both single and multiple dimensions.

SINGLE-DIMENSIONAL ARRAYS

Single-dimensional arrays, like the one shown in figure 8-1, are declared using the following format:



The type of the array denotes what kind of objects the array will contain. Arrays can be constructed from the following types:

- fundamental types such as `int`, `float`, `char`, *etc.*
- user-defined structures and classes
- pointers to fundamental types, user-defined types, and functions
- pointer to member
- enumeration types
- other arrays

This information is used by the compiler to determine the size of each element. The identifier is the name of the array. Name your arrays like you name your variables – something that makes sense! Here are a few examples:

```
int number_list[3];           //array of 3 integer objects
float account_balances[25];  //array of 25 float objects
char search_string[200];     //array of 200 char objects
double* double_pointers[10]; //array of 10 pointers to double objects
```

This last example is interesting in that the array of pointers itself is statically allocated, but the double objects each pointer ultimately points to can be dynamically allocated using the new operator you learned about in chapter 7. You will see a more detailed example of this later.

The examples above use integer literals as the constant expression to denote the size of each array. You can also use a named constant as the following example illustrates:

```
const int ARRAY_SIZE = 25;
int integer_array[ARRAY_SIZE]; //array of 25 integer objects
```

ACCESSING ARRAY ELEMENTS

There are two primary methods you can use to access the objects stored in an array. The first, and most common method, utilizes the array name and the subscript operator []. The second method, pointer arithmetic, treats the array name like a pointer and uses the pointer dereference operator *. Let us look closer at the subscript method.

Subscript Method

The subscript operator is overloaded in C++ meaning it is used both to declare arrays, as shown in the previous section, and to access individual array elements. To access the objects stored in an array, apply the subscript operator to the name of the array, enclosing between the brackets the element number you wish to access minus one. (Remember, the 1st element in an array is the array name plus a zero offset) The following code declares an array of integers and sets each element to a unique integer value using the subscript method:

```
1  int int_array[5];
2
3  int_array[0] = 1;
4  int_array[1] = 2;
5  int_array[2] = 3;
6  int_array[3] = 4;
7  int_array[4] = 5;
```

*8.1 declaring & using
integer array*

This code declares a five element array of integers named int_array. Lines 3 through 7 sets each integer object to the value specified. The following example prints the contents of int_array to the console:

```
1  cout<<int_array[0]<<endl;
2  cout<<int_array[1]<<endl;
3  cout<<int_array[2]<<endl;
4  cout<<int_array[3]<<endl;
5  cout<<int_array[4]<<endl;
```

8.2 using integer array

The number appearing between the subscript brackets is referred to as the array index or array subscript. It must be an integer type and can be either a literal, as is used above, or a variable. The following code sets the values of int_array using a for loop and then prints the values to the console with another:

```
1  int int_array[5];
2
3  for(int i = 0; i<5; i++)
4      int_array[i] = i+1;
5
6  for(int i = 0; i<5; i++)
7      cout<<int_array[i]<<endl;
```

*8.3 manipulating array
with for statement*

POINTER ARITHMETIC METHOD

The subscripted array name returns the object located at that memory location. The subscripted array name is exactly equivalent to the array name plus the index dereferenced with the * operator. For example, `int_array[1]` returns the same element as `*(int_array + 1)`. Look at example 8.4.

```

1 int int_array[5];
2
3 for(int i = 0; i<5; i++)
4     *(int_array + i) = i+1;
5
6 for(int i = 0; i<5; i++)
7     cout<<*(int_array + i)<<endl;

```

8.4 pointer arithmetic

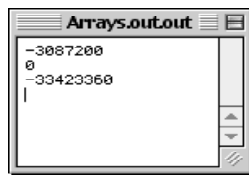
BEWARE THE UNINITIALIZED ARRAY!

It is important not to make any assumptions about the state of each object in a newly declared array. For example, consider for a moment what the value is of each array element in the following example:

```
int number_list[3]; //what's the value of each integer element?
```

If you guessed garbage you're right. Examine the code in example 8.5 and the result it produces. Keep in mind that running these examples on your computer will yield different garbage values.

8.5 garbage out



```

1 int number_list[3];
2
3 for(int i=0; i<3; i++)
4     cout<<number_list[i]<<endl;

```

Figure 8-2: Results of Running Example 8.5

Notice in figure 8-2 the values contained in the first and third elements of the array. The values represent the random state of the memory region in which the array is created. One way to ensure the values of array elements are initialized properly is to combine the definition of each array element with the array's declaration.

COMBINING ARRAY DEFINITION WITH ARRAY DECLARATION

You can define each element of an array at the same moment you declare it by enclosing element initializers in braces following the declaration as is shown in the following examples:

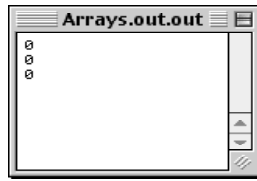
```

int number_list[3] = {1,2,3}; //array of 3 ints
int number_list[] = {1,2,3,4,5}; //array of 5 ints
int number_list[10] = {}; //array of 10 ints initialized to 0
int number_list[10] = {3}; //first element 3, the rest 0

```

Notice in the second example the constant expression between the brackets is missing. Defining an array in this manner results in an array with the same number of elements as there are initializers. In the third example, the ten element array is initialized to all zeros, whereas in the fourth example, the first element is initialized to three and the rest of the elements are initialized to zero.

Now observe the behavior of the following code:



8.6 good output

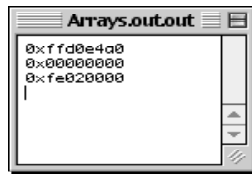
```
1 int number_list[3] = {};
2
3 for(int i=0; i<3; i++)
4     cout<<number_list[i]<<endl;
```

Figure 8-3: Results of Running Example 8.6

All the elements of `number_list` are initialized to zero.

ARRAYS OF POINTERS

An array of uninitialized pointers behaves like an array of uninitialized integers when declared in that each pointer will, unless properly initialized, contain a garbage value. Observe the behavior of the code shown in example 8.7.



8.7 uninitialized pointers

```
1 int *int_pointers[3];
2
3 for(int i=0; i<3; i++)
4     cout<<int_pointers[i]<<endl;
```

Figure 8-4: Results of Running Example 8.7

To initialize the pointers to null you can use `NULL` in the initializer list as shown in example 8-4 or you can simply use an empty initializer list as was used in example 8.6.



8.8 good output

```
1 int *int_pointers[3] = {NULL};
2
3 for(int i=0; i<3; i++)
4     cout<<int_pointers[i]<<endl;
```

Figure 8-5: Results of Running Example 8.8

It is always a good idea to set pointers to `NULL` if they aren't pointing to anything in particular.

Now that you have an array of pointers you need to create the object the pointers will ultimately point to. Figure 8-6 gives a visual idea of what needs to happen. Integer objects will be dynamically created in heap memory using the new operator and the resulting memory address assigned to an array element.

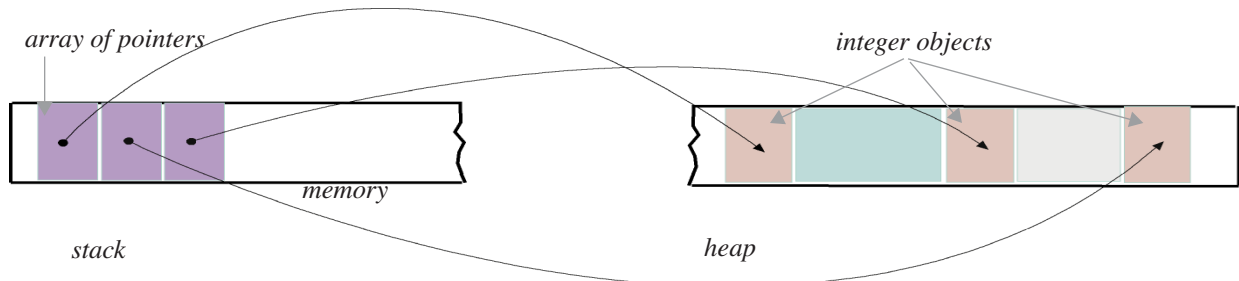


Figure 8-6: Array of Integer Pointers and Dynamically Created Integer Objects in Heap Memory

The following code declares the array of integer pointers initialized to `NULL`. It then creates three integer objects and assigns the address of each to an array element:

8.9 pointer NULL initialization

```

1 int *int_pointers[3] = {NULL};
2
3 for(int i = 0; i<3; i++)
4     int_pointers[i] = new int(i+1);

```

Note the use of the new operator to create the integer objects. The value of each integer is created by calling its constructor using the value of the for loop index value plus one.

The following code prints the values of the integer objects created above:

8.10 using delete operator on pointer array elements

```

1 for(int i = 0; i<3; i++)
2     cout<<*int_pointers[i]<<endl;
3
4 for(int i = 0; i<3; i++)
5     delete int_pointers[i]; //use delete to release memory

```

Notice how the pointer dereference operator * is used with the array to access the actual object pointed to by each array element. Just as with ordinary pointers, you need to ensure you release the memory reserved on the heap back to the operating system using the delete operator.

Multi-Dimensional Arrays

An array of more than one dimension is actually an array of arrays. This section will discuss two-dimensional arrays followed by arrays of three and more dimensions.

ARRAYS of Two DIMENSIONS

The hardest part of working with multi-dimensional arrays is deciding exactly what each dimension represents, picturing the array representation in your mind, and then translating that understanding into an array declaration. Let us pause for a moment to consider once again the single-dimensional array. Figure 8-7 offers a visual representation of a single-dimensional array of 5 integer objects.

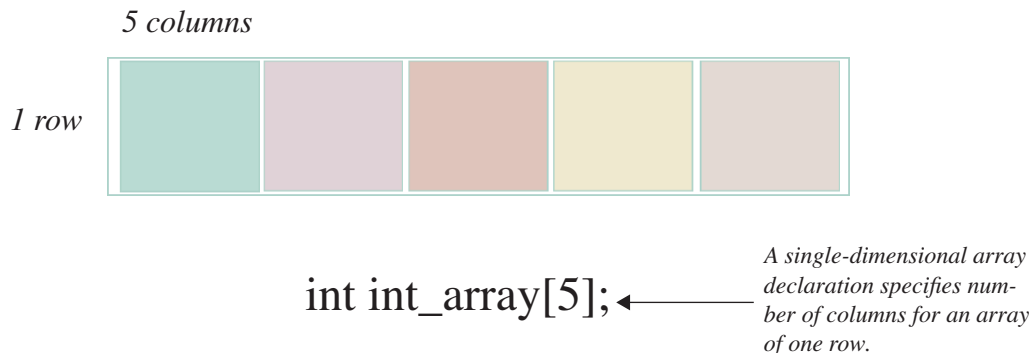


Figure 8-7: Single-Dimensional Array Representation and Declaration

You can safely think of a single-dimensional array as having one row by default and that the declaration's purpose is to establish how many elements, or columns, the row will contain. Thinking in this manner will help you visualize not only single-dimensional arrays, but arrays of two, three or more dimensions.

The declaration of a two-dimensional array looks similar to that of a single-dimensional array with the addition of another set of brackets indicating the number of rows the array will contain. Check out the following examples:

```
int integer_grid[5][5]; // a 5 x 5 integer array
float float_matrix[10][10]; //a 10 x 10 float array
```

But which number in the above declarations represents the columns and which one represents the rows? C++ stores arrays in row-major order meaning in memory, a row of elements exists, followed by the next row with its elements, and so on for each row. Named constants can be used in place of the integer literals for the constant expression and this will clarify things a little as shown in the following example:

```
const int ROWS = 5;
const int COLS = 5;

int integer_grid[ROWS][COLS];
```

Notice how ROWS comes before COLS. There is a pattern here. If you think of array dimensions as groupings or sets you will see that the smallest set of objects, which is ultimately the object itself, always moves to the right in the array declaration. Since, in a single-dimensional array, the array itself can be thought of as a grouping of elements in one row, a two-dimensional array can be thought of as a grouping of rows that each contain a grouping of elements. In other words, an array of arrays. Let us take a look at a visual representation of a two-dimensional array.

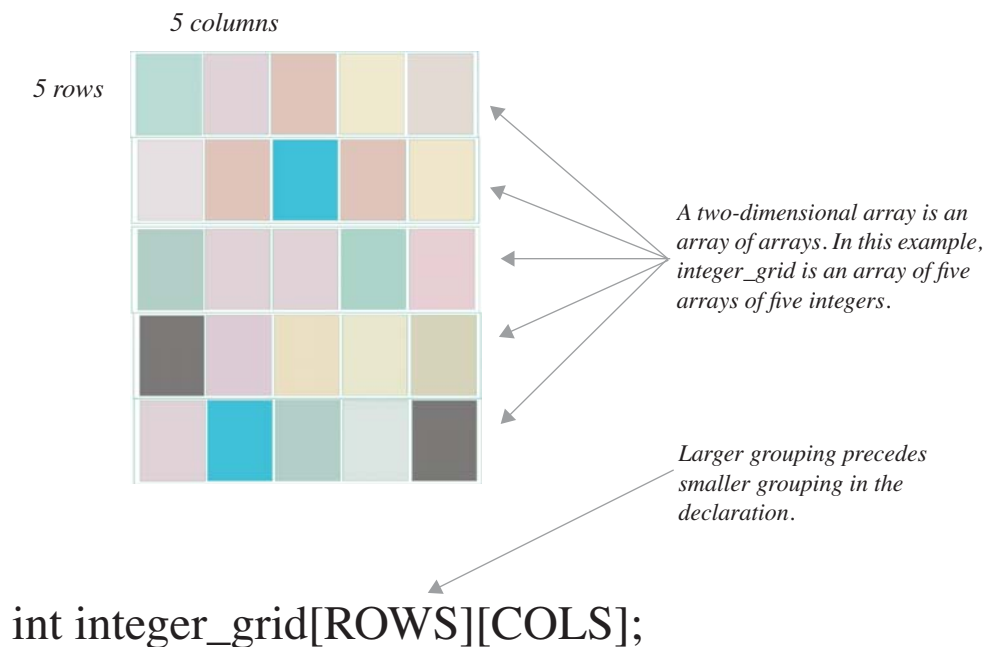


Figure 8-8: Two-Dimensional Array and Declaration

Figure 8-9 shows how the two-dimensional array depicted above might be represented in memory.

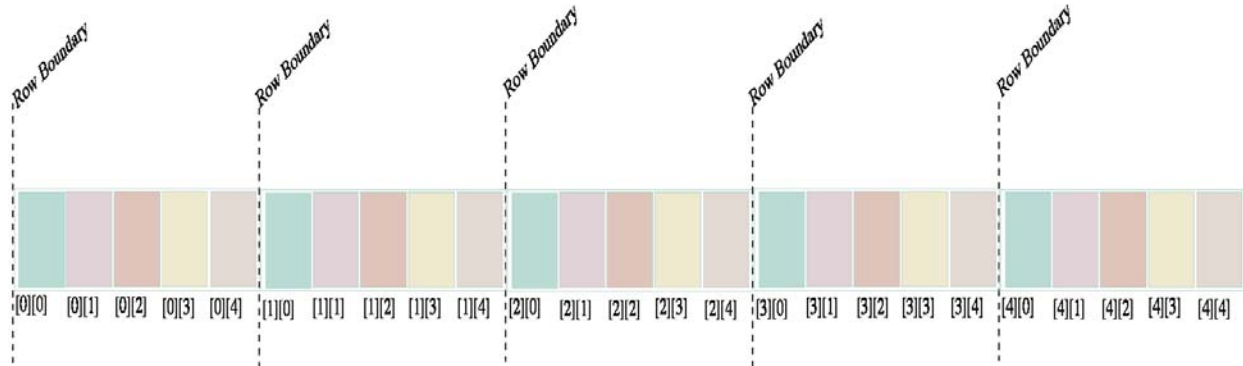


Figure 8-9: Two-Dimension Array Memory Representation

Figure 8-9 shows how C++ arranges two-dimensional arrays in memory in row-major order. Row-major order is an arrangement where a row and all its elements are stored followed by the next row and all its elements and so forth for the entire array.

ARRAYS OF THREE OR MORE DIMENSIONS

Keeping in mind that in C++ multi-dimensional arrays are simply arrays of arrays helps tame the complexity when thinking of arrays of three or more dimensions. The following code declares a three-dimensional array of integers:

```
int integer_cube[5][5][5];
```

Using integer constants to name each dimension results in a clarification of the intended use of each dimension:

```
const int DEPTH = 5;
const int HEIGHT = 5;
const int WIDTH = 5;

int integer_cube[DEPTH][HEIGHT][WIDTH];
```

Another programmer might choose different constant names to reflect a different intended use for the array:

```
const int SHEETS= 5;
const int ROWS= 5;
const int COLUMNS= 5;

int integer_cube[SHEETS][ROWS][COLUMNS];
```

Figure 8-10 shows one possible visual representation of a three-dimensional array.

Just as with two-dimensional arrays, the larger dimensional grouping displaces the smaller groupings to the right, so the [SHEETS] dimension is declared before [ROWS], which is declared before [COLUMNS]. A drawing of the memory representation of a three-dimensional array is left as a skill building exercise.

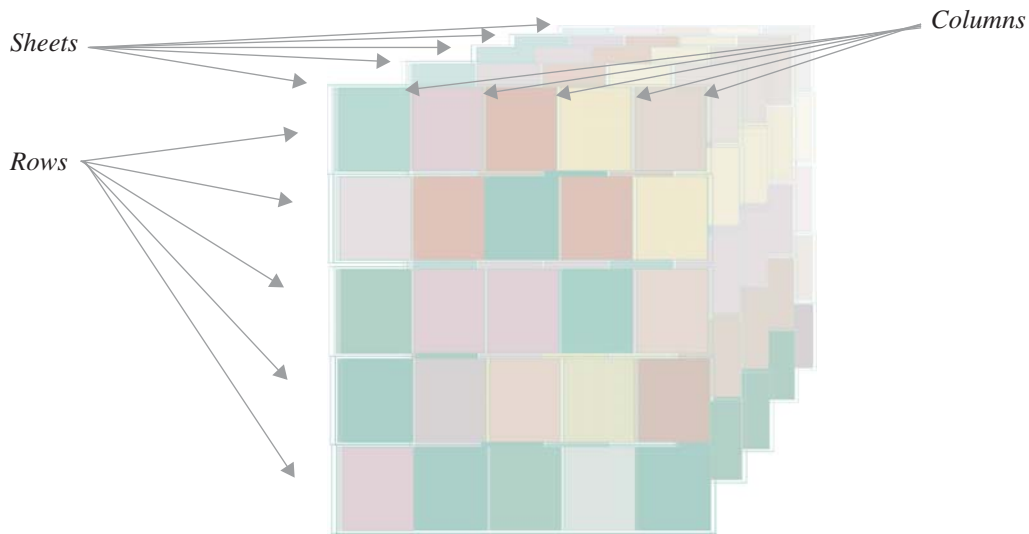


Figure 8-10: Visual Representation of a Three Dimensional Array

Although most of your programming needs can be satisfied using either one-, two-, or three-dimensional arrays, sometimes arrays with more than three dimensions are required. The following code declares an array of four dimensions:

```
const int BOOKS = 2;
const int SHEETS = 5;
const int ROWS = 5;
const int COLUMNS = 5;

int four_d_int_array[BOOKS][SHEETS][ROWS][COLUMNS];
```

Figure 8-11 offers a visual representation of this array.

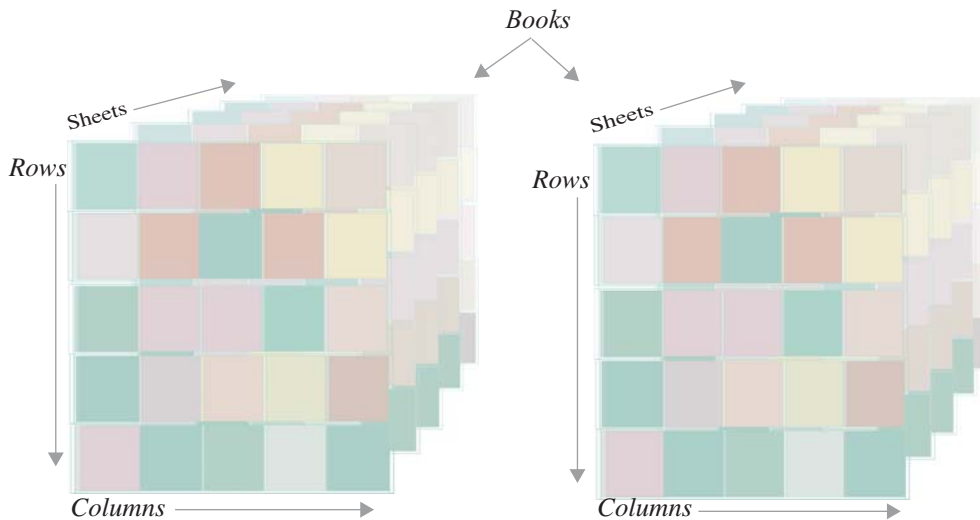


Figure 8-11: Visual Representation of a Four-Dimensional Array

AUTOMATIC INITIALIZATION OF MULTI-DIMENSIONAL ARRAYS

Multi-dimensional arrays can be initialized at the point of declaration like their single-dimensional counterparts. Example 8.11 demonstrates the declaration and initialization of a three-dimensional array:

```

1 const int SHEETS    = 5;
2 const int ROWS      = 5;
3 const int COLUMNS  = 5;
4
5 int three_d_int_array[SHEETS][ROWS][COLUMNS] = {};
6
7 for(int i = 0; i<SHEETS; i++){
8     for(int j = 0; j<ROWS; j++){
9         for(int k = 0; k< COLUMNS; k++){
10            cout<<three_d_int_array[i][j][k];
11            }
12            cout<<endl;
13        }
14        cout<<endl;
15    }

```

8.11 using 3-dimensional array

Figure 8-12 shows the results of running example 8.11.

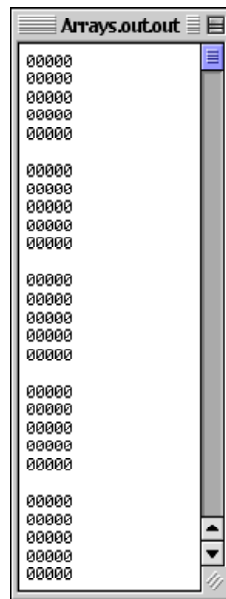


Figure 8-12: three_d_int_array Initialized to Zeros

Using different combinations of braces and integer values results in different possible array initializations. Consider the code shown in example 8.12. This results in all the rows of the first sheet being initialized as shown in figure 8-13. Example 8.13 declares the array and initializes the first row of each sheet. Examine the use of the braces in the declaration and see if you can spot a pattern. Figure 8-14 shows the results of initializing the array in this fashion.

The outermost brace pair represents the array. In the case of three_d_int_array it is an array of five user-conceptualized elements called SHEETS. Each sheet is an array of five user-conceptualized elements called ROWS, and each row is comprised of five integer elements conceptualized as COLUMNS. Figure 8-15 illustrates the relationship between the braces in the array declaration and each of these conceptualizations.

8.12 initializing 3-dimensional array

```

1 int three_d_int_array[SHEETS][ROWS][COLUMNS] = {{{1,2,3,4,5},
2                                                    {1,2,3,4,5},
3                                                    {1,2,3,4,5},
4                                                    {1,2,3,4,5},
5                                                    {1,2,3,4,5}}};
    
```

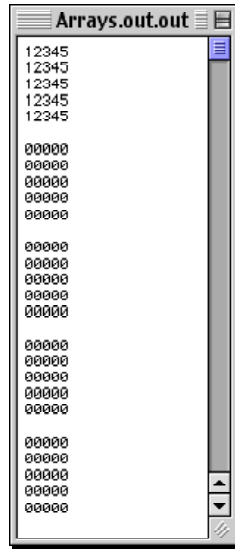


Figure 8-13: All Rows of First Sheet Initialized

```

1 int three_d_int_array[SHEETS][ROWS][COLUMNS] = {{{1,2,3,4,5}},
2                                                    {{1,2,3,4,5}},
3                                                    {{1,2,3,4,5}},
4                                                    {{1,2,3,4,5}},
5                                                    {{1,2,3,4,5}}};
    
```

8.13 brace usage

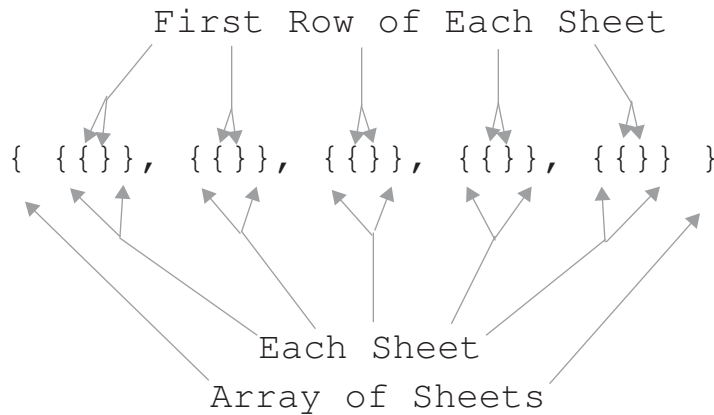


Figure 8-15: Relationship of Declaration Braces to Array Elements for three_d_int_array

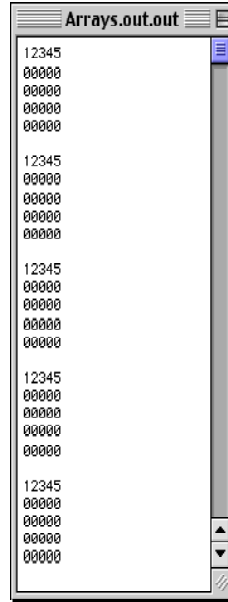


Figure 8-14: First Row of Each Sheet Initialized

Using the brace map shown in figure 8-15 as a guide I'll show you one more initialization scenario. Let us initialize the first three rows of `three_d_int_array` with the values 1,2,3,0,0, and the rest of the rows to all zeros.

```
1 int three_d_int_array[SHEETS][ROWS][COLUMNS] = {{ {1,2,3}, {1,2,3}, {1,2,3} },
2                                                    { {1,2,3}, {1,2,3}, {1,2,3} },
3                                                    { {1,2,3}, {1,2,3}, {1,2,3} },
4                                                    { {1,2,3}, {1,2,3}, {1,2,3} },
5                                                    { {1,2,3}, {1,2,3}, {1,2,3} }};
```

8.14 brace usage

Figure 8-16 shows the results of the initialization shown in example 8.14 above.

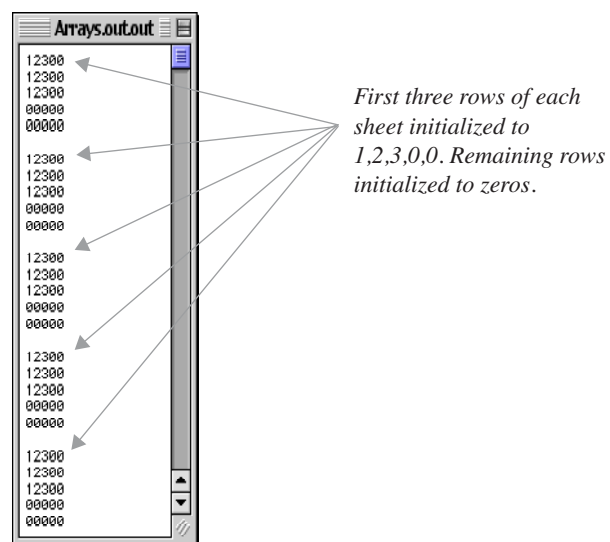


Figure 8-16: Results of Initialization Shown In Example 8.14

DECLARING AND DEFINING DYNAMIC ARRAYS

A dynamic array is one that has been created in application heap memory using the `new[]` operator. The benefit of dynamic array creation is that the array size can be determined at runtime and an array meeting your exact storage needs can be created on the fly. In this section I'll show you how to declare and create dynamic arrays. Once a dynamic array is created its elements are accessed like an ordinary array.

DYNAMICALLY ALLOCATED SINGLE DIMENSIONAL ARRAYS

Figure 8-17 shows a memory representation of a dynamic array of three integer pointers.

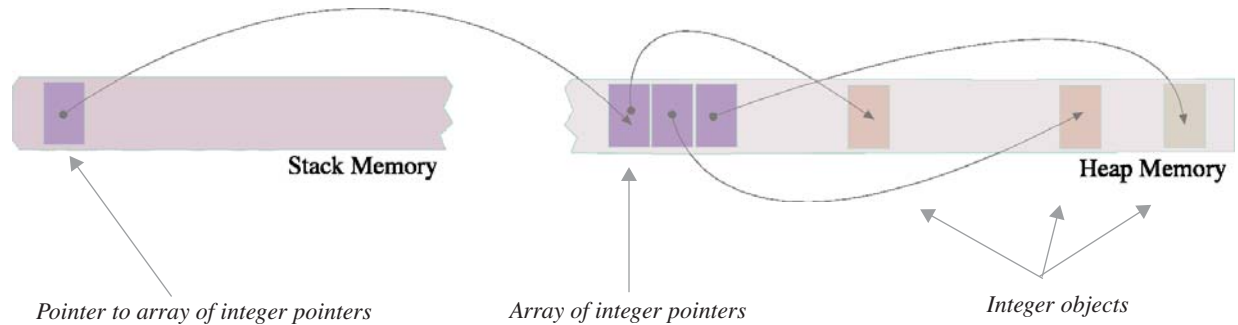


Figure 8-17: Dynamic Array of Three Integer Pointers

Before showing you the code to create the array depicted in figure 8-17 it will help you to know the steps involved with the dynamic array creation process.

You first need to declare a pointer to the type of objects the array will contain. In figure 8-17 the array contains integer pointers so the pointer to the array must be a pointer to a pointer. If, however, you simply wanted an array of integers, you would just need a pointer to an `int`. For now, let us stick with an array of integer pointers. Here is the declaration for the pointer:

```
int** int_pointer_array;
```

The second step is to dynamically allocate the memory space for the array of integer pointers in the heap and assign the address to the pointer. You do this with the `new[]` operator:

```
int_pointer_array = new int*[3];
```

This line allocates memory for an array of three integer pointers in the heap and assigns the address of the first element to the `int_pointer_array` pointer located in the stack.

Once the array of integer pointers is created you can dynamically create each integer object. You will do this with the `new` operator as you've done before with regular pointers.

```
int_pointer_array[0] = new int(1);
```

This line allocates space on the heap for an integer object and assigns its address to the first element of `int_pointer_array`. Notice how ordinary array subscript notation is used on the dynamic array.

When your program no longer needs the dynamically allocated array you must remember to release the memory resources back to the operating system using the `delete` and `delete[]` operators. The following code releases each object pointed to by the elements of `int_pointer_array` and then deletes the array itself:

```

    for(int i=0; i<3; i++){
        delete int_pointer_array[i];
    }
    delete [] int_pointer_array;

```

Example 8.15 shows a complete example:

```

1  int** int_pointer_array = new int*[3];
2
3  for(int i = 0; i<3; i++)
4      int_pointer_array[i] = new int(i+1);
5
6  for(int i = 0; i<3; i++)
7      cout<<*int_pointer_array[i]<<endl;
8
9  for(int i = 0; i<3; i++)
10     delete int_pointer_array[i]; //release memory for each object
11
12 delete[] int_pointer_array; //then release the array

```

8.15 dynamic array allocation

Notice on line 1 how the creation of the dynamic array takes place on the same line as its declaration. On line 7 each integer value is accessed using ordinary pointer dereferencing.

The integer value 3 on line 1 of example 8.15 can be replaced with an integer variable. The variable's value can be set at runtime and used to set the size of the array. Example 8.16 prompts the user to enter a value to be used for dynamic array allocation. This example simply declares an array of integers, not an array of integer pointers, although you can dynamically create an array of any type.

```

1  int array_size = 0;
2  int* int_array = NULL;
3
4  cout<<"Please enter array size: ";
5  cin>>array_size;
6
7  int_array = new int[array_size];
8
9  for(int i=0; i<array_size; i++)
10     int_array[i] = i+1;
11
12 for(int i=0; i<array_size; i++)
13     cout<<int_array[i]<<endl;
14
15 delete[] int_array; //release the array when done

```

8.16 dynamic array allocation

DYNAMICALLY ALLOCATED MULTI-DIMENSIONAL ARRAYS

Dynamically allocated multi-dimensional arrays are not as easy to create as their statically allocated cousins, but

once you have created them they're as easy to use as a normal array. I'll show you two methods for dynamically creating a multi dimensional array.

The first method involves knowing the dimension of the smallest set of elements and dynamically allocating the largest. This is demonstrated in example 8.17.

*8.17 dynamic multi-dimensional
array allocation*

```

1  int    rows = 0;
2  const int cols = 5;
3  int  (*two_d_int_array)[cols];
4
5  cout<<"Please enter the number of rows: ";
6  cin>>rows;
7
8  two_d_int_array = new int[rows][cols];
9
10 for(int i=0; i<rows; i++)
11     for(int j=0; j<cols; j++)
12         two_d_int_array[i][j] = j+1;
13
14 for(int i=0; i<rows; i++){
15     for(int j=0; j<cols; j++){
16         cout<<two_d_int_array[i][j];
17     }
18     cout<<endl;
19 }
20 delete[] two_d_int_array; //release the array

```

Study the syntax on line 3. A pointer named `two_d_int_array` is declared to point to arrays containing 5 integer objects. What must be done next is to dynamically allocate the number of rows the two-dimensional array will contain. This is accomplished by getting an integer value from the user on lines 5 and 6 and assigning it to the variable `rows`. Next, the variable `rows` is used to create the array using familiar syntax on line 8. The rest of the code initializes the newly created array of integers and prints the values to the screen. This method is restrictive in that the values of the smaller dimensions must be known at compile time. Running the code from example 8.17 with a row value of 6 produces the results shown in figure 8-18.

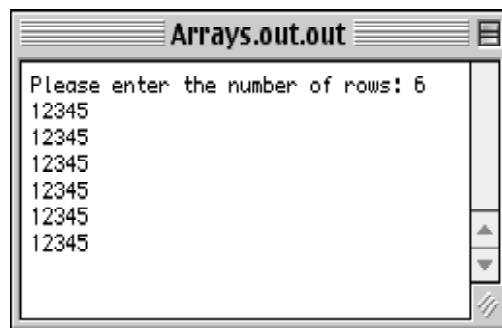


Figure 8-18: Results of Running Example 8.17 Using Row Value 6

To dynamically allocate both dimensions you must resort to the fundamental knowledge that a multi-dimensional array is an array of arrays. The following example allows the user to dynamically allocate both the rows and cols of a two-dimensional array.

8.19 dynamically allocating
2-dimensional array

```

1  int rows = 0;
2  int cols = 0;
3
4  int** two_d_int_array;
5
6  cout<<"Please enter number of rows and columns: ";
7  cin>>rows>>cols;
8
9  two_d_int_array = new int*[rows];
10
11 for(int i=0; i<rows; i++)
12     two_d_int_array[i] = new int[cols];
13
14 for(int i=0; i<rows; i++)
15     for(int j=0; j<cols; j++)
15         two_d_int_array[i][j] = j+1;
17
18 for(int i=0; i<rows; i++){
19     for(int j=0; j<cols; j++){
20         cout<<two_d_int_array[i][j];
21     }
22     cout<<endl;
23 }
24 for(int i = 0; i<rows; i++)
25     delete[] two_d_int_array[i]; //release each row array
26
27 delete[] two_d_int_array; //release array of rows

```

As you study example 8.19 you will notice there is an extra step required when utilizing this method. First, on line 4, a pointer to a pointer to an integer is declared. Lines 6 & 7 get the array dimensions from the user. Line 9 creates an array of integer pointers using the rows variable. The extra step occurs at line 11. Here the for loop allocates an array of integers for each row using the variable cols. The remaining code uses normal array notation to initialize each element of the array and print the values to the screen. Figure 8-19 shows the results of running this code using a rows value of 10 and cols value of 6.

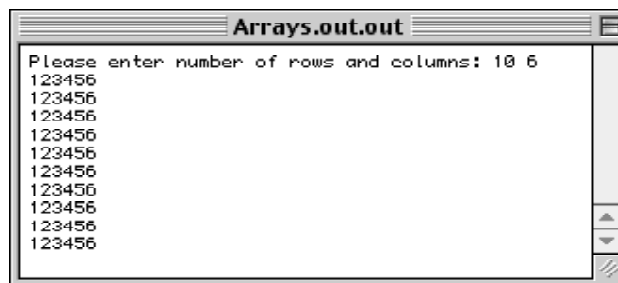


Figure 8-19: Results of Running Example 8.19 Using rows = 10 & cols = 6

STRINGS

A string in C++ is an array of characters terminated by the null character ‘\0’. The following code declares a character array and initializes it to a string literal:

```
char string1[] = "C++ For Artists";
```

You could then print this string to the screen by using the name of the array as is demonstrated by the following code:

```
cout<<string1<<endl;
```

You could have initialized the array using an initializer list. Examine the following code:

```
char string1[] = {'C', '+', '+', ' ', 'F', 'o', 'r', ' ', 'A', 'r', 't', 'i', 's', 't', 's', '\0'};
```

Notice the explicit inclusion of the ‘\0’ character. You will need to allow for one extra space in any array you intend to use for strings for the purpose of including the null character terminator.

SUMMARY

This chapter introduced you to arrays, their purpose, and their use. An array is a contiguous allocation of memory to homogeneous objects. Contiguous means the objects are stored one after another in memory, and homogeneous means the objects in the array are all the same type. Arrays can contain user-defined data types as well as fundamental data types like int or float. They can also contain pointers to these types and function pointers. Arrays of user-defined data types are covered in subsequent chapters.

An array begins at a certain memory address. A static array name is a const pointer. For statically allocated arrays of objects the array name points to the first element. Each successive object in an array is located one allocation unit from the previous object. The size of the allocation unit is determined by the type of objects an array is declared to contain.

There are two ways to access array elements: array subscripting using the [] operator and an index value, or via pointer arithmetic using the pointer dereference operator *. Thus, array_name[0] will yield the same element as *(array_name).

Beware the uninitialized array! Use an initializer list to set the values of an array at declaration.

Multi-dimensional arrays are arrays of arrays. Multi-dimensional arrays are stored in memory in row major order. Using named constants or variables in array declarations can help clarify the intended use of each dimension. Remember the pattern: smaller dimensional units are displaced to the right in array declarations.

A dynamic array is one that has been created in the application heap using the new[] operator. Do not forget to release dynamically allocated arrays using the delete[] operator or you will suffer memory leaks. Also, if you have an array of pointers to object, do not forget to release the memory for each dynamically created object using the delete operator before releasing the array.

Strings are arrays of characters terminated with a null character ‘\0’. Allow for one extra element in the sizing of any array in which you intend to hold or manipulate strings.

Skill Building Exercises

- Memory Representation:** Draw the memory representation for a statically allocated three dimensional array of integers.
- Static Array:** Write a program that declares a static array of ints with ten elements. Prompt the user to enter ten integer values. Store each value in the array and print the contents of the array to the screen.
- Static Arrays:** Write a program that declares five static arrays of ints each with ten elements. Use an initializer list to initialize each array to the following values. Print the values to the screen.

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0
7, 9, 10, 125, 256, 3, 25, 67, 78, 9
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1
12, 23, 45, 0, 0, 4, 5, 0, 8
```

- Array of Pointers:** Write a program that creates an array of integer pointers 5 elements long. Use an initializer list to set each pointer to NULL. Use a for loop to create the integer objects and assign their addresses to each array element. Print the values of each integer object to the screen. Remember to release the memory for each dynamically created object using the delete operator.
- Array of floats:** Write a program that creates an array of floats seven elements long. Prompt the user to enter the total of their daily expenses for seven days and assign each value to an array element. Sum the array elements to create a grand total and print the total to the screen.
- Array of ints:** Write a program that creates a two-dimensional array of ints. Make each dimension five elements long. Initialize each element value to zero using an initializer list. Use a nested for statement to print the contents of the array to the screen so that it looks similar to this:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

- String Processing:** Write a program that creates a char array 25 elements long. Prompt the user to enter a short character string and assign the string to the array. Reverse the order of characters in the array and print the results to the screen.
- Dynamic Arrays:** Write a program that creates a dynamic array of ten integer pointers. Prompt the user to enter ten numbers and dynamically create each integer object using the value entered by the user. Print the integer object values to the screen. Also print the pointer values. Remember to release all dynamically allocated memory.
- Dynamic Array:** Write a program that creates a dynamic array of integer objects. Prompt the user to enter the size of the array, and then prompt the user to enter the values for each integer. Sum the contents of the array and print the results to the screen. Remember to release all dynamically allocated memory before exiting the program.
- String Concatenation:** Write a program that creates two char arrays. Prompt the user to enter two short character strings and store one string in each array. Print the contents of each array to the screen. Next, calculate the size of each string and create a dynamic array large enough to hold both strings and combine the two. Print the combined string to the screen.

SUGGESTED PROJECTS

1. **Matrix Multiplication:** Given two matrices A_{ij} and B_{jk} the product C_{ik} can be calculated with the following equation:

$$C_{ik} = \sum_{j=1}^n a_{ij}b_{jk}$$

Write a program that multiplies the following matrices together and stores the results in a new matrix. Print the resulting matrix values to the screen.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

2. **Calculate Class Averages:** Computer Write a program that computes class averages. Allow the user to set the size of the array of floats based on class size. Prompt the user for each grade. Average the grades and print the results to the screen.
3. **Number Counter:** Write a program that counts the number of times a user enters a particular number between the values 1 through 20. Exit the program when the user enters a number outside that range. Before the program exits, print a histogram to the screen showing the distribution of the numbers entered by the user. The histogram may look something like this...

```
1: **
2: *****
3:
4: **
.
.
.
20: *****
```

...indicating the number 1 was entered two times, the number 2 was entered nine times, etc.

4. **String Reader:** Write a program that reads a string from the user of arbitrary length and counts the number of times each letter of the alphabet appear in the string. Print a histogram to the screen showing the results.
5. **Command Line Echo:** Write a program that echoes command line arguments by implementing the `main(int argc, char *argv[]){}` function. The `argc` argument is a count of the number of command line arguments with which the command was evoked. It includes the command itself so the value of `argc` will at least be 1 and will equal 1 in the absence of any command line arguments. The `argv` argument is an array of char pointers, each pointing to a particular command line string. For example, if the name of the program was `echo_args` then the following command line...

```
[localhost:~] swodog% echo_args This works great!
```

...would result in `argc = 4`, and `argv[0] = "echo_args"`, `argv[1] = "This"`, `argv[2] = "works"`, and `argv[3] = "great!"`.

SELF TEST QUESTIONS

1. An array is a _____ allocation of memory to _____ objects.
2. A static array name is what type of pointer?
3. (T/F) An array name points to the first element of the array.
4. How are multi-dimensional arrays stored in memory.
5. Multi-dimensional arrays are arrays of _____.
6. What determines the size of an allocation unit?
7. How does a dynamically allocated array differ from a statically allocated array?
8. List at least four types of objects an array can contain.
9. Describe the two methods available to access array elements.
10. To what values will the array elements be initialized to in the following declaration:

```
int int_array[25];
```
11. To what values will each array element be initialized to in the following declaration:

```
int int_array[25] = {1, 2, 3};
```
12. What's the difference between the new operator and the new[] operator?
13. Why is it important to release dynamically allocated array memory with the delete[] operator?
14. List and discuss the steps required to create a single dimensional dynamic array.
15. Discuss the two methods for creating multi-dimensional dynamic arrays. Which method would you prefer to use and why?

REFERENCES

Thomas H. Corman, et. al., *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990. ISBN: 0-262-03141-8.

Brian W. Kernighan, et. al., *The C Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1988. ISBN: 0-13-110370-9

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Paul J. Lucas. *The C++ Programmer's Handbook*. Prentice Hall PTR, Englewood Cliffs, New Jersey, 1992. ISBN: 0-13-118233-1

NOTES

CHAPTER 9



CHARLOTTE AND GEORGE

FUNCTIONS

LEARNING OBJECTIVES

- STATE THE PURPOSE AND USE OF FUNCTIONS IN C++
- EXPLAIN HOW TO DECLARE AND DEFINE FUNCTIONS
- STATE THE PURPOSE AND USE OF FUNCTION RETURN TYPES
- STATE THE PURPOSE AND USE OF FUNCTION PARAMETERS
- DESCRIBE THE CONCEPT OF FUNCTION CALLING
- EXPLAIN THE USE OF LOCAL FUNCTION VARIABLES AND THEIR SCOPING RULES
- DESCRIBE HOW TO PASS ARGUMENTS TO A FUNCTION BY VALUE AND BY REFERENCE
- DESCRIBE HOW TO MAXIMIZE FUNCTION COHESION AND MINIMIZE COUPLING
- DESCRIBE THE CONCEPT OF FUNCTION SIGNATURES
- DESCRIBE HOW TO OVERLOAD FUNCTIONS
- EXPLAIN THE CONCEPT OF RECURSION
- EXPLAIN THE CONCEPT AND USE OF FUNCTION POINTERS
- EXPLAIN HOW TO CREATE FUNCTION LIBRARIES
- UTILIZE FUNCTIONS IN YOUR C++ PROGRAMMING PROJECTS
- DESCRIBE HOW FUNCTIONS ARE USED TO MODULARIZE C++ PROGRAM FUNCTIONALITY
- DEMONSTRATE YOUR ABILITY TO MINIMIZE FUNCTION COUPLING AND MAXIMIZE FUNCTION COHESION IN C++ PROGRAMMING PROJECTS

INTRODUCTION

Functions provide a convenient way to package program behavior into manageable units with an eye towards reuse. However, as you will soon learn, writing effective and efficient functions takes more than simply breaking a large program into arbitrarily sized blocks of code.

In your studies of C++ and object-oriented programming you will be confronted by the terms function and member function, and wonder what is the difference between the two. The difference lies not in shape and form but rather in how each is employed.

When you think in terms of functions you are thinking in terms of program functional decomposition and the C-style way of writing programs. Chapter 3 provided a complete example of this approach to implementing a solution to a programming problem. In C-style programming the data structures required to implement a program are established and followed by the functions that manipulate those data structures. Although they are highly cohesive, which is a good thing, they are at the same time tightly coupled to the problem being solved via the file scope variables, which is a bad thing. It would be hard to reuse the functions appearing in chapter 3 in another program without rewriting them.

In C++ programming a member function is the term used to describe a function defined as part of a structure or class interface. A member function is a critical component of the data structure itself. There is a different type of thinking that goes into the creation of member functions. A member function is, by its very nature, tightly coupled to the member attributes of the class in which it appears because of the free access it has to those attributes. But now that is a good thing because when you think in object-oriented terms you stop thinking in functional decomposition terms, and think instead of a program's objects and the messages they pass between each other. These messages are passed via member functions. This is discussed in detail in chapters 11 and 12.

As you study object-oriented programming in general you will encounter the term method used interchangeably with the term member function. (e.g., "Invoke a method on an object...") The term method has its roots in the Small-talk programming language. In C++ you declare and define functions, not methods, although it is safe to think of one being like the other.

Ultimately, the only significant difference between a function and a member function is the function exists on its own, whereas the member function is declared as being associated with a class of objects, and thus has free access to the internals of those objects.

In this chapter I intend to show you how to write functions. Everything you learn here will apply directly to writing member functions. This will require you to understand and master several key concepts regarding functions to include function declaration and definition, function argument passing, and function variable scoping rules. You will learn how to write functions that minimize their connection or coupling to the outside world while maximizing the cohesion of the statements that comprise the body of the function. I will also show you how to share the functions you write via function libraries.

WHAT IS A FUNCTION?

A function is a collection of logically related program statements written to perform a specific processing activity. A function is also a code module. A function is given a name and with this name the function can be called or executed by any program that needs to use the function. The program statements that comprise the body of the function give the function its behavior. Function behavior can be built upon the behavior of other functions. In other words, functions can call other functions. Once a function is written and its behavior defined, you can effectively forget about the details of how the function performs its duties and call the function in the program when needed.

Grouping often-repeated program statements into a function saves memory space, but, most importantly, allows you to break apart a complex processing problem into a set of process abstractions. Just like well-chosen variable names lend a level of abstraction to data, well-chosen function names provide a way to achieve process abstraction.

A well-written or well-formed function can be used in many different programs on many different computers by many different programmers. This functionally is achieved by giving the function a singular purpose, or, in other words, maximizing its cohesiveness while at the same time decoupling it as much as possible from other program elements. The rule of thumb: maximize cohesion – minimize coupling.

A function can simply be a collection of often-repeated statements that returns no value. Functions of this type are extremely useful in their own right and are what Pascal programmers would call procedures. On the other hand, functions can communicate the results of their processing via either return values or parameter references.

INTERFACE VS. IMPLEMENTATION

All you need to know about a function to use it is its interface. A function's interface describes certain characteristics of the function such as its name, what type, if any, it returns, and what parameters, if any, it needs to perform its job. When writing a function, you will first declare the function's interface and then define what it means to be that function. You declare a function's interface by declaring a function prototype. You define what a function does by adding the necessary code to the body of a function definition to achieve the function's purpose.

PUT FUNCTION INTERFACE DECLARATIONS IN HEADER FILES

Place function declarations in header files. This is a good habit to form early in your programming career because you will do the exact same thing with class declarations. If you're writing a small program with only a few functions you can group all the required function declarations into one header file. If you're writing a large application, you can logically group related function declarations together. Putting function declarations in header files is the key to writing function libraries.

#ifndef...#define...#endif

Your header file should use the `#ifndef`, `#define`, & `#endif` preprocessor directives to allow you to include it in any file that needs access to the function declarations it contains. The structure of a typical header file will look like this:

```
#ifndef HEADER_NAME_H
#define HEADER_NAME_H

// function declarations appear here

#endif
```

Substitute your own header name where `HEADER_NAME_H` appears above. I use the name of the header file, in all caps, separating words and extensions with underscores. For example, if I name a header file `myheader.h` then the `#ifndef`, `#define`, `#endif` structure would look like so:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// function declarations appear here

#endif
```

The file `myheader.h` can now be included in other source files using the `#include` preprocessor directive without fear of getting multiple declaration errors:

```
#include "myheader.h"
```

PUT FUNCTION DEFINITIONS IN IMPLEMENTATION FILES

Place function definitions in a separate implementation file. An implementation file in C++ has the extension `.cpp` or `.hpp`. Whereas the header file contains the function's interface declaration, the implementation file will contain the definition of the function. A function definition gives meat or meaning to a function. It is where you as a programmer define exactly what it is the function will do using C++ programming statements in the body of the function. Every-

thing you have learned about C++ up to this chapter, and a whole lot more, can be used when you write the function code.

CHARACTERISTICS OF A WELL-WRITTEN FUNCTION

A well-written function should exhibit several fundamental characteristics. Several of these were briefly discussed above. A function should serve one purpose and its purpose should be reflected in its name, so it should be named well too. A function that does what it should and nothing surprising is said to be highly cohesive. In function writing you should strive to maximize the cohesiveness of your functions.

A function, to the fullest extent possible, should stand on its own, and not be too tightly coupled to other program elements. This characteristic, referred to as coupling, should, to the fullest extent possible, be minimized. The danger of having tightly coupled functions or code modules is that a change to one function may affect the behavior of another function or code module somewhere else in your program. Perhaps the most difficult task you face as a programmer of not just functions, but of object-oriented programs in general, is the minimization of intermodule dependencies or intermodule coupling.

Table 9-1 summarizes the characteristics of a well-written function.

Characteristic	Description
Singular purpose/Maximally cohesive	The program statements in the body of the function are logically related and exist to implement function behavior as described by the function's interface declaration (prototype). There should be nothing surprising going on in the body of the function that isn't hinted at in the function name.
Well-named	A function's name should reflect the function's purpose. Since most functions perform an action, function names should be formed from action words (verbs).
Minimally coupled	Take steps to reduce a function's dependency on other program elements. You can do this by using local function variables when possible, and passing other required program elements to a function via arguments.

Table 9-1: Characteristics of Well-Written Functions

DECLARING AND DEFINING FUNCTIONS

Now that you know what a function is it is time to learn how to create them. To write and use functions you will need to know how to do four things:

1. How to name the function
2. How to declare the function,
3. How to define the function, and
4. How to call the function in a program

NAMING FUNCTIONS

A function's name should reveal its purpose. A program written with well-named functions is easy to read and easy to maintain. If you are writing programs for the first time you will be tempted to write short, cryptic function names in the interest of completing a project on time. The problem with this shortcut approach is that if you have problems getting your program to run, your instructors have to decipher your code before they can help you. Some so-called professional programmers do not follow this advice because they think they're so good they do not need to follow any rules. These cowboy programmers produce code that's hard if not impossible to maintain when they finally do leave the project. Develop good naming habits early. Doing so will pay off in the long run.

A function name can be a valid identifier, but since functions invoke some sort of processing they should be

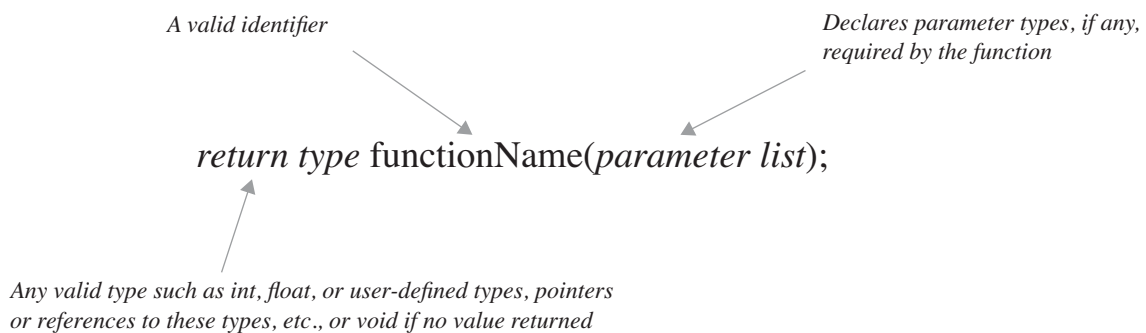
named using action words. The following examples show good function naming form:

```
getClassCount
setTemperatureValue
computeSum
addArrayElements
```

The naming convention used here is to lowercase the first letter of the first word and then use uppercase letters for the first letter of each word thereafter.

FUNCTION DECLARATION

Before a function can be defined or called from another function it must be declared. A function declaration takes the following form:



Here are a few examples of function declarations:

```
int getClassCount();
void setTemperatureValue(float temp_val);
float computeSum(float a, float b = 0);
double addArrayElements(double the_array[]);
```

The `getClassCount()` function is declared to return an integer type and take no arguments. The `setTemperature()` function is declared to return no value and take one floating point argument. The third function, `computeSum()`, takes two floating point arguments and returns a float value. The parameter `b` is set to a default value of zero. In the absence of a second argument, `computeSum()` will set the value of the parameter `b` to zero for use in the body of the function. This means that `computeSum()` can be called with either one or two arguments. The `addArrayElements()` function takes an array of doubles as an argument and returns a double value.

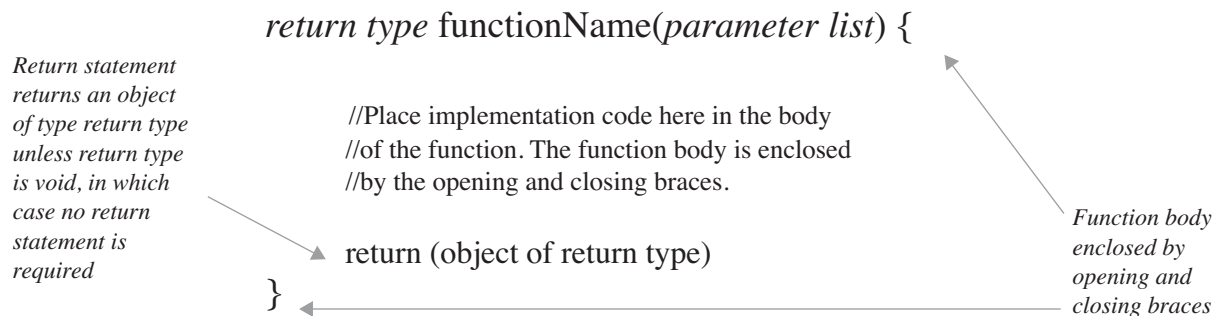
FUNCTION DEFINITION

Once a function has been declared it can be defined. Any programming statements required to give a function its behavior go into the body of the function definition. You will need a couple of things handy before you write a function definition. First, you will need access to the declaration of the function you are defining. If you have placed the declaration of the function in a separate header file you will need to include that file in the function implementation file. Second, you will need to include the header files for any other functions you are using to define your function.

Before studying a complete example take a look at the form of a function definition:

FUNCTION CALLING

A function is invoked via a function call. To call a function simply use the name of the function, supplying to it any arguments it requires. If you're calling functions written by other programmers, like third party libraries, then you will need to include the header file that contains the function declaration, and the library code so your development



environment can link to the function's object code. If you're writing the function, you will need to add the function's implementation file to your project before you compile. Let us take a look at a complete example.

A COMPLETE EXAMPLE

In this example I will step you through the declaration, definition, and use of a simple function called `testFunctionOne()` that prints a short message to the screen. First, create the header file and to it add the function declaration. I am naming the file `testfunctionone.h`:

```

1 #ifndef TEST_FUNCTION_ONE_H
2 #define TEST_FUNCTION_ONE_H
3
4 void testFunctionOne();
5
6 #endif

```

9.1 *testfunctionone.h*

Next, create the definition for `testFunctionOne()`. The definition should go in its own `.cpp` file. I am naming the file `testfunctionone.cpp`:

```

1 #include "testfunctionone.h"
2 #include <iostream>
3
4 using namespace std;
5
6 void testFunctionOne() {
7     cout<<"Function Called: testFunctionOne() "<<endl;
8 }

```

9.2 *testfunctionone.cpp*

Notice on line 1 I've included the header file `testfunctionone.h`. This will make the `testFunctionOne()` function declaration accessible to the `testfunctionone.cpp` file. If you fail to declare a function before you define it you will receive a compiler error stating something to the effect, "...function does not have a prototype." The exact message you receive will depend on your development environment.

On line 2 I've included the `iostream` header file. I need to do this because I'm using the `cout` object which is declared in that header file. The function definition for `testFunctionOne()` appears on lines 6 through 8. Since it is a void function no return statement is required. All that's left now is to use `testFunctionOne()`. I will use it in the `main()` function as is shown in the following example:

```

1 #include "testfunctionone.h"
2
3 int main() {
4     testFunctionOne();
5     return 0;
6 }

```

9.3 main.cpp

Contents of main.cpp

The main() function is in a file to itself called main.cpp. Notice on line 1 that testfunctionone.h is included. This allows access to the declaration of the function so it can be called in the main() function. testFunctionOne() is then called on line 4, followed by a return statement, which is required by the main() function.

The two files, testfunctionone.cpp and main.cpp can now be compiled. Figure 9-1 is a CodeWarrior project screen shot showing the two files as part of the project:

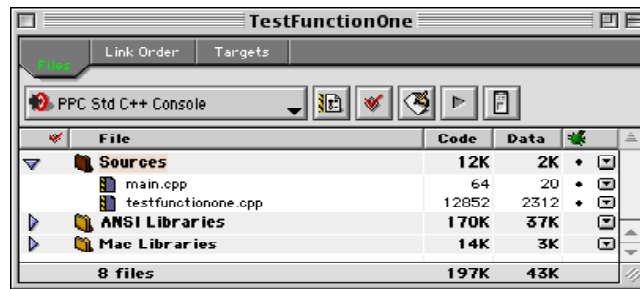


Figure 9-1: TestFunctionOne Project Screen Shot

The testfunctionone.h file is not explicitly added to the project but will be brought into the compilation process via the #include directive. It simply needs to reside in the same directory as the other project files. If you're using an IDE other than CodeWarrior your process may differ somewhat but the basic steps are the same. (see chapter 2)

When all files are ready to go the project can be compiled and run. Figure 9-2 shows the results of running the program and the message printed to the screen by testFunctionOne():

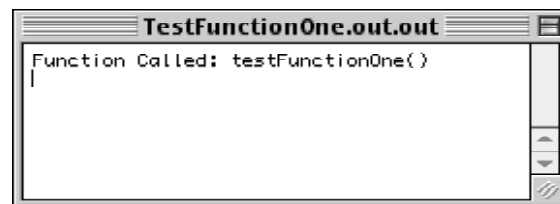


Figure 9-2: Results of Calling testFunctionOne()

Quick Review

Before a function can be defined or called it must first be declared. Start your function creation process by declaring the function in its own header file. That way you can use the #include directive to provide access to the function declaration to any file that needs it.

LOCAL FUNCTION VARIABLE SCOPING

testFunctionOne(), although complete, is a simple example. Most functions you write will be far more complex and will no doubt require arguments to be passed to it and a value of some sort returned. To master the art of writing meaningful functions you will need to get smart about variable scoping rules and how they apply to functions.

It is helpful to think of a function as a world unto itself. Variables can be declared and used within the body of a function. Any variables declared within the body of a function are referred to as local variables. Local variables exist for the life of the function, that is, when the function is called, any local variables required are set up and ready for use by the function. When the function returns, its stack frame collapses and with it go all the local variables. Therefore, local function variables, unless declared as being static, exist and retain their values only for as long as the function exists actively in memory.

In this section I will discuss how local variables can be declared and used within a function, how global or file scope variables can be masked or hidden by local variables, how to use scoping blocks within a function, how to declare and use static function variables, and how function parameters are used by a function.

DECLARING LOCAL VARIABLES

Variables can be declared and used within the body of a function. Any variables declared within the function are considered local to that function and are referred to as local variables. Study example 9.4:

```

1 void testFunctionTwo() {
2     int i = 2;
3     cout<<"Local i = "<<i<<endl;
4 }

```

9.4 local function variables

Line 2 declares and initializes a variable named `i` within the body of `testFunctionTwo()` and prints its value to the screen.

HIDING GLOBAL VARIABLES WITH LOCAL VARIABLES

A local variable declared within a function will hide a global variable of the same name. Consider the following example:

```

1 #include "testfunctiontwo.h"
2 #include <iostream>
3 using namespace std;
4
5 int i = 1;
6
7 void testFunctionTwo() {
8     int i = 2;
9     cout<<"Local i = "<<i<<endl;
10    cout<<"Global i = "<<::i<<endl;
11 }

```

9.5 masking global variables

The variable `i` declared at file scope on line 5 will be masked or hidden by the declaration of `testFunctionTwo()`'s local variable `i` declared on line 8. There are several lessons to be learned here. First, functions have access to any global variables declared within their translation unit unless hidden by a local variable of the same name. Second, if a local variable hides a global variable, the global variable can be accessed via the `::` operator as shown on line 10.

Using Scoping Blocks in Functions

Scoping blocks can be used within the body of a function to redeclare variables of the same name. Variables introduced in this fashion are considered to be within an enclosed scope and hide variables of the same name in the

enclosing scope. Study the following example:

```

1  #include "testfunctiontwo.h"
2  #include <iostream>
3  using namespace std;
4
5  int i = 1;
6
7  void testFunctionTwo() {
8      int i = 2;
9      cout<<"Local i = "<<i<<endl;
10     cout<<"Global i = "<<::i<<endl;
11     {
12         int i = 3;
13         cout<<"Block i = "<<i<<endl;
14     }
15 }
```

9.6 block scope

Variable i declared within scoping block hides variable of same name in outer block

You generally will not use scoping blocks for the express purpose of hiding outer block variables, but you will regularly experience their effect when you use iteration or looping statements. Consider the following code:

```

1  #include "testfunctiontwo.h"
2  #include <iostream>
3  using namespace std;
4
5  int i = 1; //global i
6
7  void testFunctionTwo() {
8
9      int i = 2; //local i
10     cout<<"Local i = "<<i<<endl;
11     cout<<"Global i = "<<::i<<endl;
12
13     for(int i=0; i<5; i++)
14         cout<<"for loop i = "<<i<<endl;
15 }
```

9.7 scope of variables in looping statements

The for loop's i variable has scope within the body of the for statement and hides the local variable i

The variable `i` declared in the for statement on line 13 hides the local variable with the same name declared on line 9. The global `i` variable can be accessed within the body of the for loop using the `::` operator.

STATIC FUNCTION VARIABLES

The local function variables you have seen so far are created and destroyed with each invocation of their associated function. If you want a local variable to retain its value between function calls you must declare the variable as being static using the `static` keyword.

A static local function variable will be initialized when the function is first called. Any change to the value of a static variable will be preserved for use by the next invocation of the function. Examine the code in example 9.8. Figure 9-3 shows the results of invoking `testFunctionTwo()` five times. On each invocation of the function both variables are incremented after they are printed to the screen as is shown on lines 9 and 10 of example 9.8. When `testFunctionTwo()` is called the first time, both variables are initialized to zero. On the second call to `testFunctionTwo()` and on each call thereafter, the local variable `i` is reinitialized to zero while the static variable `j` retains its incremented value from the prior function call.

Static function variables come in handy when you need to preserve function state between function calls. Although the value of the static variable is preserved between function calls, its scope is still local to the function in which it is declared.

```

1 #include "testfunctiontwo.h"
2 #include <iostream>
3 using namespace std;
4
5 void testFunctionTwo() {
6     static int i = 0;
7     int j = 0;
8
9     cout<<"Local Static i = "<<i++<<endl;
10    cout<<"Local Auto    j = "<<j++<<endl;
11 }

```

9.8 static function variables

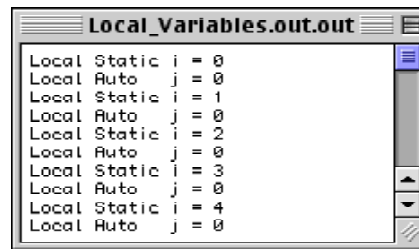


Figure 9-3: Results of Calling testFunctionTwo() Five Times with Static Variable

SCOPE OF FUNCTION PARAMETERS

When a function has a parameter list, the parameter names have local scope within the function. Any attempt to redeclare a local variable within the function that has the same name as one of its parameters will result in a compiler error. Function parameters hide global variables with the same name. Examine the following code:

```

1 #include "testfunctionthree.h"
2 #include <iostream>
3 using namespace std;
4
5 int i = 25;
6
7 void testFunctionThree(int i) {
8     cout<<"Parameter i = "<<i<<endl;
9     cout<<"Global    i = "<<::i<<endl;
10 }

```

9.9 masking function parameters

Function takes one integer argument named *i*. *i* has local scope within testFunctionThree()

In this example, the function testFunctionThree() is defined to take one integer argument. The parameter name used to access the value of the argument supplied to the function when it is called is *i*. The following example shows testFunctionThree() being called from a main() function with the argument 5:

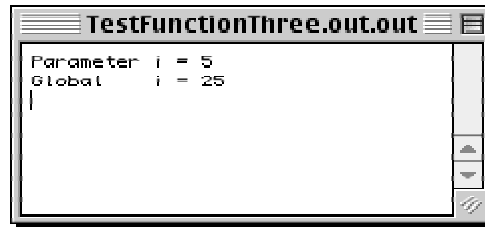
```

1 #include <iostream>
2 #include "testfunctionthree.h"
3 using namespace std;
4
5 int main() {
6     testFunctionThree(5);
7     return 0;
8 }

```

9.10 function call with argument

Figure 9-4 shows the result of running this program.



```

TestFunctionThree.out.out
Parameter i = 5
Global i = 25

```

Figure 9-4: Results of Calling testFunctionThree() with an Argument Value of 5

Quick Review

Variables declared inside the body of a function have local scope within that function. Local variables will hide global variables of the same name. Local variables exist for the life of the function and will be reinitialized each time the function is called. If you need a local variable to retain its value between function calls use the static keyword in its declaration. Function parameters have local scope within the function and mask global variables with the same name. To access global variables with the same name as local variables use the `::` operator.

PASSING ARGUMENTS TO FUNCTIONS

Arguments are passed to functions in two ways; by value or by reference. It is important to know the difference between the two and the effect each form of argument passing produces. But first, I want to show you some of the mechanics of a function call so you will better understand what's happening behind the scenes when one function calls another.

FUNCTION CALLING

It is helpful to understand the concepts of function activation records and function calling protocols. Figure 9-5 shows the sequence of activation records of a calling function and a called function before, during, and after a function call.

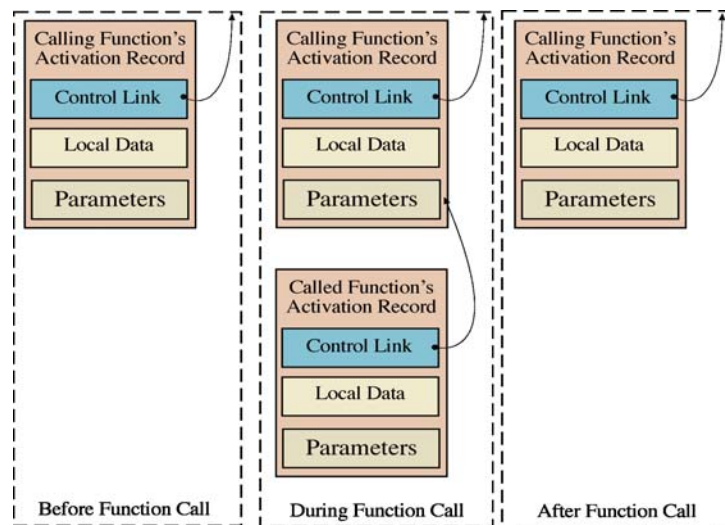


Figure 9-5: Function Activation Record Sequence

Every programming language that implements functions assigns a set of responsibilities to the function making the call and the function being called. These responsibilities are referred to as function calling conventions or function calling protocols, or simply calling conventions. The important thing to know about calling conventions is that if you're only programming in C++ you can read this section and then safely forget about them. On the other hand, if you're planning to do mixed language programming, you will have to be aware that some programming languages have different calling conventions than others.

Responsibilities of the Calling Function

A calling function is usually responsible for providing a called function access to any arguments with which it was called. As you will see below, a calling function will usually make a called function's arguments available in processor registers.

Responsibilities of the Called Function

A called function must save the contents of any processor register it intends to use, and restore the contents of those registers before returning to the called function. As figure 9-5 illustrates, a link between the called function and calling function is established in each activation record. The control link facilitates a called function's return to its caller.

PASSING ARGUMENTS by VALUE

You saw an argument passed to a function by value in the testFunctionThree() function shown in examples 9.9 and 9.10. When an argument is passed to a function by value, the value of the argument is copied to the function parameter for use inside the function. Figure 9-6 shows a partial disassembly of the main.cpp file from the testFunctionThree program. The listing is given in PowerPC assembly.

```

1  mflr      r0 ←————— move contents of link register to r0
2  stw      r0, 8 (SP) ←————— store r0 to stack pointer + 8
3  stwu     SP, -64 (SP) ←————— allocate some stack space
4  li       r3, 5 ←————— load 5 into r3
5  bl      .testFunctionThree__Fi ←—— call testFunctionThree
6  nop
7  li       r3, 0
8  lwz     r0, 72 (SP)
9  addi    SP, SP, 64
10 mtlr   r0
11 blr

```

} ————— Cleanup and return

Figure 9-6: Partial Disassembly of main.cpp

The first thing main() does on lines 1 and 2 is to preserve the contents of the link register. This value will be used to return control to the function that called this program. Next, on line 3, the stack pointer is reset. Remember, the stack grows down and the heap grows up. That's why a negative value is added to the stack pointer.

On line 4, the integer value 5 is loaded into r3. This is followed by a branch to testFunctionThree(). Upon testFunctionThree()'s return, the r3 is set to zero; this is the return value. Next, the saved link register value is retrieved from the stack and placed in r0. The stack is reset to its previous position and the return is made.

What has main() done? Essentially, it placed the value of testFunctionThree()'s argument in a register so it would be accessible to it when called. After the call to testFunctionThree(), main() placed a zero in the same register so it would be available to main()'s caller.

What do you suppose testFunctionThree() will do? Before looking at figure 9-7 go back and look at the C++ listing for testFunctionThree(). It has access to two values: a global i which equals 35, and a local parameter named i which has been set by main(). Remember, every function will go through the same house keeping chores. These include preserving the return link and preserving and restoring any registers used by the function.

```

1  Hunk:Kind=HUNK_GLOBAL_IDATA  Align=4  Class=RW  Name="i" (20)  Size=4
2  00000000: 00 00 00 19                                     '....'
3
4  Hunk:Kind=HUNK_LOCAL_IDATA   Align=1  Class=RW  Name="@661" (21)  Size=15
5  00000000: 50 61 72 61 6D 65 74 65 72 20 69 20 3D 20 00  'Parameter i = .'
6
7  Hunk:Kind=HUNK_LOCAL_IDATA   Align=1  Class=RW  Name="@662" (22)  Size=15
8  00000000: 47 6C 6F 62 61 6C 20 20 20 20 69 20 3D 20 00  'Global i = .'
9
10 Hunk:Kind=HUNK_GLOBAL_CODE   Align=4  Class=PR  Name=".testFunctionThree__Fi" (23)  Size=148
11 mflr      r0
12 stw      r0,8(SP)
13 stwu     SP,-64(SP)
14 stw     r3,88(SP)
15 lwz     r3,cout__3std(RTOC)
16 lwz     r4,@661(RTOC)
17 bl     .__ls<Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream
18 <c,Q23std14char_traits<c>>PCC
19 nop
20 lwz     r4,88(SP)
21 bl     .__ls__Q23std39basic_ostream<c,Q23std14char_traits<c>>Fi
22 nop
23 lwz     r4,end1<c,Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream
24 <c,Q23std14char_traits<c>>(RTOC)
25 bl     .__ls__Q23std39basic_ostream<c,Q23std14char_traits
26 <c>>FPFRQ23std39basic_ostream<c,Q23std14char_traits
27 <c>>_RQ23std39basic_ostream<c,Q23std14char_traits<c>>
28 lwz     r3,cout__3std(RTOC)
29 lwz     r4,@662(RTOC)
30 bl     .__ls<Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream
31 <c,Q23std14char_traits<c>>PCC
32 nop
33 lwz     r4,i(RTOC)
34 lwz     r4,0(r4)
35 bl     .__ls__Q23std39basic_ostream<c,Q23std14char_traits<c>>Fi
36 nop
37 lwz     r4,end1<c,Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream
38 <c,Q23std14char_traits<c>>(RTOC)
39 bl     .__ls__Q23std39basic_ostream<c,Q23std14char_traits
40 <c>>FPFRQ23std39basic_ostream<c,Q23std14char_traits
41 <c>>_RQ23std39basic_ostream<c,Q23std14char_traits<c>>
42 lwz     r0,72(SP)
43 addi    SP,SP,64
44 mtlr    r0
45 blr

```

Figure 9-7: Partial Disassembly of testFunctionThree.cpp

The assembly code for testFunctionThree() is more complicated than main() because it is making several iostream function calls itself. I have to remind you, this is not the complete assembly listing. Due to the iostream functions the complete listing is just too long and has a lot of stuff not related to the discussion, but it is interesting to look at!

The function starts on lines 11 and 12 where the contents of the link register is saved for the return trip. Line 13 allocates some stack space for the function. The instruction on line 14 stores the value of r3 in memory for future use. Remember, r3 is where main() put the value 5. Line 15 loads the location of the cout object code in r3 in preparation for the iostream calls. (RTOC stands for Table of Contents Register) Line 16 loads the location of the string “Parameter i = “, followed by line 17’s branch to the code that prints it out.

Next, line 20 loads the value 5 from memory to r4, followed by a call to the code to print it out. Line 23 loads the endl code in r4, followed by a call to the code that prints it.

Line 29 loads the string, “Global i = “, followed by the call to the code to print the string. Next, on line 33, the global variable i’s address is loaded into r4, and then, on line 34, the value pointed to by the address is loaded into r4. Where is it getting i from? Look at lines 1 and 2. The symbolic name for that hunk of data is “i” and its value is hexadecimal 19, which equals 25 in decimal. Once its value is loaded in r4 it is printed to the screen. The rest of the function prints endl, does housekeeping and returns to main().

It is a good idea to disassemble simple functions to see how they work. Although high-level languages are meant to provide a certain level of abstraction, it is helpful to know what is happening at the assembly level. And although PowerPC assembly is used here, the same principles apply to other processors as well.

ANOTHER EXAMPLE

Let us look at another example of passing arguments by value that better shows the side effects of using this method. The header file, implementation file, and main file are shown in examples 9.11 through 9.13 below.

```
1 #ifndef TEST_FUNCTION_FOUR
2 #define TEST_FUNCTION_FOUR
3
4 void testFunctionFour(int input);
5
6 #endif
```

9.11 testfunctionfour.h

```
1 #include <iostream>
2 #include "testfunctionfour.h"
3 using namespace std;
4
5 void testFunctionFour(int input){
6     input++;
7     cout<<"Function argument input = "<<input<<endl;
8 }
```

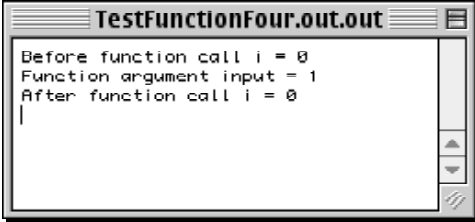
9.12 testfunctionfour.cpp

```
1 #include <iostream>
2 #include "testfunctionfour.h"
3 using namespace std;
4
5 int main(){
6     int i = 0;
7     cout<<"Before function call i = "<<i<<endl;
8     testFunctionFour(i);
9     cout<<"After function call i = "<<i<<endl;
10    return 0;
11 }
```

9.13 main.cpp

In this example, an integer variable named *i* is declared in the `main()` function and initialized to zero. The value of *i* is printed to the screen before *i* is passed to the function `testFunctionFour()` as an argument. `testFunctionFour()` is declared to take an integer argument with the parameter name `input`. Inside the body of `testFunctionFour()`, `input` is incremented by one and its value is then printed to the screen. `testFunctionFour()` then returns control to the `main()` function and the value of *i* is again printed to the screen. What do you suppose the value of *i* will be when it is printed the second time? Figure 9-8 shows the results of running this program.

As you can see, the manipulation of `testFunctionFour()`'s `input` parameter had no effect on the value of the variable *i*. That's because the value of *i* was copied to `testFunctionFour()`'s `input` parameter, and it was this copy that was incremented and printed to the screen inside the function. When you need to manipulate directly the values of the



```

TestFunctionFour.out.out
Before function call i = 0
Function argument input = 1
After function call i = 0
|

```

Figure 9-8: Results of Running testFunctionFour Program

arguments passed to functions you should pass those arguments by reference. This is discussed in the next section.

PASSING ARGUMENTS by REFERENCE

When you need to directly manipulate a function argument or intend for a function to work on large objects, you should pass the argument to the function by reference, meaning, you need to supply the argument's memory address to the function so it has direct access to the argument. The argument is then accessed via its memory address. In the case of large objects, passing arguments by reference can significantly increase processing efficiency.

It is important to note what is really happening when you pass an argument by reference. The memory address is being copied to the function parameter (pass by copy), but the object can then be manipulated directly via this address. The efficiency comes from only copying word sized objects (memory addresses) no matter the size of the object itself. The following test program demonstrates how addresses are passed by copy.

```

1 #include <iostream>
2 using namespace std;
3
4 /*****
5     Function Declaration
6     *****/
7
8 void addressCopyTest(int* ipA);
9
10 /*****
11     Function Definition
12     *****/
13
14 void addressCopyTest(int* ipA){
15     cout<<"Address of ipA = "<<ipA<<"   Value at ipA = "<<*ipA<<endl;
16     ipA++;
17     cout<<"Address of ipA = "<<ipA<<"   Value at ipA = "<<*ipA<<endl;
18 }
19
20 /*****
21     main() Function
22     *****/
23
24 int main(){
25     int i = 3;
26     cout<<"Address of i = "<<&i<<"   value of i = "<<i<<endl;
27     addressCopyTest(&i);
28     cout<<"Address of i = "<<&i<<"   value of i = "<<i<<endl;
29
30     return 0;
31 }

```

9.14 passing addresses
by copy

The function `addressCopyTest()` takes an integer pointer as an argument. Inside the function the address and value referenced by the address of the `ipA` parameter is printed to the screen. The `ipA` parameter is then incremented and the address and value is again printed to the screen.

In the `main()` function, an integer variable named `i` is declared and initialized to 3. `i`'s address and value is printed to the screen both before and after the call to the `addressCopyTest()` function. Figure 9-9 shows the results of running this program. Note the address of `i` has not changed even though the address passed to the function was incremented during the function call. This happens because although you're passing arguments by reference, you're passing the address to these arguments by copy. The addresses you see if you run this program on your computer will be different, but the ultimate result will be the same.


```

AddressCopyTest.out.out
Address of i = 0x178ee098 value of i = 3
Address of ipA = 0x178ee098 Value at ipA = 3
Address of ipA = 0x178ee09c Value at ipA = 0
Address of i = 0x178ee098 value of i = 3

```

Figure 9-9: Results of Running addressCopyTest Program

CONTINUING THE STORY...

Passing by reference can be accomplished in two ways. You can use pointers, and do all the pointer dereferencing yourself, or you can use references, and have the pointer dereferencing done for you. There are advantages and disadvantages to each method.

The advantage of using pointers is the flexibility they provide. If you know how to manipulate pointers the C++ programming world is your oyster! (Remember chapter 7? Vaguely? I thought so!) The biggest disadvantage I can think of to using pointers for passing arguments by reference is that sloppy use can result in bugs that are difficult, if not impossible, to detect. Pointer misuse often results in memory leaks. However, experience and attention to detail should mitigate this disadvantage to a large extent.

If you use references to pass arguments to functions your code is cleaner and easier to read. However, you are restricted to what you can do with a reference, hence, you lose the programming flexibility otherwise enjoyed with the use of pointers.

Which method you use depends directly on what you need to do inside the function. For some jobs, either method will work fine; for others, your only option will be to use pointers. Let us take a closer look at each method.

PASSING POINTERS

To pass an argument to a function using a pointer you must declare the function to take a pointer type argument. The following statement declares a function that takes an integer pointer as an argument:

```
void testFunctionFive(int* ipA);
```

When `testFunctionFive()` is called, the address of an integer object must be supplied as an argument. The following example gives the complete definition of `testFunctionFive()`:

```

1 #include "testfunctionfive.h"
2
3 void testFunctionFive(int* ipA) {
4     (*ipA)++;
5 }

```

9.15 testfunctionfive.cpp

On line 4, the parameter `ipA` is dereferenced and the resulting value is incremented. The parentheses are used to explicitly show the operator association. Example 9.16 shows `testFunctionFive()` being called in a `main()` function.

On line 5, two global variables `i` and `j` are declared and initialized. Their value is printed to the screen on line 8 prior to calling `testFunctionFive()` for the first time. Lines 9 and 10 show `testFunctionFive()` being called twice, once with the address of `i`, and the second time with the address of `j`. Note how the address of each variable is passed to the function by using the `&` operator.

The values of `i` and `j` are printed to the screen again on line 11. Figure 9-10 shows the results of running this program. Notice that after the function calls the values of the global variables have changed.

This example showed you how to pass the address of a variable. You prefix the `&` operator to the variable name to get its address and it is this address that is passed to the function. If the variable you want to pass to the function is a pointer then you pass only the name of the variable and leave off the `&` operator. Example 9.17 uses `testFunctionFive()` again to demonstrate.

```

1 #include <iostream>
2 #include "testfunctionfive.h"
3 using namespace std;
4
5 int i=1, j=2;
6
7 int main(){
8     cout<<"Value of i: "<<i<<" j: "<<j<<endl;
9     testFunctionFive(&i);
10    testFunctionFive(&j);
11    cout<<"Value of i: "<<i<<" j: "<<j<<endl;
12    return 0;
13 }
```

9.16 main.cpp

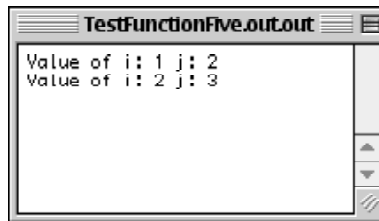


Figure 9-10: Results of Running testFunctionFive Program

```

1 #include <iostream>
2 #include "testfunctionfive.h"
3 using namespace std;
4
5 int main(){
6     int* ip1 = new int(3);
7
8     cout<<"Value of integer pointed to by ip1 = "<<*ip1<<endl;
9     testFunctionFive(ip1);
10    cout<<"Value of integer pointed to by ip1 = "<<*ip1<<endl;
11    delete ip1;
12    return 0;
13 }
```

9.17 main.cpp

Referring to example 9.17, an integer pointer is declared and initialized to the address of an integer object allocated on the heap. The pointer is passed to testFunctionFive() on line 9. Notice that delete must be called on ip1 to release the heap memory.

PASSING REFERENCES

Remember references? They're kinda like pointers except a pointer is a variable and a reference is not. You can change what a pointer points to but once you set a reference it can't be changed to refer to anything else. To pass an argument to a function in the form of a reference requires the function prototype to declare parameters of type reference. This confuses many students because the & operator is overloaded for this purpose. The following code shows a function named testFunctionSix() that's declared to take an integer reference as an argument:

```
void testFunctionSix(int& irA);
```

A reference to an argument is treated differently than a pointer in the body of the function. Examine the definition of testFunctionSix() given in example 9.18 below.

```

1 #include "testfunctionsix.h"
2
3 void testFunctionSix(int& irA){
4     irA++;
5 }

```

9.18 testfunctionsix.cpp

When an argument is passed to a function in the form of a reference the parameter name is used as if it were that object. No pointer dereferencing is required. Notice above on line 4 how the parameter `irA` is incremented and compare this with how the equivalent operation is performed on a pointer in example 9.15.

Now, to call this function with an integer argument requires no special action on your part. You simply use the name of the object as shown in the following example:

```

1 #include <iostream>
2 #include "testfunctionsix.h"
3 using namespace std;
4
5 int main(){
6     int i = 3;
7     cout<<"Value of i before function call = "<<i<<endl;
8     testFunctionSix(i);
9     cout<<"Value of i after function call = "<<i<<endl;
10    return 0;
11 }

```

9.19 main.cpp

Figure 9.11 shows the results of running this program.

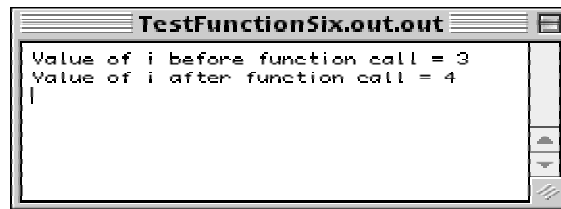


Figure 9.11 Results of Running testFunctionSix Program

PASSING ARRAYS TO FUNCTIONS

Arrays are passed to functions by reference. The name of the array is a pointer that contains the address of the first element of the array. (see chapter 8)

The first step in writing a function that takes an array as an argument is to declare the function in such a way that it knows what to expect and that readers of the function declaration can figure out what you're trying to do! The following code declares a function named `printIntArray()` that prints the contents of an integer array. It takes a single-dimensional integer array as the first argument and the number of array elements as the second:

```

1 #ifndef Print_Int_Array_H
2 #define Print_Int_Array_H
3
4 void printIntArray(int intArray[], int elements);
5
6 #endif

```

9.20 printIntArray()

The function definition follows:

```

1 #include "printintarray.h"
2 #include <iostream>
3 using namespace std;
4
5 void printIntArray(int intArray[], int elements){
6     for(int i = 0; i < limit; i++)
7         cout<<intArray[i]<<" ";
8     cout<<endl;
9 }

```

9.21 *printintarray.cpp*

The following main() function illustrates how the function is called:

```

1 #include "printintarray.h"
2
3 int main(){
4     int my_array[] = {1,2,3,4,5,6,7,8,9,10};
5     printIntArray(my_array, (sizeof my_array / sizeof(int)));
6     return 0;
7 }

```

9.22 *main.cpp*

Referring to the main() function shown above, an integer array named my_array is declared and initialized on line 4. The printIntArray() function is called on line 5. Notice that only the name of the array is passed as the first argument. The sizeof operator is used to calculate the size of the array in bytes, which is then divided by the size of the int type in bytes to determine the number of elements contained in my_array. The result of this expression is passed as an argument to printIntArray()'s elements parameter.

PASSING Multi-DIMENSIONAL ARRAYS TO FUNCTIONS

Multi-dimensional arrays are passed to functions in the same manner as their single-dimensional counterparts. Your primary concern is how to declare the function so it knows what arguments to expect. Examine the following function declarations:

```

print2DIntArray(int intArray[][5], int rows);
print3DIntArray(int intArray[][3][5], int sheets);

```

The print2DIntArray() function declares an array parameter of two dimensions, the right-most dimension being specified as 5 and the left-most dimension unspecified. There can be only one unspecified dimension, which is the case with normal static array declarations. The print3DIntArray() function declaration specifies an array parameter of three dimensions, two of which are specified and one not.

You can specify all dimensions of the multi-dimensional array if you desire. Doing so lessens ambiguity but at the price of flexibility. The following complete example declares, defines, and uses an alternative version of the print2DIntArray() function.

```

1 #ifndef PRINT_2D_INT_ARRAY_H
2 #define PRINT_2D_INT_ARRAY_H
3
4 const int ROWS = 5;
5 const int COLS = 5;
6
7 void print2DIntArray(int intArray[ROWS][COLS]);
8
9 #endif

```

9.23 *print_2d_int_array.h*

Notice on lines 4 and 5 that two integer constants have been declared to specify the dimensions of the array. These are included in the header file and can be used by any file that includes the `print_2d_int_array.h` header as is shown in the following code:

```

1 #include "print_2d_int_array.h"
2 #include <iostream>
3 using namespace std;
4
5 void print2DIntArray(int intArray[ROWS][COLS]) {
6     for(int i = 0; i < ROWS; i++){
7         for(int j = 0; j < COLS; j++){
8             cout<<intArray[i][j]<<" ";
9         }
10    cout<<endl;
11    }
12 }
```

9.24 print_2d_int_array.cpp

Notice how the constants are then used in the body of the function to manipulate the array. The following `main()` function shows the `print2DIntArray()` function in action:

```

1 #include "print_2d_int_array.h"
2
3 int main(){
4     int my_2d_array[ROWS][COLS] = {{1, 2, 3, 4, 5},
5                                     {2, 3, 4, 5, 1},
6                                     {3, 4, 5, 1, 2},
7                                     {4, 5, 1, 2, 3},
8                                     {5, 1, 2, 3, 4}};
9
10    print2DIntArray(my_2d_array);
11
12    return 0;
13 }
```

9.25 main.cpp

ANOTHER EXAMPLE

The following example program reuses the `printIntArray()` function used in examples 9.20 through 9.23. Another function called `sortIntArray()` is declared and defined. `sortIntArray()` takes an integer array as an argument and sorts the contents of the array. The `printIntArray()` function is used to print the array to the screen. The following code declares the `sortIntArray()` function:

```

1 #ifndef SORT_INT_ARRAY_H
2 #define SORT_INT_ARRAY_H
3
4 void sortIntArray(int intArray[], int elements);
5
6 #endif
```

9.26 sort_int_array.h

Example 9.27 gives the definition of `sortIntArray()`. Notice how the function performs exactly what its name implies. It performs the sort on the array elements. It doesn't print anything to the screen or otherwise do something not hinted at by its name. This is an example of a highly cohesive and loosely coupled function.

```

1 #include "sort_int_array.h"
2
3 void sortIntArray(int intArray[], int elements){
4     for(int i = 0; i<elements; i++){
5         for(int j = 1; j<elements; j++){
6             if(intArray[j-1] > intArray[j]) {
7                 int temp = intArray[j-1];
8                 intArray[j-1] = intArray[j];
9                 intArray[j] = temp;
10            }
11        }
12    }
13 }

```

9.27 *sort_int_array.cpp*

The following main() function shows how both printIntArray() and sortIntArray() are used together in a program:

```

1 #include <iostream>
2 #include "sort_int_array.h"
3 #include "printintarray.h"
4 using namespace std;
5
6 int main(){
7     int myArray[] = {10,5,9,4,3,8,2,7,6,1,0};
8
9     printIntArray(myArray, ((sizeof myArray)/sizeof(int)));
10    sortIntArray(myArray, ((sizeof myArray)/sizeof(int)));
11    printIntArray(myArray, ((sizeof myArray)/sizeof(int)));
12
13    return 0;
14 }

```

9.28 *main.cpp*

Figure 9-12 shows the results of running this program.

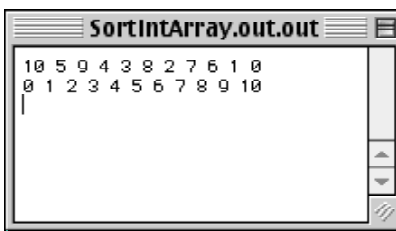


Figure 9-12: Results of Running Example 9.28

USING FUNCTION RETURN VALUES

Functions can return values which can then be used by the function or program that called the function. Up to this point, all the example functions, except main(), have been declared to return void, which means they do not return a value. Functions can return objects of fundamental data types such as int, float, char, etc., or objects of user-defined types. User-defined types are discussed in chapters 10 and 11. Functions can also return pointers and references to these types as well as pointers to other functions. The important thing to remember about functions is that they can be

used anywhere in your program where their return type and value can be used. For example, a function that returns an integer object can be used anywhere you could otherwise use an integer object. Let us first take a look at functions that return objects.

RETURNING OBJECTS

Specify the type of object you want a function to return in its declaration. Check out the following examples:

```
int returnInt();
float calculatePay( int hours_worked, float hourly_rate);
double getStarCount();
bool engineIsOn();
```

Once you have specified the type of object the function will return you have to return an object of that type when you define your function. The following example gives the function definition for the returnInt() function declared above:

```
1 #include "returnint.h"                                9.29 returnint.cpp
2
3 int returnInt(){
4     return 2;
5 }
```

This function returns the integer value 2 via the return statement on line 4. The integer value could have also been a local integer variable, or perhaps the result of a calculation. For instance, line 4 could have been written like so:

```
return 1 + 1;
```

The following main() function shows the returnInt() function in use:

```
1 #include <iostream>                                    9.30 main.cpp
2 #include "returnint.h"
3 using namespace std;
4
5
6 int main(){
7     cout<<returnInt()<<endl;
8     return 0;
9 }
```

Notice here how the returnInt() function was used as an input to the cout object. Since returnInt() returns an integer object, it can be used anywhere an integer can be used.

THE RETURN KEYWORD: MANTRA ON PROPER USAGE

The return statement should be the last line of code in the function. I'd like to say that one more time. The return statement should be the last line of code in the function. And there should only be one return statement in a function. I repeat. There should only be one return statement in a function. Now, having said that I will add that there are good reasons to violate this mantra.

RETURN KEYWORD MANTRA ANCILLARY

If you can keep the multiple return statements close together so you can see intuitively where they are in relation to the function's code, then you should be safe.

ANOTHER EXAMPLE

Let us take a look at another relatively simple example, but this time the value returned will depend on the value of the argument used when the function is called. The following code gives the declaration for a function called `square()`:

```

1 #ifndef SQUARE_H
2 #define SQUARE_H
3
4 double square(float value = 1);
5
6 #endif

```

9.31 square.h

The `square` function is declared to take a float argument whose parameter name is `value` and has a default argument value of 1.

DEFAULT ARGUMENT VALUES

Setting a parameter to a default value enables the function to be called in two different ways: 1) with an argument, and 2) without an argument. In the first case, the parameter will be set to whatever the argument's value happens to be. In the second case, the parameter value will be set to 1. The following example shows the function definition for the `square` function:

```

1 #include "square.h"
2
3 double square(float value){
4     return (value * value);
5 }

```

9.32 square.cpp

Nothing surprising going on here. The function simply returns the result of the parameter value multiplied by itself. Now, examine the `main()` function showing the `square()` function in use:

The `square()` function makes its appearance on line 14 above. After the user's input is read into the 25 character array named `input`, the array is used as an argument to a C Standard Library function named `atof()`. (ASCII to float) As its name implies, the `atof()` function converts an ASCII character string to a floating point value. Notice how the `atof()` function appears as an argument to the `square()` function. This example brings together pretty much everything you have learned about functions up to this point. The rest of the `main()` function just provides a simple user interface and program control.

RETURNING POINTERS

A function can return the address of an object. Just like the case of a function returning an object, a function that returns a pointer can be used wherever a pointer of that type is required. To illustrate this concept let us take a look at an example program that dynamically creates pointers to integer objects and stores the addresses in an array. The following code declares a function named `getNewIntAddress()` that will perform the dynamic memory allocation and return a pointer to the integer object created:

```

1 #ifndef GET_NEW_INT_ADDRESS_H
2 #define GET_NEW_INT_ADDRESS_H
3
4 int* getNewIntAddress();
5
6 #endif

```

9.34 getnewintaddress.h


```

1 #include <iostream>
2 #include <stdlib.h>
3 #include "square.h"
4
5 using namespace std;
6
7 int main(){
8     bool keep_going = true;
9     char input[25];
10
11     while(keep_going){
12         cout<<"Please enter a value to square: ";
13         cin>>input;
14         cout<<endl<<"The squared value is: "<<square(atoi(input))<<endl;
15         cout<<"Continue? Y or N: ";
16         cin>>input[0];
17         switch(input[0]){
18             case 'y':
19                 case 'Y': break;
20                 case 'n':
21                 case 'N': keep_going = false;
22                     break;
23                 default: break;
24             }
25         }
26     return 0;
27 }

```

9.33 *main.cpp*

Notice the return type declared on line 4 is of type pointer to int. Example code 9.35 gives the function definition:

```

1 #include "getnewintaddress.h"
2
3 int* getNewIntAddress(){
4     return (new int);
5 }

```

9.35 *getnewintaddress.cpp*

All `getNewIntAddress()` does is create a new integer object in the heap and return its address. Example 9.36 gives a `main()` function showing the `getNewIntAddress()` function in action.

The `getNewIntAddress()` function is called in the body of the first `for` statement on line 9. When `getNewIntAddress()` is called the resulting address is assigned to `ip_array[i]`. The assignment is enclosed in parentheses and dereferenced using the `*` operator. The resulting integer object is then assigned the value `(i+1)`. The next `for` statement on line 11 prints the contents of the newly-filled array. The `for` statement on line 16 simply modifies the integer values, followed by the `for` statement on line 19 which prints the contents of `ip_array` to the screen again. The `for` statement on line 22 iterates over `ip_array` and deletes each pointer to free up memory. Failure to release the memory using `delete` would result in a memory leak.

9.36 main.cpp

```

1  #include <iostream>
2  #include "getnewintaddress.h"
3  using namespace std;
4
5  int main(){
6      int* ip_array[10];
7
8      for(int i = 0; i < 10; i++)
9          *(ip_array[i] = getNewIntAddress()) = (i+1);
10
11     for(int i = 0; i < 10; i++)
12         cout<<*ip_array[i]<<" ";
13
14     cout<<endl;
15
16     for(int i = 0; i < 10; i++)
17         *ip_array[i] += 2;
18
19     for(int i = 0; i < 10; i++)
20         cout<<*ip_array[i]<<" ";
21
22     for(int i = 0; i < 10; i++)
23         delete ip_array[i];
24
25     return 0;
26 }

```

How Not To Return A Pointer From A Function: Avoiding The Dangling Reference

Do not do this:

```

int* badFunction(){
    int i;
    //...do something with i here
    //...and then later...
    return &i;
}

```

What is happening here is a local function variable, in this case an integer object named `i`, is declared and used inside the function body, and its address is returned. The problem is that since `i` is a local variable there is no telling what will happen to the memory in which `i` resided when `badFunction()` returns. On the other hand, `i` could be declared to be static, and therefore exist across calls to `badFunction()`, but I still do not recommend the practice because it violates loose coupling. Returning pointers in this manner creates what is referred to as a dangling reference. Avoid dangling references!

RETURNING REFERENCES

A function can return a reference. This can be real handy, especially when you start writing class member functions that return references to instance objects. Unfortunately, that material isn't discussed until chapter 11! To tide you over, I will show you a short example of reference returning in action so you get a feel for the mechanics involved.

The following example program will use a function named `getLargestInteger()` to compare two integer objects and return a reference to the largest one. With this reference, the original integer object can be manipulated. Let us start with the function declaration:

```

1  #ifndef GET_LARGEST_INTEGER_H
2  #define GET_LARGEST_INTEGER_H
3
4  int& getLargestInteger(int& a, int& b);
5
6  #endif

```

9.37 *getlargestinteger.h*

The `getLargestInteger()` function takes two integer references as an argument and returns a reference to the object with the largest positive value. Here is the function definition:

```

1  #include "getlargestinteger.h"
2
3  int& getLargestInteger(int& a, int& b){
4      if(a >= b) return a;
5          else return b;
6  }

```

9.38 *getlargestinteger.cpp*

The two integer reference parameters `a` and `b` are compared to each other on line 4 and the corresponding return statement is executed based on the results of the comparison. In this example, multiple return statements make sense and comply with the return statement mantra ancillary. The following `main()` function shows `getLargestInteger()` in use:

```

1  #include <iostream>
2  #include "getlargestinteger.h"
3  using namespace std;
4
5  int main(){
6      int ival1 = 0, ival2 = 1;
7
8      cout<<"The largest number is: "<<getLargestInteger(ival1, ival2)<<endl;
9
10     int& largest_int = getLargestInteger(ival1, ival2);
11
12     largest_int = -8;
13
14     cout<<"The largest number is: "<<getLargestInteger(ival1, ival2)<<endl;
15
16     return 0;
17 }

```

9.39 *main.cpp*

In this example, two integer objects, `ival1`, and `ival2`, are declared and initialized on line 6. `getLargestInteger()` is first called on line 8. The result of the function call then becomes the argument to the insertion operator for the `cout` object. On line 10, the reference returned by `getLargestInteger()` is used to initialize the integer reference `largest_int`. `largest_int` is then used to manipulate the object it references, which, in this case, is `ival2`. With `ival2`'s value now `-8`, `getLargestInteger()` is called again in another `cout` statement on line 14.

Quick Review

Functions can return objects, pointers to objects, or references to objects. The objects can be fundamental data types, user-defined types, or functions. (*You will begin learning about user-defined data types in chapter 10*) Declare the return type in the function declaration and use the `return` keyword to return an object of the specified type from the body of the function declaration. Try to avoid multiple return points from a function. In cases where you have to break this rule keep the multiple return statements as close together as possible for clarity.

FUNCTION OVERLOADING

Multiple functions of the same name can be declared and used in the same program. Functions with the same name but different parameter types or parameter list lengths are said to be overloaded. A function with the same name but different parameter types or parameter list length is said to have a different function signature.

Which of the overloaded functions is actually called is resolved by the compiler based on the types of arguments passed to the overloaded function. Return types play no role in resolving overloaded functions. Let us look at a simple function named `functionA()` that is overloaded in five different ways. Here is the header file:

```

1  #ifndef FUNCTION_A_H
2  #define FUNCTION_A_H
3
4  void functionA();
5  void functionA(int i);
6  void functionA(float f);
7  void functionA(int i, int j);
8  void functionA(char message[]);
9
10 #endif

```

9.40 functiona.h

`functionA()` is overloaded to take no arguments, one integer argument, one float argument, two integer arguments, or a character array. Each version of `functionA()` has a different function signature and will display different behavior when called as you will see by examining the following function definitions:

```

1  #include "functiona.h"
2  #include <iostream>
3  using namespace std;
4
5  void functionA(){
6      cout<<"functionA: no arguments"<<endl;
7  }
8
9  void functionA(int i){
10     cout<<"functionA: int argument = "<<i<<endl;
11 }
12
13 void functionA(float f){
14     cout<<"functionA: float argument = "<<f<<endl;
15 }
16
17 void functionA(int i, int j){
18     cout<<"functionA: two int arguments = "<<i<<" , "<<j<<endl;
19 }
20
21 void functionA(char message[]){
22     cout<<"functionA: char string = "<<message<<endl;
23 }

```

9.41 functiona.cpp

As you can see, each version of the function, when called, will result in a different message being displayed on the screen. The compiler will resolve the issue of which version of `functionA()` to call based on what type of argument appears in the argument list at the time of the function call. Example 9.42 gives a `main()` function showing all five versions of `functionA()` in action:

Figure 9-13 shows the results of running this program.

You will routinely overload functions in your C++ programming career, especially class constructor functions.

```

1  #include <iostream>
2  #include "functiona.h"
3  using namespace std;
4
5  int main(){
6      int ival1 = 1, ival2 = 2;
7      float fval = 25.345;
8      char char_array[] = "\"Hello World!\"";
9
10     functionA();
11     functionA(ival1);
12     functionA(fval);
13     functionA(ival1, ival2);
14     functionA(char_array);
15
16     return 0;
17 }

```

9.42 main.cpp

```

OverLoadedFunction.out.out
functionA: no arguments
functionA: int argument = 1
functionA: float argument = 25.345
functionA: two int arguments = 1, 2
functionA: char string = "Hello World!"
|

```

Figure 9-13: Results of Calling Overloaded Function functionA()

Calling Functions Recursively

In C++ a function can call itself. When it does so it is said to be making a recursive function call.

Functions intended to be called recursively are designed differently from ordinary functions. First, they are written to solve a problem that can be solved in a recursive fashion. An example of a problem of this type is one that can be continually divided into smaller pieces until the smallest piece is reached, then, each of the small pieces of the problem is solved and the solved pieces combined to form the whole solution.

Second, a recursive function has to eventually come to a point where the recursion stops. If a recursive function didn't have this stopping point it would continue to recurse forever. This behavior would eventually overwhelm the resources of the computer on which it was running.

To illustrate the concept of recursion let us examine a simple recursive function that takes an integer argument and recurses based on its value. The name of the function is `countInput()` and its declaration is given below:

```

1  #ifndef COUNT_INPUT_H
2  #define COUNT_INPUT_H
3
4  void countInput(int input);
5
6  #endif

```

9.43 countinput.h

The function definition for `countInput()` is given in example 9.44.

```

1 #include "countinput.h"
2 #include <iostream>
3 using namespace std;
4
5 void countInput(int input){
6     static int count = 0;
7     if(count < (input + count)){
8         cout<<"Still counting!"<<endl;
9         count++;
10        cout<<"The input was: "<<input<<endl;
11        countInput(input - 1);
12    }
13 }

```

9.44 countinput.cpp

Pause for a moment here to study example 9.44. The `countInput()` function is called with an integer argument named `input`. On line 6 a static integer variable named `count` is declared and initialized. The body of the `if` statement is where most of the action takes place. The variable `count` is compared with the result of `(input + count)`. If the comparison is true the body of the `if` statement executes, `count` is incremented, a message is printed to the screen, and the `countInput()` function is called recursively with a new argument, namely, `input - 1`.

The `if` statement represents the recursion stopping point. If the test is true, recursion continues. If the test result is false, then recursion stops. The following `main()` function shows the `countInput()` function in action:

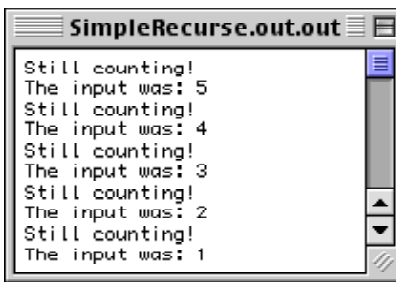
```

1 #include <iostream>
2 #include "countinput.h"
3
4 using namespace std;
5
6 int main(){
7     countInput(5);
8     return 0;
9 }

```

9.45 main.cpp

Figure 9-14 gives the results of running this program:



```

SimpleRecurse.out.out
Still counting!
The input was: 5
Still counting!
The input was: 4
Still counting!
The input was: 3
Still counting!
The input was: 2
Still counting!
The input was: 1

```

Figure 9-14: Results of Running the Simple Recurse Program

ANOTHER EXAMPLE

Although the `countInput()` function demonstrated the concept of recursion, it does not do a very good job of illustrating how recursion can be used to solve a meaningful problem. To address this I would like to show you a recursive sorting function called `quickSort()`.

The quicksort algorithm, developed by C.A.R. Hoare, partitions an input file into two parts and sorts the subparts. Improvements to quicksort have been made here and there by many computer scientists and the implementation I use below can be found in (Sedgewick). A complete analysis of quicksort is beyond the scope of this book but an excellent treatment can be found in both (Sedgewick) and (Knuth).

The following code gives the declaration of quickSort() and a utility function named swap():

```

1 #ifndef QUICK_SORT_H
2 #define QUICK_SORT_H
3
4 void quickSort(int a[], int l, int r);
5 void swap(int a[], int i, int j);
6
7 #endif

```

9.46 quicksort.h

The definitions of both functions appear below:

```

1 #include "quicksort.h"
2
3 void swap(int a[], int i, int j){
4     int temp = a[i];
5     a[i] = a[j];
6     a[j] = temp;
7 }
8
9 void quickSort(int a[], int l, int r){
10     int i, j, temp;
11
12     if(r > l){
13         temp = a[r];
14         i = l-1;
15         j = r;
16         for(;;){
17             while(a[++i] < temp);
18             while(a[--j] > temp);
19             if(i >= j) break;
20             swap(a, i, j);
21         }
22         swap(a, i, r);
23         quickSort(a, l, i-1);
24         quickSort(a, i+1, r);
25     }
26 }

```

9.47 quicksort.cpp

The swap() function is called in the body of the quickSort() function. swap() simply exchanges array elements indicated by the parameters i and j.

The quickSort() function takes as arguments the array of integers to be sorted, the left index value, and the right index value. On the first call to quickSort(), given an array to be sorted of size N, the argument value for the l parameter will be 0, and the argument value for the r parameter will be N-1. Lines 12 through 21 partition the array based on the final resting position of array element a[r]. Line 22 puts element a[r] in its final sorted position and then the two array partitions are then sorted with recursive calls to quickSort(). The following main() function shows the quickSort() function in action:

Figure 9-15 shows the results of running this program:

FUNCTION POINTERS

Function pointers are cool! Just like you can declare a pointer to an object, so too can you declare a pointer to a function. The address of an existing function can then be assigned to and called via the function pointer. There are many great uses for function pointers, as you will see shortly. Also, an understanding of function pointers will help you understand the C++ virtual function call mechanism which is implemented as an array of function pointers. Let

```

1 #include <iostream>
2 #include "quicksort.h"
3 using namespace std;
4
5 int main(){
6     int a[] = {15,200,83,1,22,5,44,77,12,23,99,100,32,64,25,0,40};
7
8     for(int i = 0; i<((sizeof a)/sizeof(int)); i++)
9         cout<<a[i]<<" ";
10    cout<<endl;
11
12    quickSort(a, 0, ((sizeof a)/sizeof(int)));
13
14    for(int i = 0; i<((sizeof a)/sizeof(int)); i++)
15        cout<<a[i]<<" ";
16
17    return 0;
18 }

```

9.48 main.cpp

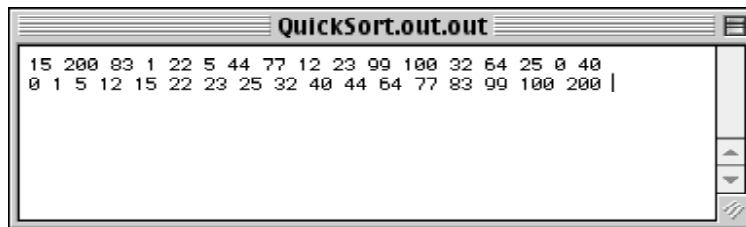


Figure 9-15: Results of Running the QuickSort Program

us start by looking at how function pointers are declared.

DECLARING FUNCTION POINTERS

The function pointer syntax looks a little strange at first glance but it is easy to master. The following code declares a pointer named `fun_ptr` to a function that returns a float and takes two float arguments:

```
float (*fun_ptr)(float, float);
```

This next example declares a function pointer named `sortDirection` that can point to a function that returns a bool and takes two integer arguments:

```
bool (*sortDirection)(int, int);
```

Now that you have a pointer to a type of function, all you need now is a function to point to!

ASSIGNING THE ADDRESS OF A FUNCTION TO A FUNCTION POINTER

Here is a function declaration for a function named `add()` that returns a float and takes two float arguments:

```
float add(float a, float b);
```

Here is the definition for this function:

```
float add(float a, float b){return a+b;}
```


Now armed with a function pointer and a function to point to, here is how you assign the function's address to the function pointer:

```
fun_ptr = add;
```

CALLING THE FUNCTION VIA THE FUNCTION POINTER

After assigning the address of a function to a function pointer you can now call the function via the pointer. The following code gives an example:

```
cout<<fun_ptr(5,5)<<endl;
```

When this line of code executes, the value 10 will be printed to the screen.

ARRAYS OF FUNCTION POINTERS

You can create interesting behavior by building an array of function pointers and assigning different functions to each pointer. The following complete example illustrates how to create an array of function pointers, assign a different function to each array element, and then iterate over the array calling each function via the function pointer. Let us start with the header file that declares four arithmetic functions `add()`, `sub()`, `mul()`, and `div()`:

```
1 #ifndef ARITH_FUNCTIONS_H 9.49 arithfunctions.h
2 #define ARITH_FUNCTIONS_H
3
4 float add(float a, float b);
5 float sub(float a, float b);
6 float mul(float a, float b);
7 float div(float a, float b);
8
9 #endif
```

The following listing gives the definition for each of these functions:

```
1 #include "arithfunctions.h" 9.50 arithfunctions.cpp
2
3 float add(float a, float b){return a+b;}
4 float sub(float a, float b){return a-b;}
5 float mul(float a, float b){return a*b;}
6 float div(float a, float b){return a/b;}
```

The following `main()` creates the array of function pointers and assigns each of these arithmetic functions to an array element:

Line 6 uses the `typedef` keyword to create a function pointer type synonym named `fun_ptr`. The synonym is then used on line 8 to create an array of function pointers named `fun_ptr_array`. Note that each array element can only point to a function of `fun_ptr` type. In other words, each element can only point to functions that return a float and take two float arguments.

On lines 10 through 13 the addresses of the functions `add()`, `sub()`, `mul()`, and `div()` are assigned to each array element. The for loop on line 15 then iterates over `fun_ptr_array` and calls each function via its pointer with two arguments.

Let us now take a look at another interesting use for function pointers — callback functions.

```

1  #include <iostream>
2  #include "arithfunctions.h"
3  using namespace std;
4
5  int main(){
6      typedef float (*fun_ptr)(float, float);
7
8      fun_ptr fun_ptr_array[4];
9
10     fun_ptr_array[0] = add;
11     fun_ptr_array[1] = sub;
12     fun_ptr_array[2] = mul;
13     fun_ptr_array[3] = div;
14
15     for(int i = 0; i < 4; i++)
16         cout<<fun_ptr_array[i](5,5)<<endl;
17     return 0;
18 }

```

9.51 main.cpp

IMPLEMENTING CALLBACK FUNCTIONS WITH FUNCTION POINTERS

It is often desirable to change the behavior of a function by calling the function with a behavior-modifying function as one of its arguments. The behavior-modifying function is referred to as a callback function. The following complete example will revisit a simple sort routine encountered in chapter 4 called `dumbSort`. By rewriting the `dumbSort` function to take a callback function, `dumbSort`'s behavior can be modified so it can sort in ascending or descending order based on what callback function is supplied as an argument when the `dumbSort` function is called. The following code gives the function declaration for `dumbSort()` and two utility functions named `compareAscending()` and `compareDescending()`:

```

1  #ifndef DUMB_SORT_H
2  #define DUMB_SORT_H
3
4  void dumbSort(int a[], int l, int r, bool (*sortDirection)(int, int));
5  bool compareAscending(int a, int b);
6  bool compareDescending(int a, int b);
7
8  #endif

```

9.52 dumbSort.h

Notice how the declaration for `dumbSort()` includes a function pointer parameter named `sortDirection`. The `sortDirection` parameter will take a pointer to either the `compareAscending()` function or the `compareDescending()` function as an argument. The `sortDirection` function pointer is then used in the body of the `dumbSort()` function to provide the comparison of two array elements. The following code gives the definitions for all three functions declared above:

```

1 #include "dumbSort.h"
2
3 bool compareAscending(int a, int b){return a>b;}
4 bool compareDescending(int a, int b){return a<b;}
5
6 void dumbSort(int a[], int l, int r, bool (*sortDirection) (int, int)){
7     for(int i = l; i < r; i++){
8         for(int j = (l+1); j < r; j++){
9             if(sortDirection(a[j-1], a[j])) {
10                int temp = a[j-1];
11                a[j-1] = a[j];
12                a[j] = temp;
13            }
14        }
15    }
16 }

```

9.53 dumbSort.cpp

The `compareAscending()` and `compareDescending()` functions each return the boolean value that results from comparing two integer arguments via the `>` or `<` operators. The `dumbSort()` function takes the `sortDirection` function pointer parameter and calls the supplied function argument in the expression test of the `if` statement on line 9. By using the appropriate callback function, `dumbSort()`'s sorting behavior can be changed to sort in either ascending or descending order. The following `main()` function shows `dumbSort()` and the callback function mechanism in action.

```

1 #include <iostream>
2 #include "dumbSort.h"
3 using namespace std;
4
5 int main(){
6
7     int int_array[10] = {34,3,16,2,8,10,1,0,5,11};
8
9     for(int i = 0; i<10; i++)
10        cout<<int_array[i]<<" ";
11    cout<<endl;
12
13    dumbSort(int_array, 0, 10, compareAscending);
14
15    for(int i = 0; i<10; i++)
16        cout<<int_array[i]<<" ";
17    cout<<endl;
18
19    dumbSort(int_array, 0, 10, compareDescending);
20
21    for(int i = 0; i<10; i++)
22        cout<<int_array[i]<<" ";
23    cout<<endl;
24    return 0;
25 }

```

9.54 main.cpp

On line 7 an array of 10 integers is declared and initialized. It is printed to the screen on lines 9 and 10. On line 13 `dumbSort()` is called to sort the array in ascending order with the `compareAscending()` callback function as an argument. The array is then printed to the screen once again and on line 19 `dumbSort()` is called to again sort the array but this time in descending order using the callback function `compareDescending()`. Figure 9-16 shows the results of

running this program.

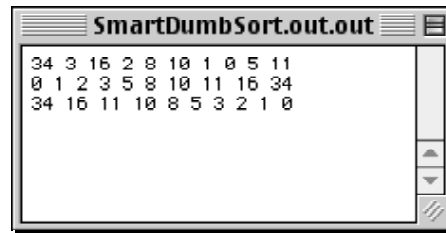


Figure 9-16: Results of Calling DumbSort() Using compareAscending() and compareDescending() CallBack Functions

CREATING A FUNCTION LIBRARY

OK. You have slaved at the computer and have developed a great function that you would like to reuse in other projects without having to recompile its source code. Or perhaps you have developed a set of functions that do something no one has ever seen done on a computer before and you would like to sell them and retire early! Whatever your motivation you will start by creating a function or code library.

A library is compiled code that can be used in other programs. All that's needed to use the library is the library object code and a header file that provides the declarations for the functions in the library. Aren't you lucky! You already know about header files and how to use them. And because you know how to create multiple file programs you already know how to do the hardest part of the library creation process.

STEPS TO CREATING A LIBRARY

The following is a quick overview to creating a function library: The first two steps you already know how to do from reading this chapter. Steps 3 through 8 apply to Metrowerks CodeWarrior but will be similar on other development environments. The final word on how to create a code library can be found in your IDE's documentation. Here is the process at a glance:

- Step 1:** Put the function declarations for any functions you want in the library in a header file. (.h file)
- Step 2:** Put the definitions for the library functions in a separate implementation file. (.cpp file)
- Step 3:** Create an empty project in your Integrated Development Environment and add the implementation file to it.
- Step 4:** Add any library files to the project required to support the implementation file. For instance, if your function uses the iostream library then you need to add that library to your project.
- Step 5:** Set the required target settings for the project.
- Step 6:** Name the library output file and set project type.
- Step 7:** Compile the project.
- Step 8:** Use the library!

That's pretty much it. Now, let us step through the process in more detail. For an example I will create a library using the dumbSort() function along with the two supporting functions compareAscending() and compareDescending(). These three functions will constitute the dumbSort library and after the library is created they can be used in other projects without having to add and recompile the dumbSort.cpp file.

CREATE EMPTY PROJECT

Since you already know how to do the first two steps, creating separate header and implementation files, I will start with creating the empty project. Start your IDE and create a new empty project. An empty project is a project with no source or library files pre-added. Since you're creating a library, all you will need is the implementation file

for the code you want to turn into a library and any libraries your code depends on. Figure 9-17 shows a screen shot for creating an empty project in CodeWarrior:

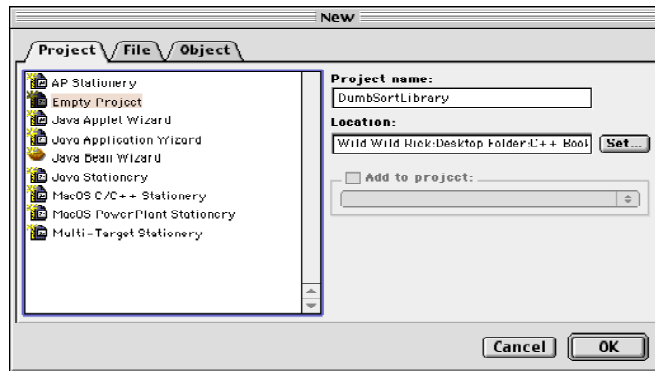


Figure 9-17: Creating an Empty Project in CodeWarrior

Add IMPLEMENTATION FILE

Add to the empty project the implementation file containing the functions of interest. Figure 9-18 shows the DumbSortLibrary project with the dumbsort.cpp implementation file added.

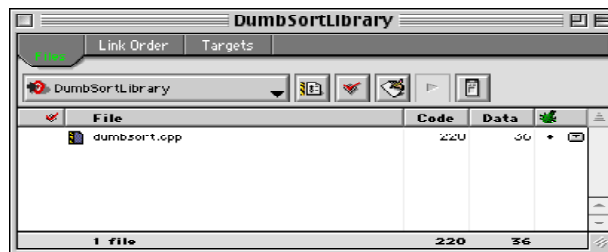


Figure 9-18: dumbsort.cpp Added to the Empty Project

SET LIBRARY TARGET SETTINGS

Now set the target setting for the library as shown in figure 9-19.

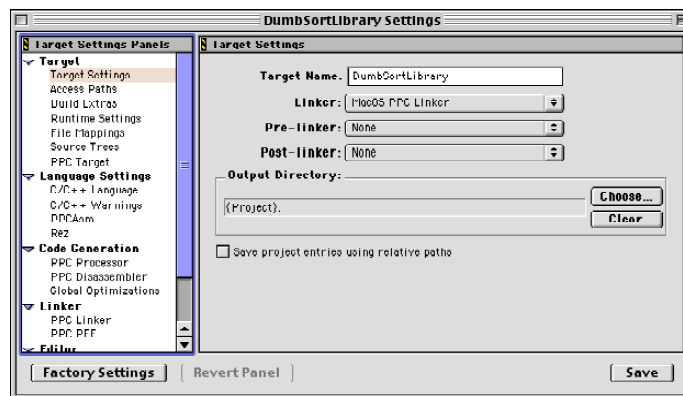


Figure 9-19: Setting Library Target Settings

NAME LIBRARY AND SET PROJECT TYPE

Next, set the name of the library file and the project type. The name shown in 9-20 below is `dumbsort.lib`. Other names for the library could have also been used such as `dumbsort.o`, `dumbsort.out`, or `dumbsortlib.lib`. This name will be used to name the code module produced by the compiler when the project is built.

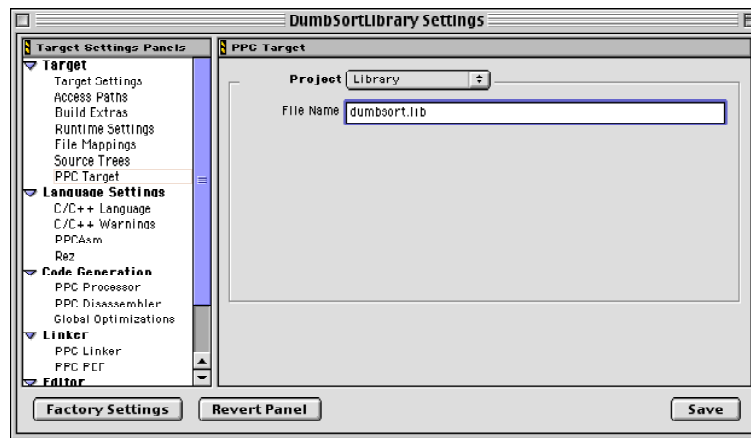


Figure 9-20: Selecting Project Type and Library Name

Build THE PROJECT

Build the project. The library file you named in the previous step will be generated. You will not be able to run this type of project, so, to test your library you will have to create a new project and write some code that uses the functions contained in the library. The following step shows this being done.

Use THE LIBRARY

To use the library you will need the library header file and the library code module. You will use the header file as normal and add the library code module to the project as shown in figure 9-21.

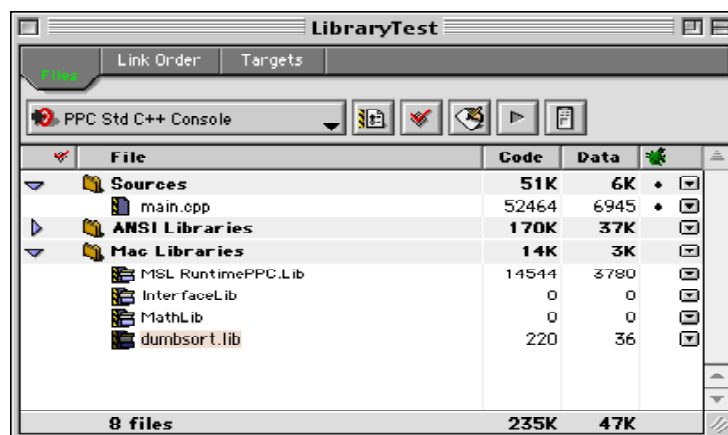


Figure 9-21: Using the dumbsort Library

The contents of the `main()` file in this project is the same as that shown in example 9.54. The difference between the previous `dumbsort` project and this project is that what was once the `dumbsort.cpp` file has been transformed into a code library.

SUMMARY

Functions provide a convenient way to package program behavior into manageable units with an eye towards reuse. A function is a collection of logically related program statements written to perform a specific processing activity. A function is also a code module; it is given a name, and with its name, it can be called or executed by any program that needs to use the function. The program statements that comprise the body of the function give the function its behavior. Function behavior can be built upon the behavior of other functions.

Every function needs to be declared and defined. A function declaration or prototype is a statement of a function's interface. Put function prototypes in separate header files and function definitions in their own implementation files. This separates a function's interface from its implementation and makes it easier to create function libraries. Prevent multiple header file inclusion by using the preprocessor directives `#ifndef`, `#define`, and `#endif`.

A well-written function has the following characteristics: It is maximally cohesive, it is well-named, and it is minimally coupled.

It is helpful to think of a function as a world unto itself or as a “black box”. Variables declared within the body of a function are called local variables. Static function variables exist across function calls, are initialized during the first call to the function in which they appear, and retain their value between function calls. Automatic local variables are initialized during every function call. Local function variables mask global variables with the same name. Function parameters have local scope.

Arguments can be passed to functions by value or by reference. The advantage to passing arguments to functions by reference is realized when using large arguments such as user-defined data structures.

Functions can return values. Where possible have only one return statement in a function. If you must have more than one return statement, keep them close together to aid clarity. Do not return the address of a local variable. This is a common programming error that results in a dangling reference.

Functions can be overloaded. An overloaded function shares a name but differs in parameter type and number. Overloaded functions are said to have different function signatures. The compiler resolves which version of an overloaded function to call based on the number and type of arguments used to call the function.

Recursive functions are written to solve special types of problems. A recursive function calls itself and must eventually come to a halt or else it will recurse forever.

Function pointers have many interesting uses, one of them being to implement callback functions. An understanding of function pointers leads to a better understanding of the C++ virtual function calling mechanism.

When you have come up with a great function, or several great functions, convert them into a library to make their reuse easier

Whew! You learned a lot in this chapter. Great! You will use every bit of it as you progress through the text. Study hard!

Skill Building Exercises

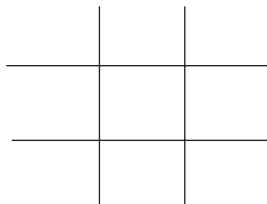
Note: For all exercises create separate header and implementation files. In exercises that result in more than one function, you can group the function declarations in one header file and all function definitions in one implementation file.

1. **Text Message Display:** Write a simple function that displays a text message on the screen when called. Give the function a name that gives a hint of what it does. Write a program that calls the function. Hint: The function should return void and take no arguments.
2. **Print Array Contents:** Write a function that takes a character array as an argument and prints the contents of the character array to the screen. Write a program that calls the function several times using different character arrays as arguments. Hint: If you use an array of chars to hold the message make sure the last character is `'\0'`.

3. **Array Reversal:** Write a function that takes a character array as an argument. Print the array to the screen, then reverse the order of characters in the array and print the array to the screen again. In the same function, count the number of characters in the array and print the count to the screen. Also include a static variable that counts the number of message arrays that have been printed to the screen. Write a program to test the function.
4. **Dynamic Array:** Write a function that takes two character arrays as arguments and creates a dynamically allocated array that contains the elements of both array arguments. Return the address of the dynamically allocated array. Caution: Do not forget to delete[] the array before the program ends. But, do not delete[] it in the function! Write a program to test the function.
5. **Array Parameter:** Write a function that takes a character array as an argument and prints out every permutation of the array. Return the integer value corresponding to the number of permutations that were calculated. Write a program to test the function.
6. **Lotto Number Generator:** Write a function that calculates every possible combination of lottery numbers. The function should take two integer arguments indicating how many numbers there are to choose from and how many numbers are required to be picked. i.e., 52 numbers to choose from, pick 6 out of 52. Return the number of possible lottery ticket combinations. Write a program to test the function.
7. **Sum of all floats:** Write a function that takes an array of floats as an argument and returns the sum of the array. Write a program to test the function.
8. **Converter:** Write a function that takes an integer argument and prints the binary and hexadecimal equivalent values to the screen. Write a program to test the function and convert a few integer values.
9. **Function Pointer:** Write a program that creates a function pointer to the type of function you created in skill building exercise 8. Use the function pointer to call the function.
10. **Research:** Research the steps involved to create a function library using your integrated development environment. Once you have figured it out, create a library from the lottery function you created in skill building exercise 6.

SUGGESTED PROJECTS

1. **Word Counter:** Write a program that reads a text file and counts the occurrence of each word. Print a summary of the statistics to the screen shown what word was found and its number of occurrences.
2. **Game Program:** Write a game program called tic-tac-toe. The object of tic-tac-toe is to get three X's or three O's in a row either horizontally, vertically or diagonally. The tic-tac-toe game board is a 3 x 3 grid and looks something like this:



During a typical game, players takes turns placing their X's and O's on the game board. The first to get their pat-

tern in a row wins. The follow game board represents what the results of a typical game might look like:

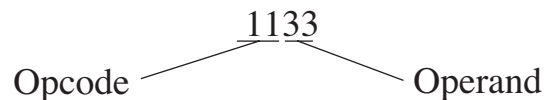
X	O	O
	X	
O		X

Use a 3 x 3 array to track player turns. Check the array after each player's turn to see if they've won. Draw the game board on the screen after each turn so players can see the game's progress. Make it a two player game.

- Quick Sort:** Modify the recursive quickSort() function given in example 9.46 so it will sort in either ascending or descending order based on a callback function.
- Computer Simulator:** You are a C++ developer with a high-tech firm engaged in contract work for the Department of Defense. Your company has won the proposal to develop a proof-of-concept model for an Encrypted Instruction Set Computer Mark I. (EISCS MKI). Your job is to simulate the operation of the EISCS MKI. To do this you must write a small computer simulator *. Here are the specifications:

EISCS MKI LANGUAGE SET

The only language a computer understands is its machine language instruction set, and the EISCS MKI is no different. An EISCS instruction consists of a four digit integer with the two most significant digits being the opcode and the two least significant digits being the operand. For example, the instruction 1133, as shown here,...



...would instruct the computer to write the contents of memory location 33 to the screen. The full set of EISCS opcodes grouped by function is given below:

Input/Output Operations	
READ = 10	Read an integer from the console into a specified memory location
WRITE = 11	Write an integer from specified memory location to the console
Load/Store Operations	
LOAD = 20	Load an integer value from a specified memory location into the accumulator
STORE = 21	Store an integer value from the accumulator into a specified memory location
Arithmetic Operations	
ADD = 30	Add an integer from a memory location to the contents of the accumulator and leave the result in the accumulator
SUB = 31	Subtract an integer from a memory location from the contents of the accumulator and leave the result in the accumulator.
MUL = 32	Multiply an integer from a memory location by the contents of the accumulator and leave the result in the accumulator.

DIV = 33	Divide the contents of the accumulator by an integer from a specified memory location. Leave the results in the accumulator.
Control/Transfer Operations	
BR = 40	Branch to a specified memory location
BRN = 41	Branch to a specified memory location if the contents of the accumulator is negative
BRZ = 42	Branch to a specified memory location if the contents of the accumulator is zero.
HALT = 43	Stop program execution

SAMPLE PROGRAM

Using the EISCS machine language instruction set above you can now write simple programs. Here is an example:

Memory Location	Instructions/ Contents	Action
00	1007	Read integer from input into memory location 07
01	1008	Read integer from input into memory location 08
02	2007	Load integer from memory location 07 into accumulator
03	3308	Divide contents of accumulator by integer from memory location 08
04	2109	Store contents of accumulator in memory location 09
05	1109	Print contents of memory location 09 to the screen
06	4010	Branch to memory location 10
07	0000	
08	0000	
09	0000	
10	4300	Exit program

BASIC OPERATION OF THE EISCS MKI

MEMORY

The machine language instructions that comprise an EISCS program must be loaded into the EISCS's memory before they can be executed. Simulate the EISCS's memory as an array of 100 integers.

INSTRUCTION DECODING

In order to execute programs correctly the EISCS must be able to separate opcodes from operands. Take as an example the instruction located at memory location 00 in the sample program above. the instruction 1007 must be separated into its opcode (READ) and operand (memory location 07). since the EISCS programs must be loaded into memory prior to execution the following statements might be used to extract an instruction from memory and decode it:

```
int instruction, op_code, operand, program_counter = 0;
```

```

instruction = memory[program_counter++];
op_code = instruction / 100;
operand = instruction % 100;

```

Additional EISCS Specifications

Allow for EISCS programs to be loaded into memory from the keyboard or read into memory from a file.

Encrypt the instruction in memory and decrypt them prior to execution by the EISCS. Use an encryption algorithm or your choice.

Write several small programs in the EISCS machine language set and run them on the EISCS.

**This project adapted from the Simpletron exercises 5.18 and 5.19 of Deitel & Deitel's C++ How To Program, Second Edition.*

5. Assembler Program: Write a program that converts a text file containing high level EISCS commands and creates a file with EISCS machine instructions. This type of program is called an assembler. For example, the text file might contain the following instructions:

```

read 07
read 08
load 07
div 08
store 09
write 09
brn 10
halt

```

The assembler program would read each of these high-level instructions and convert them to their EISCS machine instruction equivalent. The machine instructions can then be loaded into the EISCS memory.

6. Assembler Program Modified: The alternative form of the main() function is main(int argc, char * argv[]) which allows command line arguments to be read and acted upon by a program when it is first executed. The parameter argc is the argument count, or the number of command line arguments used to call the program. The parameter argv[] is an array of character strings representing the names of the command line arguments. argv[0] will always contain the name of the command that was executed. (i.e., the name of the program). Convert the assembler program written in project 5 so that the name of the input file and output file can be given as command line arguments.

7. Explore the C and C++ Standard Library: The C and C++ Standard Library provides many useful routines you can use in your programs. Research the standard library and write a brief description of each header and the functions they contain.

8. Scientific Calculator: Write a scientific calculator program using routines from the C++ Standard Library. Keep the interface text based and prompt the user for each operator and operand.

9. Mortgage Calculator: Write a program that calculates the payment schedule of a 30-year, fixed-rate mortgage.

10. Expanded Computer Simulator: Expand the EISCS MKI computer simulator to include the scientific functions you wrote for Project 8. Modify the assembler so it can generate the new machine instructions.

SELF TEST QUESTIONS

1. Describe in your own words the definition of a function.
2. List and describe the three characteristics of a good function.
3. What is the difference between a function's interface and its implementation? What constitutes a function's interface? What constitutes a function's implementation? Why is it important or desirable to separate a function's interface from its implementation?
4. What is the purpose of the `#ifndef`, `#define`, & `#endif` preprocessor directives as they apply to header files?
5. List at least three benefits to giving functions good names.
6. In what two ways can arguments be passed to functions? What is the difference between the two ways? What advantages or disadvantages are associated with each way?
7. What is the difference between an automatic local variable and a static local variable?
8. What is meant by the phrase, "Maximize Cohesion — Minimize Coupling"?
9. Describe how functions can be overloaded.
10. Given the following function pointers describe what type of function each can point to:

```
(void) (*fun_ptr) ();  
(float) (*fun_ptr) (int, char*, float);  
(char*) (*fun_ptr) (float, float);
```

REFERENCES

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Donald E. Knuth. *The Art of Computer Programming. Volume 3. Sorting and Searching*. Second Edition. Addison-Wesley, Reading, Massachusetts, 1998. ISBN:0-201-89685-0

Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, Reading, Massachusetts, 1992. ISBN: 0-201-51059-6

Harvey M. Deitel, Paul J. Deitel. *C++ How To Program*. Second Edition. Prentice Hall, Upper Saddle River, New Jersey, 1997. ISBN: 0-13-528910-6

NOTES

CHAPTER 10



E ON A TREE

TOWARD PROBLEM ABSTRACTION: CREATING NEW DATA TYPES

LEARNING OBJECTIVES

- *CREATE NEW DATA TYPES TO IMPROVE PROBLEM ABSTRACTION*
- *USE THE `typedef` KEYWORD TO CREATE TYPE SYNONYMS FOR EXISTING DATA TYPES BETTER SUITED TO THE PROBLEM DOMAIN*
- *EXPLAIN HOW TYPE SYNONYMS CAN BE USED TO IMPROVE PROGRAM MAINTAINABILITY AND READABILITY*
- *CREATE AND USE ENUMERATED DATA TYPES IN YOUR PROGRAMMING PROJECTS*
- *DESCRIBE THE DEFAULT ENUM STATE VALUES AND EXPLAIN HOW THEY CAN BE CHANGED*
- *EXPLAIN HOW TO RESOLVE ENUM STATE NAME CONFLICTS*
- *CREATE AND USE STRUCTURES IN YOUR PROGRAMMING PROJECTS*
- *EXPLAIN HOW TO USE THE DOT OPERATOR TO ACCESS STRUCTURE AND CLASS ELEMENTS*
- *CREATE AND USE SIMPLE CLASSES IN YOUR PROGRAMMING PROJECTS*
- *STATE THE DIFFERENCE BETWEEN STRUCTURES AND CLASSES*
- *DESCRIBE WHEN YOU WOULD WANT TO USE STRUCTURES VS. CLASSES IN A PROGRAMMING PROJECT*
- *LIST THE KEY DIFFERENCES BETWEEN STRUCTURES AND CLASSES*
- *STATE THE PURPOSE AND USE OF THE `this` POINTER*
- *LIST AND DEFINE THE FOLLOWING TERMS: CLASS, BASE CLASS, DERIVED CLASS, SUPERCLASS, SUBCLASS, ABSTRACT BASE CLASS, VIRTUAL FUNCTION, OBJECT, MESSAGE PASSING, OOA&D, INHERITANCE, DATA ENCAPSULATION, INTERFACE, & IMPLEMENTATION*

INTRODUCTION

A computer program consists of data and instructions that manipulate data. How data is modeled in source code is the primary topic of this chapter.

The C++ programming language provides facilities for creating your own data types. Up to now you have only been exposed to the fundamental data types C++ has to offer like `char`, `int`, and `float`. You can further abstract the problem you're trying to solve by renaming these fundamental data types using the `typedef` (type definition) keyword. You saw an example of this being done in the function pointer section of chapter 9.

You can also create your own data types using the `enum` (enumeration), `struct` (structure), and `class` keywords. The `enum` keyword lets you create enumerated data types. By using enumerated types you can add a level of clarity to your program difficult to achieve otherwise. The `struct` and `class` keywords let you create complex, custom data types best suited to model the problem you are trying to solve.

In this chapter you will learn how to use `typedefs`, enumerations, and structures in your programs. You will also be introduced to classes and learn the fundamental differences between structures and classes.

This is an important chapter! After learning the material in this chapter you will know enough C++ to start solving real-world problems. You will also be prepared to approach the detailed study of classes and object-oriented design methodologies presented in subsequent chapters.

TOWARD DATA ABSTRACTION: `typedef`

When you write a program you will want to represent the data you wish to manipulate in such a way that makes it clear to you and to others what you are doing. The data also should be modeled in a manner that best suits the problem at hand. And because you are modeling a real-world problem in a computer program you will have to select data elements of the real-world problem and map them into the program you are writing. If you limit yourself to the fundamental data types C++ offers this would be a difficult task. Luckily you're not limited!

The first facility C++ provides to help with data abstraction is the type definition. The `typedef` keyword lets you declare an alternate name for an existing data type. This alternate name functions like a type synonym. There are several good reasons for creating type synonyms using `typedef`.

First, it increases program portability by allowing you to substitute appropriate machine-dependent data types when you compile your program on different computers. A good explanation of using `typedef` in this manner can be found in [Kernighan]. Second, you can use `typedef` to tame complicated type declarations. You saw an example of `typedef` used in this way in chapter 9's section on function pointers.

Another good reason for using `typedef` is better program readability. By declaring type synonyms for existing data types you can more closely match their purpose in your program. This leads to a significant improvement in your program's aesthetics. Let us take a closer look at the use of `typedef` to accomplish these last two objectives: taming complex data types and improving program aesthetics.

CREATING TYPE SYNONYMS

The following examples show `typedef` being used to create a synonym for an existing data type:

```
typedef float Currency;
typedef int Hours;
typedef char* String
```

The first example declares a type synonym for the `float` data type called `Currency`. `Currency` is now just another name for a `float`. The second example declares a synonym for `int` named `Hours`, and the third example declares a synonym for a `char` pointer named `String`. These synonyms can now be used in a program.

I have adopted a synonym naming convention here that capitalizes the first letter of the identifier as to make it distinguishable from a variable. You can place `typedef` declarations in header files. The following example program shows `typedef` in action starting with the header file that contains the `typedef` declarations:

```

1  #ifndef MY_DEFS_H
2  #define MY_DEFS_H
3
4  typedef float Currency;
5  typedef int Hours;
6
7  #endif

```

10.1 mydefs.h

The header file `mydefs.h` can now be included wherever the type synonyms `Currency` and `Hours` could be used to improve the clarity of the code. The following header file uses the declarations in `mydefs.h` to declare a function named `calculatePay()`:

```

1  #ifndef CALCULATE_PAY_H
2  #define CALCULATE_PAY_H
3  #include "mydefs.h"
4
5  Currency calculatePay(Hours hoursWorked, Currency payPerHour);
6
7  #endif

```

10.2 calculatepay.h

The `calculatePay()` function is then defined in the normal way:

```

1  #include "mydefs.h"
2  #include "calculatepay.h"
3
4  Currency calculatePay(Hours hoursWorked, Currency payPerHour){
5      return (hoursWorked * payPerHour);
6  }

```

10.3 calculatepay.cpp

The following `main()` function shows `calculatePay()` in action:

```

1  #include <iostream>
2  #include "calculatepay.h"
3  #include "mydefs.h"
4
5  using namespace std;
6
7  int main(){
8      Hours employeeWorkHours = 0;
9      Currency employeePayRate = 0.0;
10
11     cout<<"Please enter hours worked: ";
12     cin>>employeeWorkHours;
13     cout<<endl<<"Please enter employee hourly pay rate: ";
14     cin>>employeePayRate;
15
16     cout<<endl<<"The employee's pay is: "
17     <<calculatePay(employeeWorkHours, employeePayRate)<<endl;
18
19     return 0;
20 }

```

10.4 main.cpp

CREATING ENUMERATED DATA TYPES WITH ENUM

When programming, you will encounter many situations where code clarity can be improved by referring to integer values by a meaningful name rather than just the integer value itself. Enter the enumerated data type. An enumerated data type is a new type that you declare and assign possible integer value states that are referred to by name. Let us look at an example:

```
enum EyeColor {Black, Hazel, Blue, Brown};
```

This line of code declares a new data type called `EyeColor` that has four possible states: `Black`, `Hazel`, `Blue`, and `Brown`. The state names `Black`, `Hazel`, `Blue`, and `Brown` equate to the integer values 0, 1, 2, and 3 respectively. The type name `EyeColor` can now be used to declare a variable that can hold one of the four possible `EyeColor` state values:

```
EyeColor my_eye_color;
```

The variable `my_eye_color` can be assigned any one of the following values: `Black`, `Hazel`, `Blue`, or `Brown`. Observe the following line of code:

```
my_eye_color = Brown;
```

The nice thing about enum types is that the compiler will ensure you are only assigning one of the valid state values to the enum variable. For instance, the following line of code will produce a compiler error:

```
my_eye_color = 6; //Error...6 is an int not an EyeColor
```

ENUMS AND SWITCH STATEMENTS

Enumerated types can be used with switch statements to improve their readability. Observe the following example:

```

1  switch(my_eye_color) {
2      case Brown: cout<<"You have brown eyes!"<<endl;
3                  break;
4      case Black: cout<<"You have black eyes!"<<endl;
5                  break;
6      case Hazel: cout<<"You have hazel eyes!"<<endl;
7                  break;
8      case Blue:  cout<<"You have blue eyes!"<<endl;
9                  break;
10     default : cout<<"You have blue eyes!"<<endl;
11 }

```

10.5 switch statement

In this example, the value of `my_eye_color` is examined and the appropriate case statement is executed. The switch statement is rendered easier to read because the `EyeColor` state value names are used in each case statement. You no longer have to think in terms of 0, 1, 2, or 3. You are comparing the `my_eye_color` to `Black`, `Hazel`, `Blue` and `Brown`. As this switch statement shows, the order in which you perform the comparison is immaterial.

CHANGING AN ENUM'S DEFAULT STATE VALUES

The enum `EyeColor`'s states `Black`, `Hazel`, `Blue`, and `Brown` are assigned the integer values 0, 1, 2, and 3 by default. Since the name `Black` appears first in the list of possible states between the curly braces, its value is, by default, set to zero. The value of any enum state can be explicitly set in the following manner:

```
enum EyeColor {Black = 1, Hazel, Blue, Brown};
```

In this example, the EyeColor state Black has a value of 1. Each subsequent state will have the next integer value unless explicitly set in similar fashion. For instance, Hazel has the value 2, Blue has the value 3, and Brown has the value 4.

Although the values associated with the EyeColor enum states may not be important, there are cases where an enum's state values might be a concern. Examine the following enum declaration:

```
enum BeveragePackage {Single = 1, SixPack = 6; TwelvePack = 12, EconoPack = 24};
```

In this example the enum BeveragePackage has four states, each with a non-default, non-consecutive value.

ENUM STATE NAME CONFLICTS

Two different enumerated types declared within the same namespace that contain one or more identical state names will cause a compiler error. Examine the following code:

```
enum EyeColor {Black, Hazel, Blue, Brown};
enum HairColor {Black, Blond, Red, Brown}; //error
```

If these two enum declarations appeared in the same header file the compiler would generate an error because EyeColor has two states, Black and Brown, that also appear in HairColor. This can be fixed by changing the case of the first letter of either of the enum state names. For example, changing the state names in HairColor to begin with lowercase letters removes the conflict:

```
enum EyeColor {Black, Hazel, Blue, Brown};
enum HairColor {black, blond, red, brown}; //OK
```

Although this works, it is not the preferred resolution. I recommend instead putting each enum declaration in its own namespace as shown in the following example:

10.6 namespaces

```
1 namespace EyeColor{
2     enum EyeColor {Black, Hazel, Blue, Brown};
3 }
4
5 namespace HairColor{
6     enum HairColor {Black, Blond, Red, Brown};
7 }
```

Putting enum declarations in separate namespaces is like wrapping them in a cocoon. With each enum declaration in its own namespace the state names can safely begin with capital letters. To access each enum and state name, prefix each with its associated namespace using the scope resolution operator as shown in the following example:

```
HairColor::HairColor my_hair_color = HairColor::Black;
EyeColor::EyeColor my_eye_color = EyeColor::Black;
```

Example 10.7 shows the switch statement of example 10.5 rewritten to use the EyeColor namespace:

The Utility of NAME SPACES

Namespaces are good places to put related declarations and attributes to prevent identifier name collisions with identical identifier names in the global namespace. The topic of namespaces will arise again in the study of structs and classes below.

```

1  switch(my_eye_color) {
2      case EyeColor::Brown: cout<<"You have brown eyes!"<<endl;
3                          break;
4      case EyeColor::Black: cout<<"You have black eyes!"<<endl;
5                          break;
6      case EyeColor::Hazel: cout<<"You have hazel eyes!"<<endl;
7                          break;
8      case EyeColor::Blue: cout<<"You have blue eyes!"<<endl;
9                          break;
10     default : cout<<"You have blue eyes!"<<endl;
11 }

```

10.7 switch statement

Quick SUMMARY

Typedefs let you create synonyms for existing data types. A typedef synonym is not a new type, just a nickname by which the typedefed name can be referenced. Typedefs can make your code easier to read and maintain. Enums are new types that can take on a range of integer values represented by names rather than numbers. When a variable of an enum type is declared, the compiler will check to ensure no other values are assigned to that variable other than those authorized. Enums can make your source code easier to read and help prevent programming errors by catching mistakes at compile time.

STRUCTURES: C-Style

Structures are a double-edged sword in C++. Some authors do not bother to give structures much coverage because they are closely related to classes in capability, as you will soon see. I take the opposite approach because I can foresee times when a C++ programmer will be tasked with either maintaining C code or converting legacy C code to C++. When these situations arise it is helpful to know the procedural mind set behind the use of structs.

In C, a structure is a set of heterogeneous data elements grouped together to form a new data type. The C struct mind set separates the structure from the functions that manipulate structure elements. It goes something like this: declare a structure and put in it any required data elements, then, declare and write some functions that can be used to manipulate the structure's data elements. The C way of thinking of structures is decidedly procedural.

Let us examine an extended C-style example that declares a struct named Person and some functions that manipulate Person data elements. Example 10.8 gives the source code for the personstruct.h header file.

```

1  #ifndef __PERSON_STRUCT_H
2  #define __PERSON_STRUCT_H
3
4  namespace HairColor{
5      enum HairColor {Black, Red, Auburn, Brown, Blond, Silver, Grey};
6  }
7
8  namespace EyeColor{
9      enum EyeColor {Black, Hazel, Blue, Brown};
10 }
11
12 const int NAMESIZE = 26;
13
14 struct Person {
15     char f_name[NAMESIZE];
16     char l_name[NAMESIZE];
17     HairColor::HairColor hair_color;
18     EyeColor::EyeColor eye_color;
19 };
20 #endif

```

10.8 personstruct.h

In example 10.8 two name spaces are declared that contain enum type declarations. You have seen these in previous examples above. On line 12 a constant named NAMESIZE is declared and initialized to the value 26. NAMESIZE is used to set the size of each character array declared in the body of the Person structure.

The declaration of the Person structure begins on line 14. The Person structure contains four elements: two character arrays named f_name and l_name to contain a person's first name and last name respectively, a hair_color attribute of type HairColor::HairColor and an eye_color attribute of type EyeColor::EyeColor. The data elements of Person represent the attributes of a particular person.

Before moving on I would like to show you how to access structural elements.

ACCESSING STRUCTURAL ELEMENTS

There are two ways to access the data elements appearing inside of a structure. If you have an ordinary variable of a struct data type you can use the dot "." operator. If you have a pointer to a struct object then you can use either the dot operator applied to a dereferenced pointer or the shorthand member access "->" operator. Let us examine each of these structural member access methods a little closer.

ACCESSING STRUCTURAL DATA MEMBERS VIA THE DOT "." OPERATOR

Given a variable of type Person...:

```
Person Bill;
```

...you can access Bill's f_name element using the dot operator like so:

```
Bill.f_name
```

Having access to each of Bill's elements in this manner offers you the ability to manipulate Bill in just about any way you need to as shown in the following code:

```

1  strcpy(Bill.f_name, "Bill");
2  strcpy(Bill.l_name, "Smith");
3  Bill.eye_color = EyeColor::Blue;
4  Bill.hair_color = HairColor::Blond;
5
6  cout<<"First Name: "<<Bill.f_name<<endl;
7  cout<<" Last Name: "<<Bill.l_name<<endl;
8  cout<<"Hair Color: "<<Bill.hair_color<<endl;
9  cout<<"Eye Color: "<<Bill.eye_color<<endl;
```

10.9 accessing struct elements

This code, when appearing inside a main() function, should produce the results shown in figure 10-1:

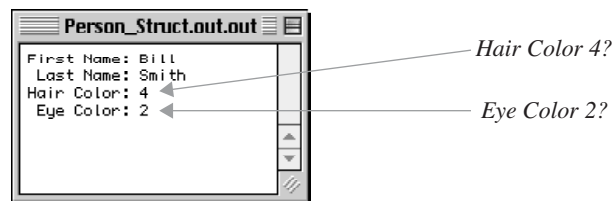


Figure 10-1: Example 10.9 Output

Wait a second! Notice the output shown for hair color and eye color in figure 10-1. This highlights a slight problem with enumerated data types; they are simply integer values. If your compiler gives you a problem compiling the code shown in example 10.9 try adjusting the C++ language settings so enums are treated as ordinary integers. You may have to do this if you have problems sending enum variables or elements to the output stream. The C++ language settings window for CodeWarrior is shown in figure 10-2.

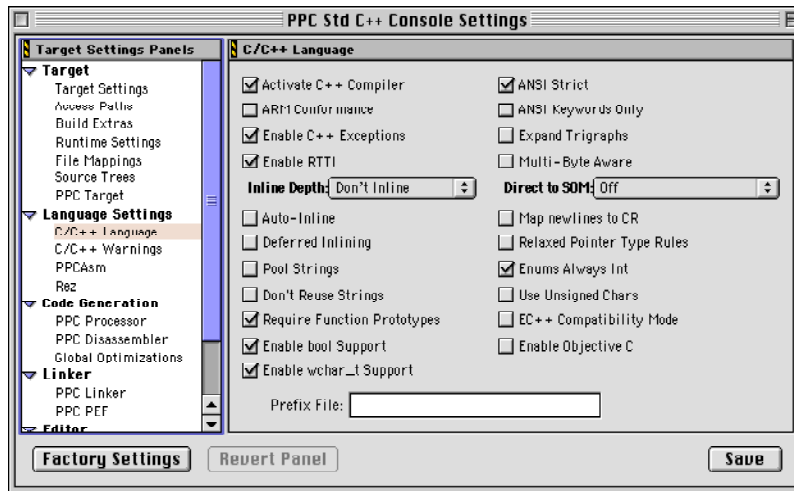


Figure 10-2: C++ Language Settings: Set Enums Always int

ACCESSING STRUCTURAL ELEMENTS VIA THE SHORTHAND MEMBER ACCESS “->” OPERATOR

You can use the “->” operator to access structural elements via pointers. Examine the following code:

```

1  Person* Bill = new Person;
2
3  strcpy(Bill->f_name, "Bill");
4  strcpy(Bill->l_name, "Smith");
5  Bill->eye_color = EyeColor::Blue;
6  Bill->hair_color = HairColor::Blond;
7
8  cout<<"First Name: "<<Bill->f_name<<endl;
9  cout<<" Last Name: "<<Bill->l_name<<endl;
10 cout<<"Hair Color: "<<Bill->hair_color<<endl;
11 cout<<" Eye Color: "<<Bill->eye_color<<endl;
12
13 delete Bill;

```

*10.10 accessing
struct elements via
pointers*

On line 1 an object of type `Person` is dynamically allocated from the heap. Using the pointer `Bill`, you can access `Bill`'s data members using the short-hand member access operator “->” as is shown throughout the rest of the program. An alternative to the short-hand access operator is the long way, which is simply dereferencing the pointer and applying the dot operator like so:

```
strcpy((*Bill).f_name, "Bill"); //The long way
```

Continuing with the C-style way of structure programming, the following header file declares several functions that will be used to make manipulating `Person` objects slightly easier. Example 10.11 shows a header file named `personfunctions.h` that contains eight functions specifically designed to manipulate `Person` structural elements. Example 10.12 shows the implementation file for these functions named `personfunctions.cpp`.

```

1 #ifndef __PERSON_FUNCTIONS_H
2 #define __PERSON_FUNCTIONS_H
3 #include "personstruct.h"
4
5 void setFirstName(Person& person, char* f_name);
6 void setLastName(Person& person, char* l_name);
7 void setHairColor(Person& person, HairColor::HairColor hair_color);
8 void setEyeColor(Person& person, EyeColor::EyeColor eye_color);
9 char* getFirstName(Person& person);
10 char* getLastName(Person& person);
11 char* getHairColor(Person& person);
12 char* getEyeColor(Person& person);
13
14 #endif

```

10.11 personfunctions.h

```

1 #include "personstruct.h"
2 #include "personfunctions.h"
3 #include "string.h"
4
5 char HairColorStrings[][7] = {"Black", {"Red"}, {"Auburn"}, {"Brown"}, {"Blond"},
6                               {"Silver"}, {"Grey"}};
7 char EyeColorStrings[][7] = {"Black"}, {"Hazel"}, {"Blue"}, {"Brown"}};
8
9 void setFirstName(Person& person, char* f_name){
10     strcpy(person.f_name, f_name);
11 }
12
13 void setLastName(Person& person, char* l_name){
14     strcpy(person.l_name, l_name);
15 }
16
17 void setHairColor(Person& person, HairColor::HairColor hair_color){
18     person.hair_color = hair_color;
19 }
20
21 void setEyeColor(Person& person, EyeColor::EyeColor eye_color){
22     person.eye_color = eye_color;
23 }
24
25 char* getFirstName(Person& person){
26     return person.f_name;
27 }
28
29 char* getLastName(Person& person){
30     return person.l_name;
31 }
32 char* getHairColor(Person& person){
33     char* hair_color = NULL;
34     switch(person.hair_color){
35         case HairColor::Black : hair_color = HairColorStrings[HairColor::Black];
36                                 break;
37         case HairColor::Red : hair_color = HairColorStrings[HairColor::Red];
38                                 break;
39         case HairColor::Auburn : hair_color = HairColorStrings[HairColor::Auburn];
40                                 break;
41         case HairColor::Brown : hair_color = HairColorStrings[HairColor::Brown];
42                                 break;
43         case HairColor::Blond : hair_color = HairColorStrings[HairColor::Blond];
44                                 break;
45         case HairColor::Silver : hair_color = HairColorStrings[HairColor::Silver];
46                                 break;
47         case HairColor::Grey : hair_color = HairColorStrings[HairColor::Grey];
48                                 break;
49         default : hair_color = HairColorStrings[HairColor::Blond];
50     }
51     return hair_color;
52 }

```

10.12 personfunctions.cpp

```

53 char* getEyeColor(Person& person){
54     char* eye_color = NULL;
55     switch(person.eye_color){
56         case EyeColor::Black : eye_color = EyeColorStrings[EyeColor::Black];
57                               break;
58         case EyeColor::Hazel : eye_color = EyeColorStrings[EyeColor::Hazel];
59                               break;
60         case EyeColor::Blue  : eye_color = EyeColorStrings[EyeColor::Blue];
61                               break;
62         case EyeColor::Brown : eye_color = EyeColorStrings[EyeColor::Brown];
63                               break;
64         default : eye_color = EyeColorStrings[EyeColor::Blue];
65     }
66     return eye_color;
67 }

```

10.12 Continued

Referring to example 10.12, on lines 5 and 7 two arrays of strings are declared and initialized to hold the string representations of both EyeColor and HairColor. These arrays are used, along with their related enumerated types, in the last two functions to render a Person object's hair_color and eye_color attribute into a string representation.

The functions in example 10.12 can be classified two ways: functions that change or set a person's attributes, and functions that simply return the value of a person's attributes. Functions that change or manipulate structural data members are referred to as mutator functions; functions that simply return the state of a data member are referred to as accessor functions.

The setFirstName() function takes the first argument, a reference to a Person object named person, and sets person's f_name element to the string of characters pointed to by the second argument. Notice how setFirstName() is simply calling the strcpy() library function. The strcpy() function is a standard C++ library functions whose declaration can be found in the string.h header file. The rest of the set functions operate in similar fashion, although setHairColor() and setEyeColor() need only make a simple assignment to complete their mission.

The getFirstName() and getLastName() functions return pointers to the person object's f_name and l_name arrays. The two functions getHairColor() and getEyeColor() are somewhat more involved. Both getHairColor() and getEyeColor() operate the same so only the getHairColor() function will be discussed in detail.

The getHairColor() function takes a reference to a person object and in the switch evaluation section checks the state of the hair_color attribute and compares it with each of the HairColor::HairColor states. When the matching case is found, the local char pointer variable hair_color is set to point to the proper string located in the HairColorStrings array. Assume for a moment that the person's hair_color was Black. The first case statement would apply and the local hair_color variable would be set to point to the string "Black" located in the HairColorStrings array by using the enumeration state HairColor::Black as an offset value into the array. The main() function in example 10.13 shows these functions in action on several person objects.

Referring to example 10.13, on line 7 a Person variable named Bob is declared. On lines 8 through 11 each of the set functions is called using Bob as the first argument to set each of Bob's attributes. Bob's attributes are then printed to the screen on lines 12 through 15 using the get functions.

On line 18 another Person object is created dynamically and its address is assigned to the Person pointer Bill. Since Bill is a pointer the "*" operator must be used on Bill to gain access to what Bill points to, namely, a Person object in dynamic memory. Otherwise, all the functions perform the same tasks on Bill as they do on Bob.

Quick SUMMARY

C-style structures provide a way to collect related data elements together and create a new composite data type. The Person structure can be thought of as stamp or a mold with which new Person objects can be created. Every Person object created will have four attributes: f_name, l_name, hair_color, and eye_color.

There are two primary ways to access structure data elements. When dealing with objects, use the dot "." operator. When dealing with pointers to objects use the shorthand member access "->" operator and let the compiler do the pointer dereferencing for you.

The C-style way of using structures is based on procedural programming techniques. In the procedural programming paradigm, data structures are defined separately from the functions used to manipulate or access those data structures.

```

1  #include <iostream>
2  #include "personstruct.h"
3  #include "personfunctions.h"
4  using namespace std;
5
6  int main(){
7      Person Bob;
8      setFirstName(Bob, "Bob");
9      setLastName(Bob, "Smith");
10     setHairColor(Bob, HairColor::Red);
11     setEyeColor(Bob, EyeColor::Blue);
12     cout<<"First Name: "<<getFirstName(Bob)<<endl;
13     cout<<" Last Name: "<<getLastName(Bob)<<endl;
14     cout<<"Hair Color: "<<getHairColor(Bob)<<endl;
15     cout<<" Eye Color: "<<getEyeColor(Bob)<<endl;
16     cout<<endl<<endl;
17
18     Person* Bill = new Person;
19     setFirstName(*Bill, "Bill");
20     setLastName(*Bill, "Jones");
21     setHairColor(*Bill, HairColor::Blond);
22     setEyeColor(*Bill, EyeColor::Brown);
23     cout<<"First Name: "<<getFirstName(*Bill)<<endl;
24     cout<<" Last Name: "<<getLastName(*Bill)<<endl;
25     cout<<"Hair Color: "<<getHairColor(*Bill)<<endl;
26     cout<<" Eye Color: "<<getEyeColor(*Bill)<<endl;
27     delete Bill;
28
29     return 0;
30 }

```

10.13 main.cpp

STRUCTURES: C++-STYLE

In the previous section I introduced you to structures and showed you an example of using structures with a C-style, procedural mind set. In this section I want to show you what structures can do in C++. This section marks the beginning of your initiation into the world of object-oriented programming.

In C++, structures are more than just a collection of heterogeneous data elements. Structures can contain both data elements and the functions that manipulate those data elements. Let us revisit the Person structure and redesign it in C++ fashion, combining both the data elements and the functions needed to manipulate those data elements into one structure.

PERSON STRUCTURE REDESIGN

Interestingly enough, it will be much easier to completely redesign the functions rather than copy and paste them from the previous project so that's what I will do. Example 10.14 gives the revised header file personstruct.h.

OK, what is going on here? This version of the Person struct looks a lot different from its C-style counterpart. Namely, both structure data elements like the `f_name` and `l_name` arrays, `hair_color` and `eye_color`, and functions, are all part of the structure. This forces the functions to be slightly redesigned. No longer are the functions separate from the data structure they are intended to manipulate, so, there is no need to supply a reference to a Person object.

Structure functions have complete access to all structure attributes. Because the functions have access to Person data elements the function parameter names that remain were changed to make them unique. For example, in the `setFirstName()` function, an underscore “_” was added to the parameter `f_name` to make it `_f_name`. Making member function parameter names unique eliminates potential name conflicts between structural data members and member function parameter names.

Public INTERFACE FUNCTIONS AND THE Public Access Specifier

With the addition of the functions to Person struct comes the notion of an interface to Person objects. The functions represent operations that can be performed on Person objects. The functions have been collectively declared as being publicly accessible through the use of the public access specifier appearing on line 15 in example 10.14. C++ structures have public accessibility by default so the use of the public access specifier at the top of the structure is

10.14 personstruct.h

```

1  #ifndef __PERSON_STRUCT_H
2  #define __PERSON_STRUCT_H
3
4  namespace HairColor{
5      enum HairColor {Black, Red, Auburn, Brown, Blond, Silver, Grey};
6  }
7
8  namespace EyeColor{
9      enum EyeColor {Black, Hazel, Blue, Brown};
10 }
11
12 const int NAMESIZE = 26;
13
14 struct Person {
15     public:
16         void setFirstName(char* _f_name);
17         void setLastName(char* _l_name);
18         void setHairColor(HairColor::HairColor _hair_color);
19         void setEyeColor(EyeColor::EyeColor _eye_color);
20         char* getFirstName();
21         char* getLastName();
22         char* getHairColor();
23         char* getEyeColor();
24
25     private:
26         char f_name[NAMESIZE];
27         char l_name[NAMESIZE];
28         HairColor::HairColor hair_color;
29         EyeColor::EyeColor eye_color;
30 };
31 #endif

```

redundant but a good documentation technique. The set of public interface functions is intended to be the only authorized way to set, manipulate, or otherwise access any object's attributes.

PRIVATE DATA MEMBERS AND THE PRIVATE ACCESS SPECIFIER

Since the public interface functions are going to be the only authorized form of access into the internals of a Person object, Person data elements have been declared as being private through the use of the private access specifier. A private data element is accessible only from within the structure itself. This means that all the functions have access to the private data elements but no data element can be accessed directly from outside an object. Observe the following example:

```

Person Bob;
Bob.setFirstName("Bob"); //OK...Public function
cout<<Bob.f_name<<endl; //Error! Private attribute

```

These few lines of code teach us a lot. First, a Person object named Bob is declared. Next, because the setFirstName() function is a part of the Person structure, it is called on an object of type Person rather than with an object of type Person as an argument, as was done in the C-style example. Since the setFirstName() function is declared as being public, its use on a Person object is authorized. However, since f_name is declared as being private, its use in this fashion is unauthorized and your compiler will produce an error stating that you have attempted to access a private data member. Now that you have seen the personstruct.h header file let us take look at the implementation file, shown in example 10.15.

Referring to example 10.15, compare example 10.15 against example 10.12. The string arrays declared at the beginning of the file remain unchanged from the previous version. The format of each function has changed somewhat to reflect the fact that the functions are now declared inside the Person structure namespace. The format for defining a function declared inside a structure is shown in figure 10-3.

Since the setFirstName() function belongs to the Person structure, it has direct access to Person's f_name data element. All it needs to do is call the strcpy() function using Person's f_name and its _f_name parameter. This is where things start to get somewhat confusing, so hang on. To what Person object does f_name belong? Well, it depends upon what object the function was called.

10.15 personstruct.cpp

```

1  #include "personstruct.h"
2  #include <string.h>
3
4  char HairColorStrings[][7] = {"Black"}, {"Red"}, {"Auburn"}, {"Brown"},
5                               {"Blond"}, {"Silver"}, {"Grey"};
6
7  char EyeColorStrings [] [7] = {"Black"}, {"Hazel"}, {"Blue"}, {"Brown"};
8
9  void Person::setFirstName(char* _f_name){
10     strcpy(f_name, _f_name);
11 }
12
13 void Person::setLastName(char* _l_name){
14     strcpy(l_name, _l_name);
15 }
16
17 void Person::setHairColor(HairColor::HairColor _hair_color){
18     hair_color = _hair_color;
19 }
20
21 void Person::setEyeColor(EyeColor::EyeColor _eye_color){
22     eye_color = _eye_color;
23 }
24
25 char* Person::getFirstName(){
26     return f_name;
27 }
28
29 char* Person::getLastName(){
30     return l_name;
31 }
32
33 char* Person::getHairColor(){
34     char* h_color = NULL;
35     switch(hair_color){
36         case HairColor::Black : h_color = HairColorStrings[HairColor::Black];
37                                 break;
38         case HairColor::Red : h_color = HairColorStrings[HairColor::Red];
39                                 break;
40         case HairColor::Auburn : h_color = HairColorStrings[HairColor::Auburn];
41                                 break;
42         case HairColor::Brown : h_color = HairColorStrings[HairColor::Brown];
43                                 break;
44         case HairColor::Blond : h_color = HairColorStrings[HairColor::Blond];
45                                 break;
46         case HairColor::Silver : h_color = HairColorStrings[HairColor::Silver];
47                                 break;
48         case HairColor::Grey : h_color = HairColorStrings[HairColor::Grey];
49                                 break;
50         default : h_color = HairColorStrings[HairColor::Blond];
51     }
52     return h_color;
53 }
54
55 char* Person::getEyeColor(){
56     char* e_color = NULL;
57     switch(eye_color){
58         case EyeColor::Black : e_color = EyeColorStrings[EyeColor::Black];
59                                 break;
60         case EyeColor::Hazel : e_color = EyeColorStrings[EyeColor::Hazel];
61                                 break;
62         case EyeColor::Blue : e_color = EyeColorStrings[EyeColor::Blue];
63                                 break;
64         case EyeColor::Brown : e_color = EyeColorStrings[EyeColor::Brown];
65                                 break;
66         default : e_color = EyeColorStrings[EyeColor::Blue];
67     }
68     return e_color;
69 }

```

A structure name is a namespace...

Do not forget the scope resolution operator!

```
return_type struct_name::function_name(parameters, if any){
    //Function Body
}
```

Figure 10-3: Format of Structure Function Definition

Observe the following code:

```
Person Bob, Bill;
Bob.setFirstName("Bob"); //called on the Bob object
Bill.setFirstName("Bill"); //called on the Bill object
```

Here, two Person objects are declared, Bob and Bill. Next, setFirstName() is called on the Bob object, and Bob's f_name array is initialized to the string "Bob". Then, the same function is called on the Bill object and the same operation is performed, only on Bill's f_name array instead of Bob's.

THE DEEP SECRET: THE THIS POINTER

Well, it is not really a secret but I got your attention didn't I? When objects are created from structures there is a different set of data elements for each object but only one set of functions. The name of the object upon which a function is called directs the function's actions to a specific set of data elements pointed to by the object name. You can differentiate between structure member data elements and function parameters having the same name by prefixing the keyword "this" to a data element as shown in the following code:

```
void Person::setFirstName(char* f_name) {
    strcpy(this.f_name, f_name);
}
```

Notice in this example the underscore was removed from the parameter `_f_name` to yield `f_name`. However, to distinguish between the two identical identifiers the `this` keyword must be added to the one that belongs to the object. You will see many good uses for the `this` pointer before you reach the end of this book! A thorough treatment of the `this` pointer is offered in chapter 11, so for now, just be aware of its existence. Example 10.16 shows a `main()` function using the new Person struct with its built-in functions:

Quick SUMMARY

The C++-style of using structures is decidedly object-oriented. Both structural data elements and the functions that manipulate those data elements are combined into one structure to form a new data type. New data types created by you the programmer, be they structures done C-style or C++-style, enums, or classes, are collectively referred to as user-defined types. You will also see and hear the term abstract data type used to describe new data types created by programmers in an effort to abstract the problem being solved.

Structure data and function members have public accessibility by default. You can control which members have what level of accessibility by using access specifiers. Two different access specifiers were discussed above: public and private. Declaring the member functions of a structure public while keeping its data members private is called data encapsulation. As a rule, and it is a good rule, only the member functions of a structure need have direct access to a structure's data members. The rest of this book is largely devoted to showing you why this is a good object-oriented design policy.

```

1  #include <iostream>
2  #include "personstruct.h"
3  using namespace std;
4
5  int main(){
6      Person Bob;
7
8      Bob.setFirstName("Bob");
9      Bob.setLastName("Smith");
10     Bob.setHairColor(HairColor::Black);
11     Bob.setEyeColor(EyeColor::Blue);
12
13     cout<<"First Name: "<<Bob.getFirstName()<<endl
14         <<" Last Name: "<<Bob.getLastName()<<endl
15         <<"Hair Color: "<<Bob.getHairColor()<<endl
16         <<" Eye Color: "<<Bob.getEyeColor()<<endl;
17
18     cout<<endl<<endl;
19     return 0;
20 }

```

10.16 main.cpp testing
Person struct

CLASSES: A GENTLE INTRODUCTION

Everything you learned above concerning C++ structures can be directly applied to classes. The only change required to convert the Person struct to a class is to change the keyword struct to the keyword class. Observe the following new header file now named personclass.h:

```

1  #ifndef __PERSON_CLASS_H
2  #define __PERSON_CLASS_H
3
4  namespace HairColor{
5      enum HairColor {Black, Red, Auburn, Brown, Blond, Silver, Grey};
6  }
7
8  namespace EyeColor{
9      enum EyeColor {Black, Hazel, Blue, Brown};
10 }
11
12 const int NAMESIZE = 26;
13
14 class Person {
15     public:
16         void setFirstName(char* _f_name);
17         void setLastName(char* _l_name);
18         void setHairColor(HairColor::HairColor _hair_color);
19         void setEyeColor(EyeColor::EyeColor _eye_color);
20         char* getFirstName();
21         char* getLastName();
22         char* getHairColor();
23         char* getEyeColor();
24
25     private:
26         char f_name[NAMESIZE];
27         char l_name[NAMESIZE];
28         HairColor::HairColor hair_color;
29         EyeColor::EyeColor eye_color;
30 };
31 #endif

```

10.17 personclass.h

Change the keyword
from struct to class...

Two additional changes were made on lines 1 and 2 to reflect the fact that this header file now includes a class declaration instead of struct declaration. Example 10.18 gives the implementation code for this class declaration. The only changes made to it were to the name of the file, from personstruct.cpp to personclass.cpp, and to the name of the #include file, from personstruct.h to personclass.h.

Example 10.19 gives the main() function that uses the Person class to create a Person object named Bob and per-

10.18 *personclass.cpp*

```

1 #include "personclass.h"
2 #include <string.h>
3
4 char HairColorStrings[][7] = {"Black"}, {"Red"}, {"Auburn"}, {"Brown"}, {"Blond"},
5                               {"Silver"}, {"Grey"};
6
7 char EyeColorStrings [] [7] = {"Black"}, {"Hazel"}, {"Blue"}, {"Brown"};
8
9
10 void Person::setFirstName(char* _f_name){
11     strcpy(f_name, _f_name);
12 }
13
14 void Person::setLastName(char* _l_name){
15     strcpy(l_name, _l_name);
16 }
17
18 void Person::setHairColor(HairColor::HairColor _hair_color){
19     hair_color = _hair_color;
20 }
21
22 void Person::setEyeColor(EyeColor::EyeColor _eye_color){
23     eye_color = _eye_color;
24 }
25
26 char* Person::getFirstName(){
27     return f_name;
28 }
29
30 char* Person::getLastName(){
31     return l_name;
32 }
33
34 char* Person::getHairColor(){
35     char* h_color = NULL;
36     switch(hair_color){
37
38         case HairColor::Black : h_color = HairColorStrings[HairColor::Black];
39                                 break;
40     case HairColor::Red : h_color = HairColorStrings[HairColor::Red];
41                             break;
42     case HairColor::Auburn : h_color = HairColorStrings[HairColor::Auburn];
43                             break;
44     case HairColor::Brown : h_color = HairColorStrings[HairColor::Brown];
45                             break;
46     case HairColor::Blond : h_color = HairColorStrings[HairColor::Blond];
47                             break;
48     case HairColor::Silver : h_color = HairColorStrings[HairColor::Silver];
49                             break;
50     case HairColor::Grey : h_color = HairColorStrings[HairColor::Grey];
51                             break;
52     default : h_color = HairColorStrings[HairColor::Blond];
53     }
54     return h_color;
55 }
56
57 char* Person::getEyeColor(){
58     char* e_color = NULL;
59     switch(eye_color){
60     case EyeColor::Black : e_color = EyeColorStrings[EyeColor::Black];
61                             break;
62     case EyeColor::Hazel : e_color = EyeColorStrings[EyeColor::Hazel];
63                             break;
64     case EyeColor::Blue : e_color = EyeColorStrings[EyeColor::Blue];
65                             break;
66     case EyeColor::Brown : e_color = EyeColorStrings[EyeColor::Brown];
67                             break;
68     default : e_color = EyeColorStrings[EyeColor::Blue];
69     }
70     return e_color;
71 }

```

form a few operations on the Bob object. This main() function looks exactly like the main() function used to test the struct version of Person. The one change to the main.cpp file is the name of the include file, from personstruct.h to personclass.h.

```

1 #include <iostream>
2 #include "personclass.h"
3 using namespace std;
4
5 int main() {
6     Person Bob;
7
8     Bob.setFirstName("Bob");
9     Bob.setLastName("Smith");
10    Bob.setHairColor(HairColor::Black);
11    Bob.setEyeColor(EyeColor::Blue);
12
13    cout<<"First Name: "<<Bob.getFirstName()<<endl
14         <<" Last Name: "<<Bob.getLastName()<<endl
15         <<"Hair Color: "<<Bob.getHairColor()<<endl
16         <<" Eye Color: "<<Bob.getEyeColor()<<endl;
17
18    cout<<endl<<endl;
19    return 0;
20 }

```

10.19 main.cpp testing
Person class

Quick SUMMARY

I told you this would be a gentle introduction to classes! A struct has all the functionality of a class in C++ but there are differences between the two. These differences are discussed in the next section. Programmatic differences aside, the primary difference between the two forms of data types has more to do with how you think about designing and writing programs. Starting now, unless there is a pressing need to put data elements in a struct, you can use classes for all your abstract data type needs.

THE DIFFERENCES BETWEEN STRUCTURES & CLASSES

The differences between structures and classes are summarized in table 10.1.

Feature	Structures	Classes
<i>Keyword</i>	struct	class
<i>Default member access</i>	public	private
<i>Used in object-oriented thinking</i>	No	Yes

Table 10-1: Differences Between Structures and Classes

Pretty big table! If structures are used at all by C++ programmers it is to reinforce the notion that the type they are creating is a simple aggregation. Using structs in this manner does not go against the grain of object-oriented programming per se, but, as I said earlier, anything you can do with a struct can be done with a class.

Quick SUMMARY

The syntactic and semantic differences between structures and classes lies in their keywords and default member accessibility. How you think about programming with each makes all the difference in the world. To say more about their differences would belabor the point!

OBJECT-ORIENTED THINKING

Up to now, with the exception of the last two sections, all the material in this book has been presented from a procedural programming perspective. I did this deliberately, for as a C++ programmer, even though you will make the intellectual leap to thinking in objects, you eventually have to implement your member functions, which ultimately means putting one line of source code after another.

I took this pedagogical approach because if you are completely new to programming, or are approaching C++ with a procedural programming background, learning the syntax and the semantics of the language, while at the same time having classes and objects forced upon you, makes for a steep learning curve. But now, all that is behind you. You are here, reading this paragraph. You have learned the basics of the language. You know how to write procedurally oriented programs in C++ and are ready for more excitement — ready to make the jump to object speed! I'm ready if you are ready, so let us start with a whole new vocabulary. It is time for you to learn object speak!

OBJECT SPEAK: A NEW VOCABULARY

A change in thinking requires a vocabulary of new words with which you can express the wonderful new thoughts you will soon have about the design and operation of programs. The following list will get you started. I have used some of them in this and previous chapters in a subliminal attempt to alter your thought processes. These words are testable:

Term	Definition
<i>Class</i>	This word has several meanings. Primarily, the word class denotes a set of objects that share common structure and behavior. A class declaration introduces a new user-defined data type and specifies the structure and behavior of objects of that type.
<i>Abstract Base Class</i>	A class that contains one or more pure virtual functions. An abstract base class serves to publish an interface. There can be no instance of an abstract base class.
<i>Base Class</i>	A class whose functionality is inherited by another class. A base class might be the most generalized class in a class inheritance hierarchy, but usually the term base class is applied to the immediate class from which a derived class inherits its functionality. In object-oriented design, general class functional characteristics are implemented in a base class with the intention of making this general class functionality available for inheritance.
<i>Superclass</i>	Another term for base class.
<i>Inheritance</i>	The act of adopting the behavior of a particular class of objects by another class of objects.
<i>Derived Class</i>	A class that inherits functionality from one or more base classes.
<i>Subclass</i>	Another term for derived class.
<i>Object</i>	This term has several meanings. The term object fundamentally means a region of memory. When a variable of a particular type is declared, a region of memory is set aside for the storage of its contents. Every object created resides in a different region of memory. In the case of complex user-defined class types, the memory regions occupied by class objects may be large compared to fundamental data types like char, int, and float. The term object is also used when you picture the interaction between object-oriented components in your mind. This is an example of reducing the concept of an object to its most abstract form. An object in this sense is a component in a complex system that interacts with other objects.
<i>Interface</i>	A publicly accessible set of functions intended to be the authorized way to access an object's functionality. In some cases, direct access to data members is provided. Doing so violates the concept of data encapsulation.

Table 10-2: Object-Oriented Terminology

Term	Definition
<i>Sending a message to an object</i>	Calling one of an object's interface functions. When one object sends a message to another object, the sender is simply calling one of several possible interface functions available for use in the target object. The object sending the message is said to be the client while the object receiving the message is said to be the server.
<i>Invoking a method on an object</i>	Calling one of an object's interface functions.
<i>Object-Oriented Analysis & Design</i>	The act of designing software in terms of objects and their interfaces. Ideally, all that you need to know about an object-oriented software application to understand its operation at a conceptual level is the objects that comprise it and their public interfaces. The details are left to the implementation.
<i>Data Encapsulation</i>	The act of shielding class data members from public access by declaring them private. A class's data members should not be shared horizontally or vertically, meaning they should not be made available to objects of another class, as in a client server relationship, nor should they be shared via inheritance.
<i>Virtual Function</i>	A function declared in a base class that can be overridden in a derived class.

Table 10-2: Object-Oriented Terminology

This list of object-oriented vocabulary will grow as you progress through the remaining chapters of this book.

SUMMARY

You can abstract the problem you are solving with a computer by declaring synonyms for existing data types or designing your own. The typedef keyword lets you create a synonym for an existing data type making it better suited to represent your problem domain. Enumerations are new types that allow integer values to be referenced by name. Your compiler can be set to ensure you don't try to assign an unauthorized state value to an enumerated type variable, thus reducing the risk of making programming errors while at the same time making your source code easier to read and manage.

Structures can be used in two ways: C-style, where only data members are contained in the structure, or C++-style, where both data members and functions are placed in the structure. The C-style method of using structures is based on a procedural programming mind set where data structures are defined separate from the functions that manipulate those structures. It is a good idea to understand the C-style method in case you have to maintain legacy C code.

Access structure data and function members using the dot "." operator. If you have a pointer to a structure object then use the short hand member access "->" operator. It is called the short hand operator because using it lets the compiler handle pointer dereferencing operations for you. The alternative to the "->" operator is to use the "." operator in conjunction with the "*" operator.

C++ structures have all the functionality of classes but differ in their default member accessibility. The default member accessibility of structures is public, whereas the default member accessibility for classes is private. The primary difference between structures and classes lies in the way you think about programming.

A new way of programming requires a new way of thinking which, in turn, requires a new vocabulary. From now on you will speak in object speak — do I make myself clear? Right! Now, go forth and do all the exercises. I will see you in chapter 11.

Skill Building Exercises

1. **typedef:** Use the typedef specifier to declare synonyms for the integer or float data types better suited to model data from the domains below. For example, if the domain is Banking, and the data to model is Currency, then you could create a type synonym with typedef in the following manner;

```
typedef float Currency;
```

Domain	Data to Model
<i>Banking</i>	Currency, Transactions
<i>Physics</i>	Speed, Time, Velocity, Acceleration
<i>Computers</i>	Operand, Opcode, Instruction
<i>Chemistry</i>	Molecules
<i>Writing</i>	Words, Paragraphs, Sentences
<i>Census</i>	Population
<i>Government</i>	Laws, Statutes
<i>Plumbing</i>	Fixtures

2. **enum:** a. Write the declaration for an enumerated type named Direction and give it the state values North, North-East, East, South-East, South, South-West, West, & North-West. b. Write the declaration for an enumerated type named PenPosition and give it the state values Down and Up.
3. **enum:** a. Write the declaration for an enumerated type named Safety and give it the state values Safe and Armed. Assign Safe the value 1 and Armed the value 2.
4. **structures:** Give the declaration for a structure named BlockType that contains the following data members: An array of 15 characters named Model, and two floats named height and width.
5. **structures:** Give the declaration for an Employee structure containing the following data members: Two character arrays of length 26 named employee_f_name and employee_l_name, a float member named pay_amount, and six functions named setEmployeeFirstName, setEmployeeLastName, getEmployeeFirstName, getEmployeeLastName, setEmployeePayAmount, and getEmployeePayAmount. Declare the data members private and the member functions public.
6. **structures:** Give the declaration for a structure named RobotRat containing the following data members: Two ints named row and col, an enumerated type variable named pen_position, and another enumerated type variable named direction. The enumerated types for these variables were declared in Skill Building Exercise 2.
7. **Array of structures:** Using the Employee structure declared in skill building exercise 5, write a program that declares an array of five employees and use each function to set the data members of each employee. Use a for loop to add up all the employee pay amounts and print the value to the screen.
8. **Array of pointers to structures:** Convert the array program you wrote in the previous exercise to use pointers to Employee objects. Dynamically create six Employees and call the functions to set their attributes. Sum the total of their pay as you did before.

9. **classes:** Convert the Employee structure to a class and rebuild your program. What changes did you have to make to get it to compile and run?
10. **Array of classes:** Convert the array of Employee structs program you wrote in exercise 7 to an array of Employee classes. What changes, if any, did you have to make to the program?

SUGGESTED PROJECTS

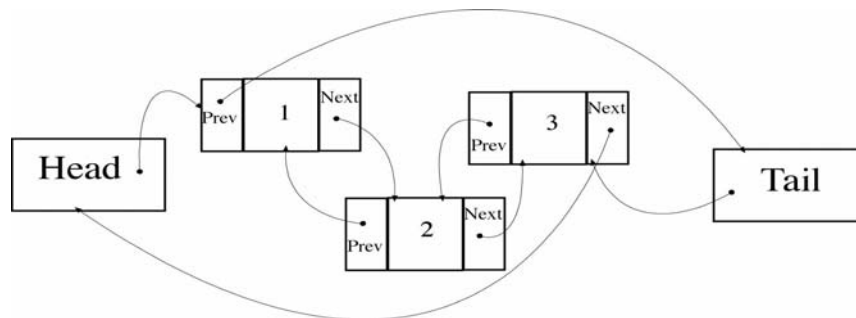
1. **Library Manager:** Write a program to keep track of your books. Declare a structure called Book that contains the necessary data members to describe a book. A few of these data members might be named Title, Author, and ISBN, etc. Allow users to save and load your library information from disk. The hardest part of this project will be determining how to represent the collection of books in memory. You might choose to use an array. You could also explore some of the Standard Template Library collection classes that may make your life easier.
2. **RobotRat Structure:** Convert the RobotRat project implemented in chapter 3 to use a structure to represent the RobotRat. The structure declaration for RobotRat was given as an exercise in skill building exercise 6.
3. **Computer Simulator Structure:** Convert the computer simulation project from chapter 9 to use a structure to represent the computer. The computer structure might include its memory, the program counter, and the accumulator. Possible functions for the computer might include loadProgram(), runProgram(), fetch(), decode(), execute(), and store().
4. **Linked List:** A special property of structures and classes in C++ is that the name of the structure or class can appear in the body of the declaration. Consider the following example:

```

struct Node{
    int item;
    Node* previous;
    Node* next;
};

```

Here, the struct name Node appears in the body of the Node declaration to declare two Node pointers named previous and next. This method is used to declare data structures fit for use in a linked list. Use the struct declaration above to write a linked list. This is as much an exercise in the use of pointers as it is the use of structures. Here are a few hints to get you started:



The diagram above shows three Node objects inserted into a doubly linked list. There are two Node pointer variables named Head and Tail. The Head pointer is set to point to the first Node object inserted into the list. The Tail pointer should always point to the last Node object inserted into the list. The Node pointers in each Node object are set to point to the previous Node object and the next Node object as required. Examine node 1 in the diagram. The previous pointer points to Tail, which it should always point to, and the next pointer points to node 2.

You are to write functions that let you insert and remove nodes at or from any position in the list, and functions

that let you traverse the list both forward and backward.

5. **Linked List:** Convert the library program described in project 1 to use a linked list data structure. After completing the project, write a brief paragraph describing why linked lists are a more efficient use of memory than an array.

SELF TEST QUESTIONS

1. (T/F) The typedef keyword lets you create a new data type that better represents your problem domain.
2. (T/F) Enumerations are new data types.
3. What type of operator would you use to access a structure object's data or function members?
4. What type of operator would you use to access a structure object's data or function members via a pointer?
5. Describe how to use the "." operator in conjunction with the "*" operator to access a structure object's data or functions members via a pointer.
6. List and discuss the three major differences between structures and classes.
7. Describe the code changes that were necessary to convert the struct version of Person to the class version.
8. Why do you suppose the default access for a class is private?
9. Define the term data encapsulation.
10. What is meant by the term interface?

REFERENCES

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language*. Second Edition. Prentice Hall, Englewood Cliffs, New Jersey, 1988. ISBN: 0-13-110370-9

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994. ISBN: 0-8053-5340-2

Paul J. Lucas. *The C++ Programmer's Handbook*. Prentice Hall P T R, Englewood Cliffs, New Jersey, 1992. ISBN: 0-13-118233-1

Robert C. Martin. *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice Hall, Englewood Cliffs, New Jersey, 1994. ISBN: 0-13-203837-4

NOTES

CHAPTER 11



STREET SCENE OPUS ONE

DISSECTING CLASSES

LEARNING OBJECTIVES

- STATE THE PURPOSE AND USE OF THE CLASS CONSTRUCT IN C++
- LIST AND DESCRIBE THE PARTS OF A CLASS DECLARATION
- STATE THE IMPORTANCE OF THE TERMINATING SEMICOLON OF A CLASS DECLARATION
- EXPLAIN HOW TO USE ACCESS SPECIFIERS TO CONTROL HORIZONTAL MEMBER ACCESS
- STATE THE FUNCTION AND PURPOSE OF CONSTRUCTORS
- STATE THE PURPOSE AND USE OF OVERLOADED CONSTRUCTORS
- EXPLAIN HOW TO OVERLOAD CONSTRUCTORS
- EXPLAIN HOW TO USE THE INITIALIZER LIST TO INITIALIZE CLASS ATTRIBUTES
- STATE THE PURPOSE AND USE OF DESTRUCTORS
- EXPLAIN HOW TO OVERLOAD CLASS MEMBER FUNCTIONS
- EXPLAIN THE IMPORTANCE OF SEPARATING THE CLASS INTERFACE FROM ITS IMPLEMENTATION
- EXPLAIN HOW TO CALL CLASS MEMBER FUNCTIONS FROM WITHIN CLASS MEMBER FUNCTIONS
- UTILIZE COMPLEX CLASS CONSTRUCTS IN YOUR C++ PROGRAMMING PROJECTS
- UTILIZE INITIALIZER LISTS TO INITIALIZE CLASS ATTRIBUTES
- LIST AND DEFINE THE FOLLOWING TERMS: CONSTRUCTOR, DESTRUCTOR, DEFAULT CONSTRUCTOR, OVERLOADED CONSTRUCTOR, AND OVERLOADED FUNCTIONS

INTRODUCTION

This chapter deals exclusively with the C++ class construct. Here you will learn about data encapsulation, horizontal member access, special member functions, initializer lists, member function overloading, and data member accessor and mutator functions. I will also introduce you to the Unified Modeling Language (UML) class diagram. All the material presented here is intended to give you a solid foundation for further study of C++ in the context of good object-oriented design.

You caught a glimpse of data encapsulation in action in chapter 10. There, structure and class data members were declared private to prevent unauthorized access. The material in this chapter continues that discussion and shows you how to control the horizontal access granted to a class's data and function members using the access specifiers `public`, and `private`. The protected access specifier is covered in detail in chapter 13.

Something you did not see in the previous chapter were the special member functions called constructors and destructors. Constructors are used to initialize class objects to some known state. There should be no surprises regarding the state of instance data members when an object is created. Destructors are used to free any resources allocated for an object's use during its lifetime. An initializer list is an extremely important part of a class constructor because it is the only place where class constants can be initialized. You will also need to know about initializer lists when you get to chapter 13.

Just as regular functions can be overloaded, so too can class member functions including constructors. You will learn several good reasons for doing so. You will also learn how to write class accessor and mutator functions to access and manipulate class data members.

The Unified Modeling Language is considered the standard for object-oriented software design. In this chapter I will introduce you to the UML class diagram. You will find it helpful to express your design ideas in pictures, especially when working with large numbers of classes. A full treatment of the UML is beyond the scope of this book, but if you are interested in learning more I suggest starting with one of the UML references listed at the end of chapter.

THE CLASS CONSTRUCT

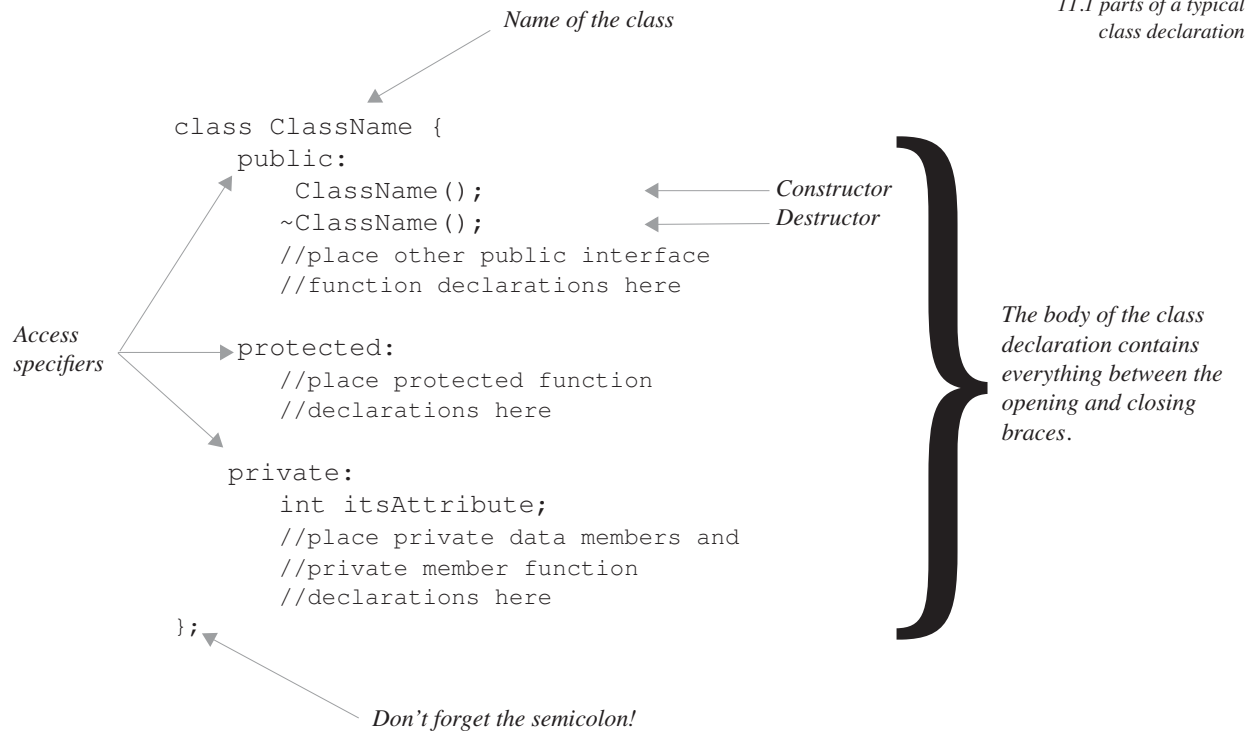
The class sits at the root of all object-oriented thinking. When you declare a new class you are declaring a new data type from which objects can be created and used in a program. This is nothing new; you were exposed to structs and classes in chapter 10, and you've seen fundamental data types like `char`, `int`, and `float` used to create variables for use in programs as well.

What is new, however, is how you begin to think of classes and their relationship to other classes in a software system. In a complex software system, a class type, or several class types, will be declared and used to create one or more objects. An object in this sense is like a simple variable created from a fundamental data type; it could be intended to be used alone, but more than likely it is designed to be used in conjunction with other class type objects. Hence, a class type object is different from an object created from a fundamental data type because of the expanded role it can play in a software system. A class type object can send or receive messages to itself or to other class type objects, something ordinary objects created from fundamental data types cannot do.

PARTS OF A CLASS DECLARATION

Example 11.1 shows the various parts of a typical class declaration. The class declaration introduces a new data type. It begins with the class keyword followed by an identifier that forms the class name. I am not going to tell you how to name your classes, however, convention suggests that typographically they begin with a capital letter, and capitalize the first letter of each word that appears in the name. Grammatically, they should be singular noun names. Note the following examples:

```
Person
Employee
FuelPump
RotorShaftSensor
```



The body of the class is that area between the opening and closing braces. In the body will appear any and all data member declarations and member function declarations. Convention suggests placing all public interface functions at the top of the class body. Since the default access to class members is private, this requires the use of the public access specifier. Class-wide constants can be, and usually are, given public access, and are placed along with the public interface functions under the public access specifier.

Protected functions are placed under the protected access specifier. The protected access specifier is used primarily to prevent horizontal access but allow vertical access to protected members by derived classes. In this sense it works exactly like the private access specifier from a horizontal access perspective.

Private data members and functions are placed under the private access specifier. Often times member functions are declared private as well as data members. I will show you why this is good design practice later in the section on access specifiers.

Under the public access specifier appear two special purpose function declarations. A constructor and a destructor. These functions are special for several reasons, but mainly, they have the same name as the class in which they appear. Constructors and destructors are covered in detail below.

Whatever you do, do not forget to put the semicolon after the closing brace of a class declaration. The compiler errors you get may be cryptic and it will be by dumb luck alone that, having exhausted every possible remedy to the problem, you decide to look to your class declaration one more time for relief. And then you notice, it is not there. You type the semicolon and compile, and a thousand compiler errors magically disappear. You have been warned, but the lesson will not have been learned until you make the mistake yourself.

A MINIMUM CLASS DECLARATION

Although example 11.1 shows a typical class declaration, the following line of code declares a complete class although it doesn't do very much!

```
class Foo{};
```

Given this class declaration you could now use it to create Foo objects. The compiler will create both a default constructor and destructor, but, since there are no data members or member functions, the best you can do with Foo is create Foo objects and destroy them.

PLACE CLASS DECLARATIONS IN SEPARATE HEADER FILES

Get into the habit of placing class declarations in header files. There should be only one class declaration per header file. Doing so makes finding a particular declaration easier than if you had placed several declarations in one header.

THE UML CLASS DIAGRAM

The UML representation for the class `ClassName` given in example 11.1 above is shown in figure 11-1 below:

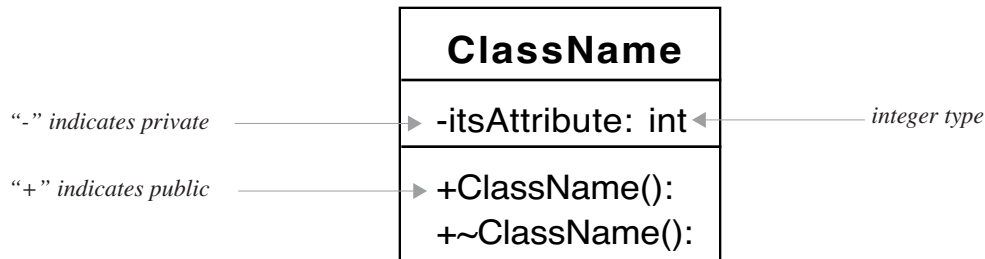


Figure 11-1: UML Representation for the Class `ClassName`

UML classes are easy to draw and require no special software to create. You could use a pencil and paper to create them although if you plan on doing serious object-oriented analysis and design I recommend purchasing a good UML design tool such as Object Plant™, Describe™, or Rational Rose™.

Referring to figure 11-1, a class is drawn as a rectangle with several compartments. The upper compartment contains the name of the class. The compartment below that contains the class's attributes, and the one below that contains the class's member functions. A minus sign in front of an attribute or member function name indicates private accessibility. A plus sign in front of an attribute or member function name indicates public accessibility.

The attribute named `itsAttribute` is declared to be of type `integer` as indicated by the keyword `int` following the colon to the right of its name. The constructor and destructor are declared to return no type as indicated by the lack of a return type to the right of their colons.

Class diagrams can be constructed from one or more UML classes. A class diagram pictorially represents the static relationship between classes in a software application. Figure 11-2 shows a class diagram for a simple navy fleet simulation application.

THE CONCEPTS OF STATE AND BEHAVIOR

When designing object-oriented software you will think of objects as having state and behavior.

Object State

An object's state is indicated by the value of its attributes at any given time during the object's lifetime. Change an attribute's value during an object's lifetime and you change its state. Access to object attributes should be provided through class interface functions; object attributes should never be exposed for direct manipulation because they are considered an implementation detail.

Object Behavior

An object derives its behavior through the implementation of its class interface functions so it is a good idea to give class interface functions names that reflect the behavior they will produce when called.

CLASS MEMBER FUNCTIONS

A class declaration introduces a new user-defined data type that can contain both attributes and functions. This

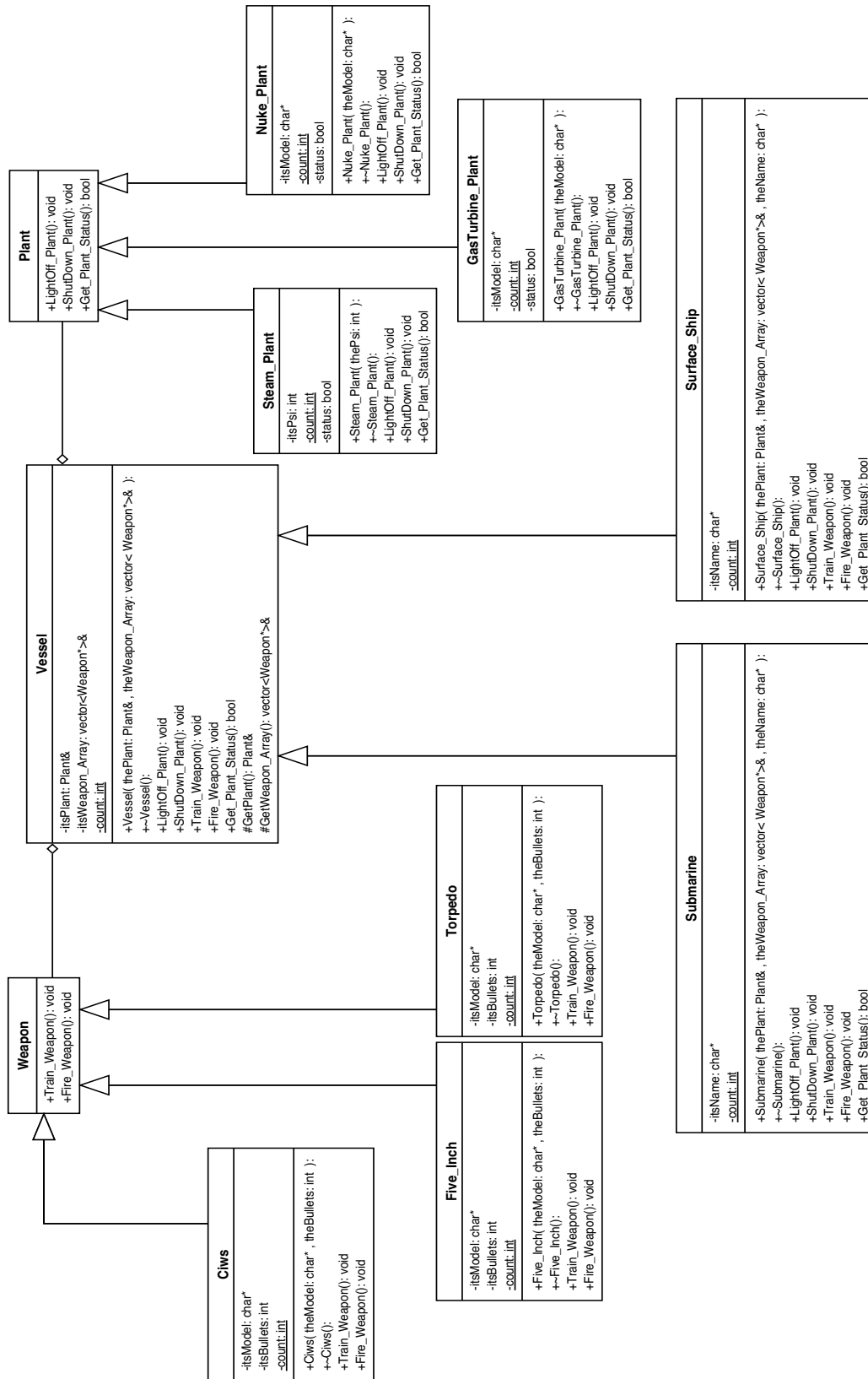


Figure 11-2: UML Class Diagram of a Simple Navy Fleet Simulation Application

section focuses on the declaration and definition of class member functions of which there are two general types: special member functions and all the rest. The special member functions include default and copy constructor, copy assignment operator, and destructor. All the rest include any additional functions required to give class objects their desired behavior. Before discussing special member functions I want to touch briefly on how member functions access class data members.

CLASS MEMBER FUNCTION ACCESS TO CLASS ATTRIBUTES

Class member functions behave as regular functions do, so all the material you learned in chapter 9 still applies. However, because class member functions belong to a particular class of object they will have direct access to all class attributes. In this regard it is helpful to think of a class attribute as being globally declared to be visible to all class member functions. And, just as with regular functions, any parameters or local function variables having the same names as class attributes will hide those class attributes.

OBTAINING ACCESS TO CLASS ATTRIBUTES FROM A MEMBER FUNCTION

A class attribute is a static class variable of which only one copy exists to be shared by all instances of that class. To access a class attribute from within a member function simply use the class attribute name. If the attribute name is masked by a local function variable prefix the class name and the scope resolution operator to the attribute name like so:

```
ClassName::class_attribute_name
```

OBTAINING ACCESS TO INSTANCE ATTRIBUTES FROM A MEMBER FUNCTION

A non-static class attribute is an instance attribute and each class instance object will have its very own copy of the attribute. To access an instance variable in a member function simply use the variable's name. If you have declared a local function variable with the same name as the instance attribute then you can prefix the keyword `this` and the indirection operator to the instance attribute's name as shown here...

```
this->instance_attribute_name
```

...which is the shorthand method of doing the following:

```
(*this).instance_attribute_name
```

SPECIAL MEMBER FUNCTIONS

A class has four types of special member functions: default constructor, copy constructor, copy assignment operator, and destructor. Special member functions are used when objects are created, copied, and destroyed.

If you do not explicitly declare and define these special member functions in your classes they will be created by the compiler, but their default behavior may or may not be what you expect. As a rule, you should implement each of the four special member functions for all classes you write. Doing so will ensure your class objects are well-behaved. The idea of well-behaved objects is covered in great detail in chapter 17.

CONSTRUCTOR

A constructor is a special member function that is called automatically when an object of a particular class type is created. A constructor has exactly the same name as the class to which it belongs. There can be more than one constructor per class. A constructor has no return value type, not even void. A default constructor is a constructor with either no parameters or one whose parameters all have default values, meaning it can be called with no arguments. The default constructor is discussed in more detail below.

PURPOSE OF A CONSTRUCTOR

The purpose of a constructor function is to initialize a class object to some known state when the object is created. By initialization I mean all of an object's instance attributes should be set to an acceptable value; there must be no surprises regarding the state of an object's attributes.

DEFAULT CONSTRUCTOR

A default constructor is a constructor that requires no arguments. If you do not explicitly define a default constructor for a class, a compiler-defined default constructor will be used instead. Compiler-supplied default constructors perform basic class data member initialization. Relying on compiler-supplied constructors to properly initialize class objects is usually a big mistake.

OVERLOADING CONSTRUCTORS

Constructors can be overloaded, just like ordinary functions, so that class objects can be initialized in different ways.

CONSTRUCTOR INITIALIZER LIST

A constructor initializer list provides an expedited way to initialize class attributes. A constructor initializer list is a set of attribute initializers that appears between a colon and the opening brace of the constructor function body. The following code shows an example:

```
ClassName::ClassName():attribute_1(initializer_val), attribute_2(initializer_val), ...,
                        attribute_n(initializer_val) {
    //constructor body
}
```

If the only purpose for a particular constructor is object attribute initialization then all initializations can be performed in an initializer list and the body of the constructor can remain empty.

There are three reasons for using an initializer list. First, when you simply want to initialize data members. Initializations performed in the initializer list are guaranteed to be performed before the body of the constructor executes. Second, when you need to call a base class constructor. This use of an initializer list in this regard will be discussed in greater detail in chapter 13. Last, when you need to initialize class or instance constants. Constants must be initialized in an initializer list since they must be initialized when they are created.

TestClass Example

Example 11.2 shows the contents of a header file named `testclass.h` that declares a class named `TestClass`. `TestClass` will be used to demonstrate constructors and will be modified later to demonstrate copy constructors, copy assignment operators, and destructors.

```
1  #ifndef TEST_CLASS_H 11.2 testclass.h
2  #define TEST_CLASS_H
3
4  class TestClass{
5      public:
6          TestClass(int const_val = 25, int i_val = 0);
7          int getConstVal();
8          void setI(int i_val);
9          int getI();
10
11     private:
12         const int CONST_VAL;
13         int i;
14 };
15 #endif
```

Referring to example 11.2, `TestClass` is declared to have four public interface functions and two private data members. One data member is a constant, the other a variable.

The first public function is the constructor declared on line 6. This is an example of a default constructor in that its two parameters have default values assigned to them. This means that if no arguments are supplied to the constructor when it is called, it will use the default values to initialize the data members. Since the `CONST_VAL` data member is a constant, it must be initialized in an initializer list. Notice too that the constructor has no return type.

The other three functions set or retrieve the object attributes indicated by their names.

Example 11.3 gives the source code for the `testclass.cpp` file.

```

1  #include "testclass.h"
2  #include <iostream>
3  using namespace std;
4
5  TestClass::TestClass(int const_val, int i_val):CONST_VAL(const_val),
6      i(i_val){
7      cout<<"CONST_VAL = "<<CONST_VAL<<endl;
8      cout<<"      i = "<<i<<endl;
9  }
10
11 int TestClass::getConstVal(){
12     return CONST_VAL;
13 }
14
15 void TestClass::setI(int i_val){
16     i = i_val;
17 }
18
19 int TestClass::getI(){
20     return i;
21 }

```

11.3 testclass.cpp

colon starts initializer list

comma separates initializers

Focus first on the `TestClass` constructor defined on line 5. The default values are not repeated in the function definition. The colon marks the beginning of the initializer list. Each data member is initialized with its corresponding parameter using constructor notation. Take for example the first initializer `CONST_VAL(const_val)`. `CONST_VAL` is the name of the data member `CONST_VAL`, and `const_val` is the name of one of the constructor parameters. The value of `const_val` is used to initialize `CONST_VAL`. The data member `i` is initialized after `CONST_VAL` and the two initializers are separated by a comma. `i_val` appears below `CONST_VAL` in this example only because of space limitations, but this is a good example of how you should line wrap initializers to aid readability. The body of the constructor appears after `i_val` and contains two output statements. Example 11.4 shows a `main()` function using `TestClass` to create several different `TestClass` objects using different forms of the constructor.

```

1  #include "testclass.h"
2  using namespace std;
3
4  int main(){
5      TestClass t1, t2(45, 5), t3(3);
6      return 0;
7  }

```

11.4 main.cpp

In this example, three `TestClass` objects are created. The first, `t1`, is created with no arguments, therefore both instance attributes will be initialized to default values. The second object, `t2`, is created using two arguments. Notice how parenthesis are used to call the constructor. The value 45 will be used to initialize `CONST_VAL`, and the value 5 will be used to initialize `i`. The third object, `t3`, is created with one argument. Which instance attribute will be set? The answer is found by studying the output of the program shown in figure 11-3.

The argument value 3 used to create `t3` is used to initialize the `const_val` parameter, which is used to initialize `CONST_VAL` as can be seen above.

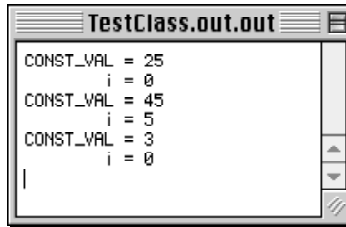


Figure 11-3: Results of Running Example 11.4

Copy CONSTRUCTOR

The copy constructor is a constructor that is used to create a new instance object using an existing instance object as a guide. The first argument to a copy constructor must be a reference to an instance of the class for which it is defined.

PURPOSE OF THE COPY CONSTRUCTOR

The purpose of the copy constructor is similar to a regular constructor in that it is used to create a new object, using an existing object's attribute values to initialize the new object's attributes. The copy constructor is not intended for human use, rather, it is the constructor that will be used when objects are passed to functions by value. An example of its use is shown in example 11.7.

OVERLOADING COPY CONSTRUCTORS

Copy constructors can be overloaded just like regular constructors but you rarely see this done in practice.

TESTCLASS EXAMPLE EXTENDED

Example 11.5 shows the TestClass declaration extended to include a copy constructor.

11.5 testclass.h

```

1  #ifndef TEST_CLASS_H
2  #define TEST_CLASS_H
3
4  class TestClass{
5      public:
6          TestClass(int const_val = 25, int i_val = 0);
7          TestClass(TestClass& tc_obj);
8          int getConstVal();
9          void setI(int i_val);
10         int getI();
11
12     private:
13         const int CONST_VAL;
14         int i;
15 };
16 #endif

```

The copy constructor declaration appears on line 7. The only parameter to the copy constructor in this example is a reference to a TestClass object named tc_obj.

Example 11.6 gives the code for the revised testclass.cpp file and example 11.7 gives the code for the revised main() function showing the copy constructor in use.

Referring to example 11.6, the copy constructor definition appears on line 11. Since TestClass functions have direct access to TestClass data members, the attributes of the copy constructor parameter tc_obj can be directly accessed via the dot operator and used to initialize the attributes of the new object being created.

```

1  #include "testclass.h"
2  #include <iostream>
3  using namespace std;
4
5
6  TestClass::TestClass(int const_val, int i_val):CONST_VAL(const_val), i(i_val){
7      cout<<"CONST_VAL = "<<CONST_VAL<<endl;
8      cout<<"          i = "<<i<<endl;
9  }
10
11 TestClass::TestClass(TestClass& tc_obj):CONST_VAL(tc_obj.CONST_VAL), i(tc_obj.i){
12     cout<<"CONST_VAL = "<<CONST_VAL<<endl;
13     cout<<"          i = "<<i<<endl;
14 }
15
16 int TestClass::getConstVal(){
17     return CONST_VAL;
18 }
19
20 void TestClass::setI(int i_val){
21     i = i_val;
22 }
23
24 int TestClass::getI(){
25     return i;
26 }

```

11.6 testclass.cpp

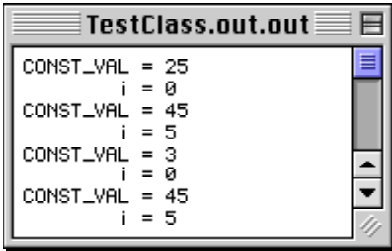
11.7 main.cpp

```

1  #include <iostream>
2  #include "testclass.h"
3  using namespace std;
4
5  int main(){
6      TestClass t1, t2(45, 5), t3(3);
7      TestClass t4(t2);
8      return 0;
9  }

```

Referring to example 11.7, the copy constructor is exercised on line 7 by creating a new TestClass object named t4 using t2 as an argument. The results of running this program are shown in figure 11-4.



```

CONST_VAL = 25
          i = 0
CONST_VAL = 45
          i = 5
CONST_VAL = 3
          i = 0
CONST_VAL = 45
          i = 5

```

Figure 11-4: Results of Running Example 11.7

Copy Assignment Operator

The copy assignment operator is an overloaded assignment “=” operator. The topic of overloading operators is formally presented in chapter 14 but because the copy assignment operator is considered a special member function I will discuss it here.

The copy assignment operator declaration takes the following form:

```
ClassName& operator=(ClassName& rhs);
```

It returns a reference to an object of type `ClassName`, where `ClassName` is the name of the class in which the function is declared. The copy assignment operator declares one parameter of type reference to `ClassName`. Here the parameter is named `rhs` which stands for “right hand side”. Now, you could name the parameter any name you want to but I like to name overloaded operator parameters `rhs` for the reasons you will see shortly.

PURPOSE of THE COPY ASSIGNMENT OPERATOR

The purpose of the copy assignment operator is to change the values of an existing object’s attributes to those of another existing object’s attribute values.

USING THE COPY ASSIGNMENT OPERATOR

The copy assignment operator is used like an ordinary assignment operator. For example, when the copy assignment is overloaded for `TestClass` objects, one object’s attributes can be set to equal another object’s attributes in the following manner:

```
t3 = t4;
```

Let’s see a complete example of the copy assignment operator declared and implemented for `TestClass`.

TESTCLASS EXAMPLE EXTENDED

Example 11.8 shows the `TestClass` declaration with the copy assignment operator function added on line 8.

```

1  #ifndef TEST_CLASS_H                                11.8 testclass.h
2  #define TEST_CLASS_H
3
4  class TestClass{
5      public:
6          TestClass(int const_val = 25, int i_val = 0);
7          TestClass(TestClass& tc_obj);
8          TestClass& operator=(TestClass& rhs);
9          int getConstVal();
10         void setI(int i_val);
11         int getI();
12
13     private:
14         const int CONST_VAL;
15         int i;
16     };
17 #endif

```

Example 11.9 shows the `testclass.cpp` file with the copy assignment operator function definition added. The function declaration begins on line 16. Notice that only the value of `i` is altered. That’s because `CONST_VAL` cannot be changed because it is a constant, and constants can only be set in a constructor initializer list.

Example 11.10 shows the code for the `main()` function where the copy assignment operator is used to assign the value of `t1` to `t2`. The assignment takes place on line 8 and two extra lines of code were added to print out the values of `t2` after the assignment.

The results of running example 11.10 are shown in figure 11-5.

DESTRUCTOR

The last type of special member function is the destructor. The destructor function has the same name as the class in which it is declared with the tilde “~” character prefixed to the name. Note the following example of a destructor declaration:

```
~ClassName ()
```

The destructor takes no arguments and returns no value. A destructor is never called directly, although when a dynamically created object is released with the `delete` operator its destructor is called. Otherwise, destructors are automatically called when objects go out of scope like at the end of a block or when a program ends.

11.9 *testclass.cpp*

```

1  #include "testclass.h"
2  #include <iostream>
3  using namespace std;
4
5
6  TestClass::TestClass(int const_val, int i_val):CONST_VAL(const_val), i(i_val){
7      cout<<"CONST_VAL = "<<CONST_VAL<<endl;
8      cout<<"          i = "<<i<<endl;
9  }
10
11 TestClass::TestClass(TestClass& tc_obj):CONST_VAL(tc_obj.CONST_VAL), i(tc_obj.i){
12     cout<<"CONST_VAL = "<<CONST_VAL<<endl;
13     cout<<"          i = "<<i<<endl;
14 }
15
16 TestClass& TestClass::operator=(TestClass& rhs){
17     i = rhs.i;
18     return *this;
19 }
20
21 int TestClass::getConstVal(){
22     return CONST_VAL;
23 }
24
25 void TestClass::setI(int i_val){
26     i = i_val;
27 }
28
29 int TestClass::getI(){
30     return i;
31 }

```

11.10 *main.cpp*

```

1  #include <iostream>
2  #include "testclass.h"
3  using namespace std;
4
5  int main(){
6      TestClass t1, t2(45, 5), t3(3);
7      TestClass t4(t2);
8      t2 = t1;
9      cout<<"CONST_VAL = "<<t2.getConstVal()<<endl;
10     cout<<"          i = "<<t2.getI()<<endl;
11     return 0;
12 }

```

```

CONST_VAL = 25
          i = 0
CONST_VAL = 45
          i = 5
CONST_VAL = 3
          i = 0
CONST_VAL = 45
          i = 5
CONST_VAL = 45
          i = 0

```

Figure 11-5: Results of Running Example 11.10

PURPOSE of DESTRUCTORS

The purpose of a destructor is to release any resources allocated for an object's use when it was created. An example of such a resource is dynamically allocated memory. If an object uses dynamically allocated memory during

its lifetime then it should insure that all such memory is freed up when it is destroyed. This is the purpose of the destructor.

TESTCLASS EXAMPLE EXTENDED

Example 11.11 gives the source code for the TestClass declaration with the destructor function declaration added.

```

1 #ifndef TEST_CLASS_H 11.11 testclass.h
2 #define TEST_CLASS_H
3
4 class TestClass{
5     public:
6         TestClass(int const_val = 25, int i_val = 0);
7         TestClass(TestClass& tc_obj);
8         TestClass& operator=(TestClass& rhs);
9         ~TestClass();
10        int getConstVal();
11        void setI(int i_val);
12        int getI();
13
14    private:
15        const int CONST_VAL;
16        int i;
17 };
18 #endif

```

The destructor function declaration appears on line 9. Since there is no dynamic memory or other resources to release the destructor will be utilized to print a short message to the screen indicating that the objects were destroyed. The modified testclass.cpp file is given in example 11.12.

```

1 #include "testclass.h" 11.12 testclass.cpp
2 #include <iostream>
3 using namespace std;
4
5
6 TestClass::TestClass(int const_val, int i_val):CONST_VAL(const_val), i(i_val){
7     cout<<"CONST_VAL = "<<CONST_VAL<<endl;
8     cout<<"      i = "<<i<<endl;
9 }
10
11 TestClass::TestClass(TestClass& tc_obj):CONST_VAL(tc_obj.CONST_VAL), i(tc_obj.i){
12     cout<<"CONST_VAL = "<<CONST_VAL<<endl;
13     cout<<"      i = "<<i<<endl;
14 }
15
16 TestClass& TestClass::operator=(TestClass& rhs){
17     i = rhs.i;
18     return *this;
19 }
20
21 TestClass::~~TestClass(){
22     cout<<"Goodbye cruel world! TestClass object destroyed."<<endl;
23 }
24
25 int TestClass::getConstVal(){
26     return CONST_VAL;
27 }
28
29 void TestClass::setI(int i_val){
30     i = i_val;
31 }
32
33 int TestClass::getI(){
34     return i;
35 }

```

There is no reason to modify the previous version of the main() function; the one shown in example 10.10 is still good. Running the program now produces the results shown in figure 11-6.

```

CONST_VAL = 25
i = 0
CONST_VAL = 45
i = 5
CONST_VAL = 3
i = 0
CONST_VAL = 45
i = 5
CONST_VAL = 45
i = 0
Goodbye cruel world! TestClass object destroyed.
Goodbye cruel world! TestClass object destroyed.
Goodbye cruel world! TestClass object destroyed.
Goodbye cruel world! TestClass object destroyed.

```

Figure 11-6: Results of Running Example 10.10 Again

Now that the destructor is declared and defined for `TestClass` you can see when each object is destroyed as the program terminates. You will see destructors used in ever-expanding roles as you progress through the text.

BEHAVIOR OF DEFAULT SPECIAL FUNCTIONS

Now that you know a little something about the four special function types I'd like to show you their default behaviors. As I said above, if you fail to implement the special functions in your classes the compiler will provide default versions for you. Their behavior may not be what you need or expect. I will demonstrate this by showing you a class called `SimpleClass`, whose declaration is given in example 11.13.

```

1  #ifndef SIMPLE_CLASS_H 11.13 simpleclass.h
2  #define SIMPLE_CLASS_H
3
4  class SimpleClass{
5      public:
6          void setI(int i_val);
7          int getI();
8      private:
9          int i;
10 };
11 #endif

```

The `SimpleClass` declaration leaves out the constructor, copy constructor, copy assignment operator, and destructor. The only two interface functions provided are a mutator function to set the value of `i` and an accessor function to get the value of `i`.

The implementation file `simpleclass.cpp` is shown in example 11.14.

```

1  #include "simpleclass.h" 11.14 simpleclass.cpp
2
3  void SimpleClass::setI(int i_val){ i = i_val;}
4  int SimpleClass::getI(){return i;}

```

Example 11.15 gives a `main()` function showing each of the special functions being tested, and the results of running the program are shown in figure 11-7.

Referring to example 11.15, when the `SimpleClass` object `s1` is created on line 6 the compiler-supplied constructor is utilized to initialize `s1`'s instance attribute `i`. But, what value will `i` be initialized to? The answer is found in figure 11-7. It is not initialized to anything; `s1`'s `i` attribute contains the garbage value found in memory when it was created. This is the default initialization behavior.

On line 8 the mutator function `setI()` is used to set the value of `s1`'s `i` attribute. The accessor function is used on line 9 to see if the change was successful, and it was.

11.15 main.cpp

```

1  #include <iostream>
2  #include "simpleclass.h"
3  using namespace std;
4
5  int main() {
6      SimpleClass s1;
7      cout<<"s1 = "<<s1.getI()<<endl;
8      s1.setI(5);
9      cout<<"s1 = "<<s1.getI()<<endl<<endl;
10
11     SimpleClass s2(s1);
12     cout<<"s2 = "<<s2.getI()<<endl<<endl;
13
14     SimpleClass s3;
15     cout<<"s3 = "<<s3.getI()<<endl;
16     s3 = s1;
17     cout<<"s3 = "<<s3.getI()<<endl;
18     return 0;
19 }

```

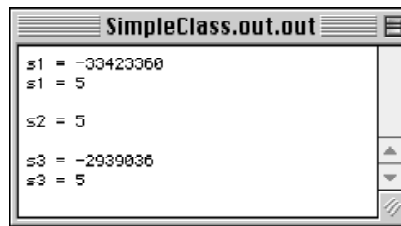


Figure 11-7: Results of Running Example 11.15

Next, a new SimpleClass object named s2 is created on line 11 using the compiler-supplied copy constructor and the s1 object as a guide. On line 14 a new object named s3 is created and on line 16 the compiler-supplied copy assignment operator is used to assign the value of s1 to s3. This works because SimpleClass objects contain one simple object of a fundamental data type. The type of copying performed by the default copy constructor and copy assignment operator is referred to as a shallow copy. Shallow copy works for simple data types, but not for complex objects with pointers to other objects.

When the program ends the compiler-supplied destructor is called for each of the SimpleClass objects but this is not evidenced by the output shown in figure 11-7.

Quick Summary

There are four special functions: default constructor, copy constructor, copy assignment operator, and destructor. The default constructor is a constructor that has either no parameters, or all parameters have default values so it can be called with no arguments. The copy constructor is used to create new objects from existing objects. The copy assignment operator sets the attributes of an existing object to the attribute values of another existing object. The destructor is used to tear down or destroy an object when it is no longer needed by the program.

Default, compiler-supplied versions of these functions may not perform as you intend for your class objects so you should implement each special function to ensure proper special function behavior. The objective of implementing the special functions explicitly is to have well-behaved objects.

ACCESSOR AND MUTATOR FUNCTIONS

Generally speaking, if a function is not one of the special member functions providing creation, copy, assignment, or destruction services, then it is an accessor or mutator function. TestClass has three such functions already

implemented: `getConstVal()`, `setI()`, and `getI()`. These functions deal exclusively with either setting or getting the value of a particular instance attribute. In the case of `TestClass`, these three public functions, along with the special functions, constitute the range of behavior expected of a `TestClass` object.

ACCESSOR FUNCTIONS

Accessor functions provide behavior while preserving the state of the object. Accessor functions may simply return the value of an object or class attribute, or it may base its behavior on the state of one or more object or class attributes and return something different, like a status value, or return nothing at all. The key concept is that an accessor function can be called to perform some operation while not messing with things inside the object.

Accessor functions can be written in many ways to reflect the behavior they produce. If a function is to simply return an attribute's value the function name can either begin with the word `get` followed by the name of the attribute whose value it is getting. Another way of writing the function is to leave off the word `get` and simply use the name of the attribute. This method actually produces easy-to-read code. Compare the following code samples:

```
if(obj.i() == some_value){//do something}
if(obj.getI() == some_value){//do something}
```

RETURNING BOOLEAN VALUES

If an accessor function returns a boolean value then by convention its name can start with the word `is`. For example:

```
if(obj.isTempHigh()){//do something}
```

MUTATOR FUNCTIONS

Mutator functions provide behavior and change the state of the object in the process. A simple mutator function, like the one named `setI()` in `TestClass`, may only make a change to one class or object attribute and do nothing more. Complex mutator functions may make bold changes to an object's state in order to provide complex behavior. Simple mutator functions can begin with the word `set`, followed by the name of the attribute's value they are changing. Complex mutator functions should be named to reflect both the behavior they produce and the values they change. Here are a few examples:

```
void setFirstName(char* f_name);
void changeLastName(char* l_name);
float addFuelAndReturnLevel(float fuel_to_add);
```

Quick SUMMARY

Accessor functions implement object behavior without changing the state of an object. Mutator functions implement object behavior while at the same time changing an object's state.

Using Access Specifiers To Control Horizontal Member Access

There are three access specifiers: `public`, `protected`, and `private`. They are all used to control both horizontal and vertical access to an object's data and function members. This section deals exclusively with horizontal access; vertical access is discussed in detail in chapter 13.

THE CONCEPT OF HORIZONTAL ACCESS

Horizontal access is the access an object of a particular class type has to the data members and member functions of another object of a different class type. Figure 11-8 gives an illustration.

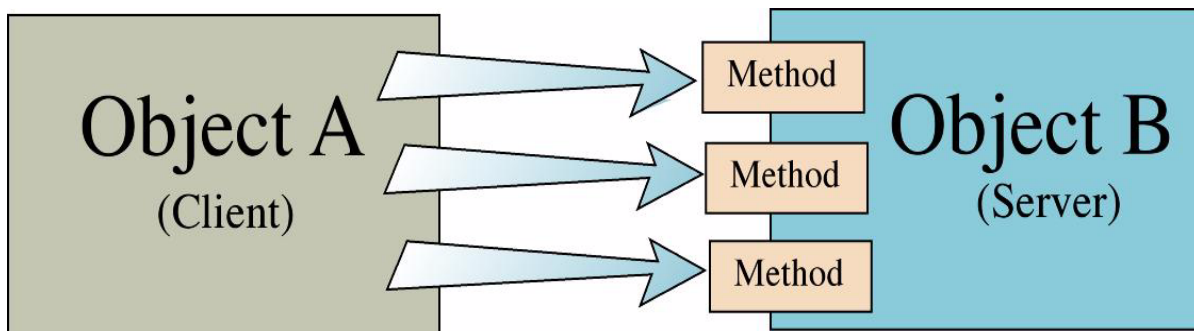


Figure 11-8: Horizontal Access

Object A could actually be something other than an object. For instance, when in a `main()` function, an object of, say, `SimpleClass` is created and one of `SimpleClass`'s public functions is invoked, then the `main()` function is horizontally accessing an object of type `SimpleClass`. The object requesting the access is considered the client object; the object providing the access is considered the server object.

DATA ENCAPSULATION

Data encapsulation is the act of hiding an object's data members from the outside world. As shown in figure 11-8, access to the functionality of an object should be through a set of functions or methods, such as accessor functions, mutator functions, or one of the special functions. At no time should a client object be allowed to muck around with a server object's private parts. The only information a client object needs to know about a server object to use it effectively is its public interface. A set of public interface functions is referred to as an Application Programming Interface (API).

ACCESS SPECIFIERS

The three access specifiers are briefly discussed below in the context of horizontal access.

The Public Access Specifier

Public accessibility enables horizontal access. Any data member or member function declared as being public is horizontally accessible by client objects. Generally speaking, the only parts of an object that should have public accessibility are its authorized public interface functions.

The Protected Access Specifier

Protected accessibility prevents horizontal access. Horizontally speaking, protected access is the same as private access.

The Private Access Specifier

Private accessibility prevents horizontal access.

OVERLOADING CLASS MEMBER FUNCTIONS

Class member functions can be overloaded just like ordinary functions. Three of the four special member functions can be overloaded with the only exception being the destructor. In this section I will introduce you to member function overloading and show you a few examples of how you do it and, more importantly, demonstrate to you why you would want to overload a member function.

FUNCTION SIGNATURES

Class member functions have function signatures just like ordinary functions. A function's signature is determined by the number and type of its parameters. An overloaded member function is a function having the same name as another member function but a different function signature. The following functions are overloaded:

```
float computeSum(float a, float b);
float computeSum(float array[]);
```

Both functions have the same name, `computeSum()`, but the first function takes two floats as arguments while the second function takes an array of floats. You can overload a function name as many times as required, which leads to the obvious question.

WHY WOULD YOU WANT TO OVERLOAD MEMBER FUNCTIONS?

There are several good reasons to overload class member functions. The primary reason to overload is to obtain different functionality via one function name. Take for example a class constructor function. By overloading a class constructor you allow the creation of class objects in different ways. An alternative to overloading the constructor is to provide default values for all constructor parameters. This creates a default constructor that can be called with no arguments or with as many arguments as required. You saw an example of this in the constructor section above. The presence of a properly done default constructor generally eliminates the need to overload the constructor for the purpose of initializing objects in different ways.

Another reason to overload member functions is to provide one set of class public interface functions and another set of functions having the same names as a private interface. These private interface functions would be available only to other member functions within the class.

Let us take a look at an example of a class with overloaded member functions. Example 11.16 gives the header file for a class named `Foo`.

The `Foo` class does not do much but it does a nice job of illustrating some important concepts. A `Foo` object will have two attributes: an integer `i` and a float `f`. There is a class-wide static variable named `foo_count` which will be used to keep track of how many `Foo` objects exist. In addition to the private attributes there are two private member functions named `printI()` and `printF()`. Because these functions are declared private they, like the private attributes, are only accessible by other member functions. In other words, they are not horizontally accessible.

Two public functions `incrementI()` and `incrementF()` are overloaded. Calling either with no parameters will result in the associated attribute being incremented by one. Calling either function with an argument will result in the associated attribute being incremented by the argument's value.

Example 11.17 gives the `Foo` class implementation file. The two private member functions `printI()` and `printF()` are called by the `printAttributes()` function located on line 38. The `foo_count` static variable is used to keep track of the number of `Foo` objects there are in existence. Notice how it is used in the constructor, copy constructor and destructor. Keeping track of the number of objects comes in handy in many ways.

Example 11.18 shows `Foo` class objects being used in a `main()` function. Figure 11-9 shows the output obtained by running example 11.18.

```
1  #ifndef _FOO_H
2  #define _FOO_H
3
4  class Foo{
5      public:
6          Foo(int ival = 0, float fval = 0.0);
7          Foo(Foo& f_object);
8          Foo& operator=(Foo& rhs);
9          ~Foo();
10         void printAttributes();
11         void setI(int ival);
12         void setF(float fval);
13         void incrementF();
14         void incrementF(float increment_value);
15         void incrementI();
16         void incrementI(int increment_value);
17
18     private:
19         static int foo_count;
20         int i;
21         float f;
22         void printI();
23         void printF();
24 };
25 #endif
```

11.16 *foo.h*

```
1  #include "foo.h"
2
3  int main(){
4      Foo f1(1, 2.5), f2(f1), f3;
5      f3 = f1;
6
7      f1.printAttributes();
8      f2.printAttributes();
9      f3.printAttributes();
10
11     f1.incrementI();
12     f1.incrementF();
13     f1.printAttributes();
14
15     f1.incrementI(5);
16     f1.incrementF(5.5);
17     f1.printAttributes();
18
19     return 0;
20 }
```

11.18 *main.cpp*

11.17 foo.cpp

```

1  #include "foo.h"
2  #include <iostream>
3  using namespace std;
4
5  int Foo::foo_count = 0;
6
7  Foo::Foo(int ival, float fval): i(ival), f(fval){
8
9      if(++foo_count == 1)
10         cout<<"There is "<<foo_count<<" foo object."<<endl;
11         else
12             cout<<"There are "<<foo_count<<" foo objects."<<endl;
13     }
14
15     Foo::Foo(Foo& f_object){
16         i = f_object.i;
17         f = f_object.f;
18
19         if(++foo_count == 1)
20             cout<<"There is "<<foo_count<<" foo object."<<endl;
21             else
22                 cout<<"There are "<<foo_count<<" foo objects."<<endl;
23     }
24
25     Foo& Foo::operator=(Foo& rhs){
26         i = rhs.i;
27         f = rhs.f;
28         return *this;
29     }
30
31     Foo::~Foo(){
32         if(--foo_count == 1)
33             cout<<"There is "<<foo_count<<" foo object."<<endl;
34             else
35                 cout<<"There are "<<foo_count<<" foo objects."<<endl;
36     }
37
38     void Foo::printAttributes(){
39         printI();
40         printF();
41     }
42
43     void Foo::setI(int ival){
44         i = ival;
45     }
46
47     void Foo::setF(float fval){
48         f = fval;
49     }
50
51     void Foo::incrementF(){
52         f += 1.0;
53     }
54     void Foo::incrementF(float increment_value){
55         f += increment_value;
56     }
57     void Foo::incrementI(){
58         ++i;
59     }
60
61     void Foo::incrementI(int increment_value){
62         i += increment_value;
63     }
64
65     void Foo::printI(){
66         cout<<"The value of i = "<<i<<endl;
67     }
68
69     void Foo::printF(){
70         cout<<"The value of f = "<<f<<endl;
71     }

```

```

OverloadedFunctions.out.out
There is 1 foo object.
There are 2 foo objects.
There are 3 foo objects.
The value of i = 1
The value of f = 2.5
The value of i = 1
The value of f = 2.5
The value of i = 1
The value of f = 2.5
The value of i = 2
The value of f = 3.5
The value of i = 7
The value of f = 9
There are 2 foo objects.
There is 1 foo object.
There are 8 foo objects.

```

Figure 11.9: Results of Running Example 11.18

SEPARATING A CLASS'S INTERFACE FROM ITS IMPLEMENTATION

The public member functions declared in a class declaration form the authorized interface to a particular class. Class declarations should be placed in separate header files, thus separating the class interface from its implementation. There are many reasons why you will want to do this and a few of them are discussed below.

MANAGE Physical Complexity

Class header files should be named to reflect the class declaration they contain. Take the Foo class as an example. Its declaration appears in a file named foo.h. Its implementation appears in a file named foo.cpp. It is admittedly hard to see the utility in doing this for small numbers of classes but when the complexity of your software project grows, so too grows the number of classes you will have to keep track of and hunt down for additions and modifications. You will see an example of this in the next chapter.

ALLOW THE CREATION OF CODE LIBRARIES

Keeping class declarations in separate header files allows you to create code libraries. These libraries can take the form of a dynamic linked library (DLL) or static library. The benefit to creating code libraries is that you can keep your secret algorithm to yourself while allowing others to benefit from its use. You simply create the library, keep your implementation file to yourself, and distribute the library with the header file. Anyone wishing to use your class library simply includes your header file and links to your library code.

A COMPLETE EXAMPLE: CLASS PERSON

Class Person is a complete example that includes most of the class functionality described in this chapter. Notable exceptions are the lack of overloaded or private member functions. Person objects use dynamic memory allocation to hold string values for a person's name. Because a person object dynamically allocates memory a person destructor must release the memory when a person object is destroyed. The Person class diagram is given in figure 11-10.

Example 11.19 gives the code for the person.h header file. Example 11.20 shows Person objects being created and used in a main() function. Example 11.21 gives the code for the person.cpp file. Showing the use of Person class objects after showing the class declaration emphasizes the fact that to use an object requires knowledge of its interface only, not its implementation.

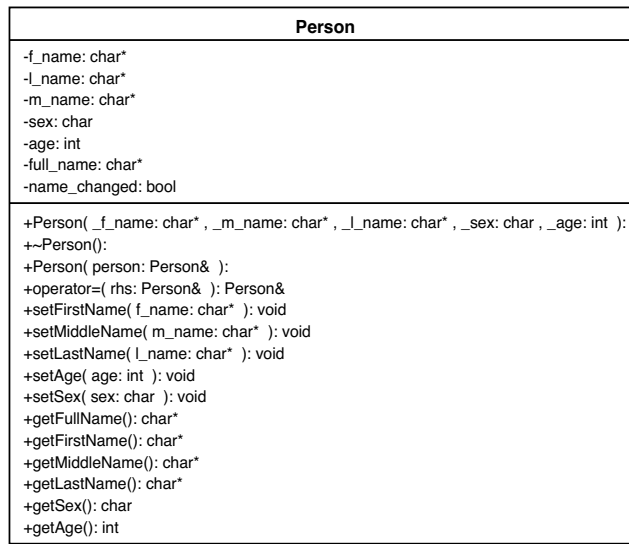


Figure 11-10: Person Class Diagram

```

1  #ifndef  __Person_H
2  #define  __Person_H
3
4
5  class Person{
6      public:
7          Person(char* _f_name = "John", char* _m_name = "M",
8                 char* _l_name = "Doe", char _sex = 'M', int _age = 18);
9          ~Person();
10         Person(Person& person);
11         Person&operator=(Person& rhs);
12         void setFirstName(char* f_name);
13         void setMiddleName(char* m_name);
14         void setLastName(char* l_name);
15         void setAge(int age);
16         void setSex(char sex);
17         char* getFullName();
18         char* getFirstName();
19         char* getMiddleName();
20         char* getLastName();
21         char getSex();
22         int getAge();
23
24     private:
25         char* f_name;
26         char* l_name;
27         char* m_name;
28         char sex;
29         int age;
30         char* full_name;
31         bool name_changed;
32
33 };
34 #endif

```

11.19 person.h

11.21 person.cpp

```

1  #include "person.h"
2  #include "person.h"
3  #include <string.h>
4
5  Person::Person( char*  _f_name, char*  _m_name, char*  _l_name, char _sex, int _age
6  ):sex(_sex),
7      age(_age), name_changed(true), full_name(NULL){
8      f_name = new char[strlen(_f_name)+1];
9      strcpy(f_name, _f_name);
10
11     m_name = new char[strlen(_m_name)+1];
12     strcpy(m_name, _m_name);
13
14     l_name = new char[strlen(_l_name)+1];
15     strcpy(l_name, _l_name);
16 }
17 }
18
19 Person::~Person(){
20     delete[] f_name;
21     delete[] m_name;
22     delete[] l_name;
23     delete[] full_name;
24 }
25
26 Person::Person(Person& person):name_changed(true), full_name(NULL){
27     f_name = new char[strlen(person.f_name)+1];
28     strcpy(f_name, person.f_name);
29
30     m_name = new char[strlen(person.m_name)+1];
31     strcpy(m_name, person.m_name);
32
33     l_name = new char[strlen(person.l_name)+1];
34     strcpy(l_name, person.l_name);
35
36     sex = person.sex;
37     age = person.age;
38 }
39 }
40
41 Person& Person::operator=( Person& rhs ){
42     delete[] f_name;
43     f_name = new char[strlen(rhs.f_name)+1];
44     strcpy(f_name, rhs.f_name);
45
46     delete[] m_name;
47     m_name = new char[strlen(rhs.m_name)+1];
48     strcpy(m_name, rhs.m_name);
49
50     delete[] l_name;
51     l_name = new char[strlen(rhs.l_name)+1];
52     strcpy(l_name, rhs.l_name);
53
54     name_changed = true;
55
56     return *this;
57 }
58 }
59
60 void Person::setFirstName(char* f_name){
61     delete[] this->f_name;
62     this->f_name = new char[strlen(f_name)+1];
63     strcpy(this->f_name, f_name);
64
65     name_changed = true;
66 }
67 }
68
69 void Person::setMiddleName(char* m_name){
70     delete[] this->m_name;
71     this->m_name = new char[strlen(m_name)+1];
72     strcpy(this->m_name, m_name);
73
74     name_changed = true;
75 }
76 }

```

```

77 void Person::setLastName(char* l_name){
78     delete[] this->l_name;
79     this->l_name = new char[strlen(l_name)+1];
80     strcpy(this->l_name, l_name);
81
82     name_changed = true;
83 }
84
85 void Person::setAge( int age ){
86     this->age = age;
87 }
88
89
90 void Person::setSex( char sex ){
91     this->sex = sex;
92 }
93
94 char* Person::getFullName(){
95
96     if((full_name != NULL) && (!name_changed))
97         return full_name;
98     else {
99
100         delete[] full_name;
101         full_name = new char[(strlen(f_name) + strlen(m_name)
102                               + strlen(l_name) + 7)];
103
104         strcpy(full_name, f_name);
105         strcat(full_name, " ");
106         strcat(full_name, m_name);
107         strcat(full_name, " ");
108         strcat(full_name, l_name);
109
110         name_changed = false;
111         return full_name;
112     }
113
114 }
115
116 char* Person::getFirstName(){ return f_name;}
117
118 char* Person::getMiddleName(){ return m_name;}
119
120 char* Person::getLastName(){ return l_name;}
121
122
123 char Person::getSex(){ return sex;}
124
125
126 int Person::getAge(){ return age;}

```

11.21 person.cpp
continued

SUMMARY

A class declaration introduces a new data type. A class type object is different from a fundamental data type object because of the expanded role it can play in a software system.

There are four special functions: default constructor, copy constructor, copy assignment operator, and destructor. The default constructor is a constructor that has either no parameters, or all parameters have default values so it can be called with no arguments. The copy constructor is used to create new objects from existing objects. The copy assignment operator sets the attributes of an existing object to the attribute values of another existing object. The destructor is used to tear down or destroy an object when it is no longer needed by the program. It is especially important to use the destructor to release any system resources an object may have access during its lifetime. An example of such a system resource is dynamic memory.

Default, compiler-supplied versions of these functions may not perform as you intend for your class objects so you should implement each special member function to ensure proper object behavior. The objective of implementing the special functions explicitly is to have well-behaved objects.

```

1  #include <iostream>
2  #include "person.h"
3  using namespace std;
4
5  int main() {
6
7      Person p1;
8
9      cout<<p1.getFirstName()<<endl;
10     cout<<p1.getFullName()<<endl;
11     p1.setFirstName("Bob");
12     p1.setMiddleName("Raymond");
13     p1.setLastName("Basmahranian");
14
15     cout<<p1.getFullName()<<" "<<p1.getAge()<<" "<<p1.getSex()<<endl;
16     cout<<p1.getFullName()<<" "<<p1.getAge()<<" "<<p1.getSex()<<endl;
17     cout<<p1.getFullName()<<" "<<p1.getAge()<<" "<<p1.getSex()<<endl;
18     cout<<p1.getFullName()<<" "<<p1.getAge()<<" "<<p1.getSex()<<endl;
19
20     Person p2(p1);
21
22     cout<<p1.getFullName()<<" "<<p1.getAge()<<" "<<p1.getSex()<<endl;
23     cout<<p2.getFullName()<<" "<<p2.getAge()<<" "<<p2.getSex()<<endl;
24     cout<<p2.getFullName()<<" "<<p2.getAge()<<" "<<p2.getSex()<<endl;
25
26     p2.setFirstName("Richard");
27     p2.setMiddleName("Warren");
28     p2.setLastName("Miller");
29
30     cout<<p2.getFullName()<<" "<<p2.getAge()<<" "<<p2.getSex()<<endl;
31     cout<<p1.getFullName()<<" "<<p1.getAge()<<" "<<p1.getSex()<<endl;
32
33     Person p3;
34
35     p1=p3;
36
37     cout<<p1.getFullName()<<" "<<p1.getAge()<<" "<<p1.getSex()<<endl;
38     cout<<p2.getFullName()<<" "<<p2.getAge()<<" "<<p2.getSex()<<endl;
39     cout<<p3.getFullName()<<" "<<p3.getAge()<<" "<<p3.getSex()<<endl;
40
41     return 0;
42 }

```

11.20 main.cpp

Accessor functions implement object behavior without changing the state of an object. Mutator functions implement object behavior while at the same time changing an object's state.

There are three member access specifiers: public, protected, and private. The public access specifier allows horizontal access to class and instance data members and member functions. The protected access specifier allows access vertically but blocks horizontal access. Vertical access is discussed in detail in chapter 13. The private access specifier blocks both horizontal and vertical access.

Data encapsulation is the act of declaring data members private and supplying public interface functions to manipulate or access those data members.

Class member functions can be overloaded, and class member functions can call other class member functions. Sometimes it is a good idea to supply private member functions that are called by the public class interface functions.

It is good programming practice to declare classes in separate header files. Doing so affords better control of a project's physical complexity and enables the creation and distribution of class library code.

Skill Building Exercises

1. **Create Person Project:** Create a project in your IDE and test the Person class code given in examples 11.19 - 11.21. Expand on the code in the main.cpp file and create additional Person objects with different attributes. Use the new operator to create Person objects using dynamic memory allocation.
2. **Class Bar:** Declare a class named Bar using the Foo class in this chapter as a guide. Declare a constructor, copy constructor, copy assignment operator, and a destructor. Declare one or more private data members and any necessary public accessor and mutator functions. Put the class declaration in a separate header file named bar.h, and the

class function implementations in a file named `bar.cpp`. Write a `main()` function and test your `Bar` objects.

3. **Research:** Do further research on different UML diagrams. Specifically, learn about statechart diagrams, object diagrams, sequence diagrams, and use case diagrams.
4. **Obtain UML Tool:** Obtain a shareware UML tool or download a commercial tool for limited use. Use the `Person` class as a guide and define a class with the same name using the UML design tool. When you have finished the design generate the source code and examine the resulting output. How does it compare with the original `Person` code?
5. **Object Modeling:** Examine the world around you and select several candidate objects to model in software. Determine the type of attributes and behavior each object has and translate your findings into one or more class declarations. Implement one or more of your classes and write a `main()` driver function to test your classes by creating and using objects. Refer to the examples in this chapter as a guide. Make sure to implement all the special class functions. Be sure to encapsulate data by declaring class data members private and class member functions public.
6. **Array of Person Pointers:** Create an array of pointers to `Person` objects. Dynamically allocate `Person` objects using the `new` operator and assign their addresses to the array elements. Using the array elements and the “->” operator, call functions on `Person` objects.

SUGGESTED PROJECTS

1. **Robot Rat Redesign:** Redesign the `RobotRat` project of chapter 3 so that several `RobotRat` objects can be created and moved around on one floor at the same time.
2. **Computer Simulator:** Convert the computer simulator described in chapter 8, suggested project 4, to a class.
3. **Research:** Study your IDE documentation and convert the `Person` class into either a static library or dynamically linked library. Use the library in another project to create `Person` objects.

SELF TEST QUESTIONS

1. What is the primary difference between primitive data type objects and class objects?
2. List the four special class member functions and describe the function of each. Which three of the four special functions can be overloaded?
3. List the three access specifiers. Describe how each access specifier affect horizontal access.
4. In your own words define the term horizontal access.
5. What is the difference between accessor and mutator functions?
6. What is the purpose of the `this` pointer?
7. What is the difference between a static class-wide variable and an instance variable?
8. Describe how member functions can be overloaded.

9. List and discuss two benefits of separating class interface from class definition. Can you think of any other benefits?
10. How would you access an instance variable masked by a local function variable of the same name?

REFERENCES

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language*. Second Edition. Prentice Hall, Englewood Cliffs, New Jersey, 1988. ISBN: 0-13-110370-9

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994. ISBN: 0-8053-5340-2

Paul J. Lucas. *The C++ Programmer's Handbook*. Prentice Hall P T R, Englewood Cliffs, New Jersey, 1992. ISBN: 0-13-118233-1

Robert C. Martin. *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice Hall, Englewood Cliffs, New Jersey, 1994. ISBN: 0-13-203837-4

Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Englewood Cliffs, New Jersey, 2002. ISBN: 0-13-092569-1

NOTES

CHAPTER 12



SAILOR IN THE STRAW HAT

COMPOSITIONAL DESIGN

LEARNING OBJECTIVES

- Explain how to design complex classes using user-defined abstract data types
- Describe the concept of aggregation
- State the relationship between aggregation and object lifetime
- Explain the difference between contains by value and contains by reference
- Describe the concept of simple aggregation
- Describe the concept of composite aggregation,
- Explain how to implement message passing between objects
- Explain how to utilize pointers and references in the design of complex classes
- Explain how to express aggregation in UML notation
- State the purpose and use of a UML sequence diagram
- Demonstrate your ability to use simple and composite aggregation to implement C++ programming projects

INTRODUCTION

Rarely is an application comprised of just one class. In reality, applications are constructed from many classes. This chapter introduces you to the concepts and terminology associated with building complex application behavior from a collection of classes. This is referred to as compositional design or design by composition.

The study of compositional design is the study of aggregation and containment. You will learn the two primary aggregation associations: simple and composite.

In this chapter you will also learn how to extend the UML class diagram to describe the static relationship between classes in a complex application. To do this you will need to know how to express simple and composite aggregation visually.

The study of aggregation also entails learning how the whole class accesses the services of its part classes. The concept of message passing and sequencing will be demonstrated by introducing you to a new type of UML diagram known as the sequence diagram.

MANAGING PHYSICAL COMPLEXITY

From this point forward the applications you will see as examples in this chapter will be more complex than anything you have seen before. Now, more than ever, you will come to rely on the technique of separating the class interface from its implementation. Placing class declarations in separate header files significantly aids your ability to manage the physical complexity of large C++ projects.

AGGREGATION

Class behavior can be implemented by building upon the behavior provided by other classes. In other words, a class type object can contain other class type objects. A class built upon the functionality of other classes is referred to as an aggregate class type or aggregation. There are two types of aggregation: simple and composite. Aggregation is also referred to as a “has-a” or a “uses-a” relationship between the whole class and its part classes where the whole or aggregate class uses the services of its part classes.

Simple vs. Composite Aggregation

The aggregation relationship is expressed in terms of the whole class and the part class. Given two classes, class B and class A, if class B contains class A then class B is referred to as the whole class and class A is referred to as the part class.

The Relationship Between Aggregation and Object Lifetime

The primary difference between simple and composite aggregation lies in who controls the lifetime of the object.

Simple Aggregation

A simple aggregate object does not control the lifetimes of its part objects. Part objects involved in simple aggregations can be associated with more than one aggregation.

Composite Aggregation

A composite aggregate object controls the lifetimes of its part objects. Part objects involved in composite aggregations cannot be associated with more than one aggregation.

AGGREGATION EXAMPLE CODE

Example 12.1 gives the class declarations for class A that will be used to demonstrate the two different types of aggregation. Class A will be the contained or part class. Example 12.2 shows the implementation code for class A which would be in a file named a.cpp

```

1  #ifndef _CLASS_A_H                                12.1 a.h
2  #define _CLASS_A_H
3
4  class A {
5      public:
6          A();
7          ~A();
8  };
9  #endif

```

```

1  #include <iostream>                                12.2 a.cpp
2  using namespace std;
3  #include "a.h"
4
5  A::A() {
6      cout<<"An object of type A created!"<<endl;
7  }
8
9  A::~A() {
10     cout<<"An object of type A destroyed!"<<endl;
11 }

```

Class A is relatively simple. All it does is print a message to the screen when a class A object is created and destroyed. These messages will come in handy for learning about the behavior of aggregate objects.

COMPOSITE AGGREGATION EXAMPLE

Example 12.3 gives the class declaration for class B containing a class A object. Class B is the aggregate or whole class and class A is the part class. Said another way, an object of type B has an object of type A.

```

1  #ifndef _CLASS_B_H                                12.3 b.h
2  #define _CLASS_B_H
3  #include "a.h"
4
5  class B{
6      public:
7          B();
8          ~B();
9      private:
10     A its_a; ← Object of type A declared
11 };
12 #endif

```

Example 12.4 shows the implementation code for class B. It looks exactly like the implementation code for class A except the name of the class has changed.

Example 12.5 gives the code for a main() function that creates a class B object and figure 12-1 shows the results obtained from running the program.

Let us pause here for a moment of discussion. Study figure 12-1. Notice how the A object's constructor was

12.4 b.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "b.h"
4
5  B::B() {
6      cout<<"An object of type B created!"<<endl;
7  }
8
9  B::~B() {
10     cout<<"An object of type B destroyed!"<<endl;
11 }

```

12.5 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "b.h"
4
5  int main() {
6      B b1;
7      return 0;
8  }

```

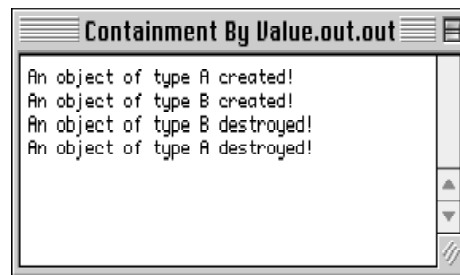


Figure 12-1: Results of Running Example 12.5

called prior to the B object's constructor. From this experiment we can deduce that ordinary part objects will be created before the whole aggregate object is created. An important distinction to make here is that object creation is not complete until the constructor has finished executing. The important thing to take away from this example is that the life of a part object is controlled by the composite aggregate whole object. Notice in example 12.5 that only a B object is created. This causes the creation of B's part object. For an object with part members to be fully created, all of its part members must first be created.

ANOTHER COMPOSITE AGGREGATION EXAMPLE

The B class will be slightly modified to show another why to create aggregate objects using pointers. Example 12.6 gives the code for the modified class B declaration.

The only change made to the B class declaration appears on line 10. B's private data member was changed from an A object to a pointer to an A object. The name of the identifier was also changed to reflect its new role as a pointer.

Example 12.7 shows the modified class B implementation code.

Several changes were made to this file. First, the code on line 6 was added to the constructor to explicitly create the A object and assign its address to its_a_ptr. The second change is to the destructor. The code on line 11 was added to explicitly call the A object's destructor by deleting the pointer.

The result of running example 12.5 again is shown in figure 12-2.

```

1  #ifndef _CLASS_B_H
2  #define _CLASS_B_H
3  #include "a.h"
4
5  class B{
6  public:
7      B();
8      ~B();
9  private:
10     A *its_a_ptr;
11 };
12 #endif

```

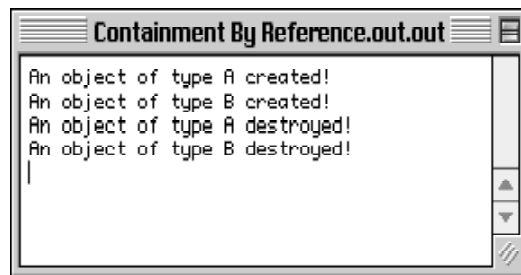
12.6 b.h

```

1  #include <iostream>
2  using namespace std;
3  #include "b.h"
4
5  B::B(){
6      its_a_ptr = new A();
7      cout<<"An object of type B created!"<<endl;
8  }
9
10 B::~B(){
11     delete its_a_ptr;
12     cout<<"An object of type B destroyed!"<<endl;
13 }

```

12.7 b.cpp



```

Containment By Reference.out.out
An object of type A created!
An object of type B created!
An object of type A destroyed!
An object of type B destroyed!
|

```

Figure 12-2: Results of Running Example 12.5 Again

The order of the messages in figure 12-2 is of minor importance. Special code is added to the constructor and destructor to create and destroy B's A object. Let us now take a look at a simple composite class.

Simple Aggregation Example

To demonstrate simple aggregation the A and B class files will be once again modified. Example 12.8 gives the source code for the revised a.h file.

The only change to the A class declaration is the addition of another public function on line 8 named sayHi(). The modified a.cpp file is shown in example 12.9.

The sayHi() function definition begins on line 13. All it will do is print a simple message to the screen. Now, in addition to the constructor and destructor messages, any object that contains an A object will be able to call the A object's sayHi() function. The modified B class declaration is given in example 12.10.

Two modifications were made to the B class declaration. A parameter is added to the B constructor function of type pointer to A. When a B object is created it will expect to be passed the address of an A object. The second modi-

```

1  #ifndef _CLASS_A_H                                12.8 a.h
2  #define _CLASS_A_H
3
4  class A {
5      public:
6          A();
7          ~A();
8          void sayHi();
9  };
10 #endif

```

```

1  #include <iostream>                                12.9 a.cpp
2  using namespace std;
3  #include "a.h"
4
5  A::A() {
6      cout<<"An object of type A created!"<<endl;
7  }
8
9  A::~~A() {
10     cout<<"An object of type A destroyed!"<<endl;
11 }
12
13 void A::sayHi() {
14     cout<<"Hi!"<<endl;
15 }

```

```

1  #ifndef _CLASS_B_H                                12.10 b.h
2  #define _CLASS_B_H
3  #include "a.h"
4
5  class B{
6      public:
7          B(A *a_ptr);
8          ~B();
9          void makeContainedObjectSayHi();
10     private:
11         A *its_a_ptr;
12     };
13 #endif

```

fication is the addition of one additional public function named `makeContainedObjectSayHi()`. This seems to be a little long winded for a function name but it accurately reflects the purpose of the function and hints at its intended behavior when called. The modified `b.cpp` implementation file is shown in example 12.11.

Several modifications were made to `b.cpp`. First, the code to create and destroy the A object from the constructor and destructor was removed. Since a pointer to an A object will be passed to a B object when one is created that code was no longer required. This emphasizes that the lifetimes of A objects are clearly not at the mercy of B objects. Next, the constructor was modified to add an initializer list to initialize the A class pointer data member named `its_a_ptr`. Lastly, the `makeContainedObjectSayHi()` function is implemented beginning on line 14. Notice that just a touch of error checking was introduced. If, for some reason, a B object is fed a NULL pointer when it is created, it would be a mistake to try and call any functions using `its_a_ptr` since it would be initialized to NULL. If `its_a_ptr` is not a NULL value then the `sayHi()` function is called on the contained-by-reference object via the shorthand pointer member access operator “->”.

```

1  #include <iostream>
2  using namespace std;
3  #include "b.h"
4  #include "a.h"
5
6  B::B(A *a_ptr):its_a_ptr(a_ptr){
7      cout<<"An object of type B created!"<<endl;
8  }
9
10 B::~~B(){
11     cout<<"An object of type B destroyed!"<<endl;
12 }
13
14 void B::makeContainedObjectSayHi(){
15     if(its_a_ptr != NULL)
16         its_a_ptr->sayHi();
17 }

```

12.11 b.cpp

All that is left now is to look at a main() function that uses the new versions of the A and B classes. Example 12.12 gives the code.

```

1  #include <iostream>
2  using namespace std;
3  #include "b.h"
4  #include "a.h"
5
6  int main(){
7      A a1;
8      a1.sayHi();
9      B b1(&a1);
10     b1.makeContainedObjectSayHi();
11     return 0;
12 }

```

12.12 main.cpp

Starting on line 7, an A object named a1 is created and on the next line the sayHi() function is called to demonstrate the existence of the A object outside of the B object. Next, a B object is created and its constructor is called with the address of the A object obtained by using the & operator. On line 10 the makeContainedObjectSayHi() function is called on the B object. This in turn calls the sayHi() function by using the pointer to the A object as shown on line 16 of example 12.11 above.

EXTENDING THE CLASS DIAGRAM

The UML class diagram can be used to show static relationships between classes in a software application. Figure 12-3 shows the UML class diagram for classes A and B as they appear in their final form in code examples 12.8 and 12.10.

The line between class B and Class A denotes an association. The open diamond placed at the class B end of the association denotes simple aggregation. The B class is the whole aggregate comprised of an A class part.

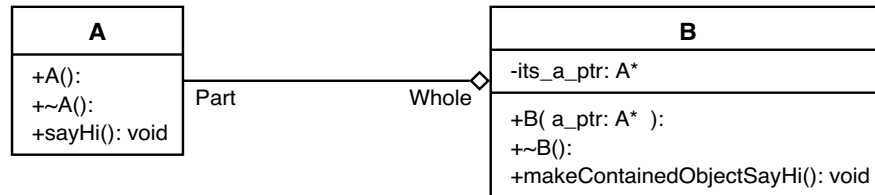


Figure 12-3: UML Diagram Illustrating Simple Aggregation

SEQUENCE DIAGRAMS

UML sequence diagrams are a great way to show graphically the order of object messaging activity in an application. Figure 12-4 shows a sequence diagram illustrating the messaging between the main() function, class B object b1, and class A object a1 as shown in the code example 12.12.

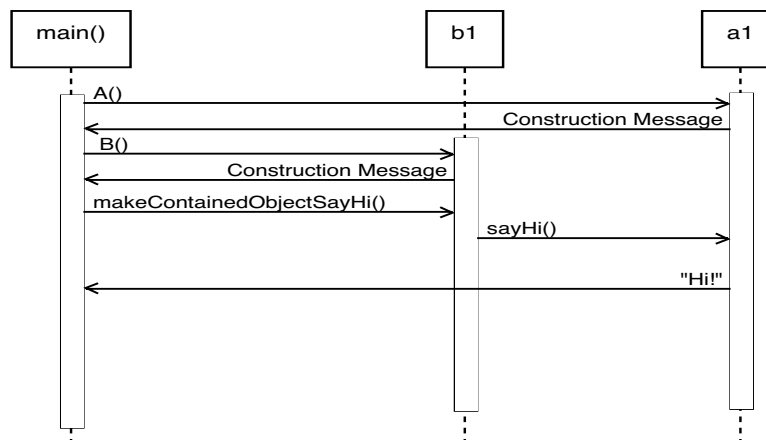


Figure 12-4: UML Sequence Diagram Illustrating Message Passing Between Objects

The main() function creates an A object and by doing so calls the A constructor. This is shown as an A() message being sent from main() to the a1 object. The result of the constructor call is the constructor message which is shown in the sequence diagram as a message from a1 back to the main() function labeled Constructor Message. Next, the b1 object is created in similar fashion. Once the b1 object is created the main() function sends it the makeContainedObjectSayHi() message. The b1 object processes this message by sending the sayHi() message to the a1 object (via the pointer). When the program terminates the destructors are called on the a1 and b1 objects. This is not shown in the diagram.

Quick SUMMARY

There are two forms of aggregation: simple and composite. In simple aggregation the whole object does not control the lifetime of its part objects. This means that part objects could play a role in more than one aggregate relationship. In composite aggregation the whole object controls the lifetime of its part objects. This means it controls their creation and destruction. Composite aggregates can be formed from simple part objects, part objects allocated dynamically, or from a combination of both. Composite aggregates control the lifetimes of their part objects; simple aggregates do not.

The UML class diagram can be used to illustrate aggregate relationships. A line between classes denotes an association. A diamond is placed at the composite end of the association line denoting aggregation. A hollow diamond indicates simple aggregation; a solid diamond indicates composite aggregation.

The UML sequence diagram is used to illustrate message passing between objects.

THE AIRCRAFT ENGINE SIMULATION: AN EXTENDED AGGREGATION EXAMPLE

It is time for a more complex example of compositional design. In this section you will study a software model of an aircraft jet engine. The name of the composite class is Engine and it contains several different part objects. Figure 12-5 shows the UML class diagram for the composite aggregate class Engine. As you can see from the diagram the diamonds are solid, indicating composite aggregation of each of the Engine's part objects.

THE PURPOSE OF THE ENGINE CLASS

Any modeling effort requires a set of simplifying assumptions so I will define a few to apply here. First, and primarily, although the Engine class and its supporting classes represent a complex set of associations between several classes, it is by no means a literal model of an engine, so you will have to use some imagination. Its purpose is to suggest how the implementation of an aggregation comprised of more than one or two classes might look in source code. To this end all the objects respond to messages with simple text messages printed to the screen. What is important to take away from this example is how you think of complex class design which should be in terms of objects and their interfaces. Lastly, the number of parts comprising the Engine aggregate are kept at a manageable five. This will give you a taste for the potential physical complexity you might encounter on larger object-oriented design projects.

AN ENGINE AND ITS PARTS

The Engine class is a composite aggregate object comprised of five user-defined abstract data types: FuelPump, OilPump, Compressor, TemperatureSensor, and OxygenSensor. An Engine's behavior is derived from the behavior of these parts. Let us take a closer look at the behavior of one of these parts. Figure 12-6 shows the class diagram for the FuelPump class.

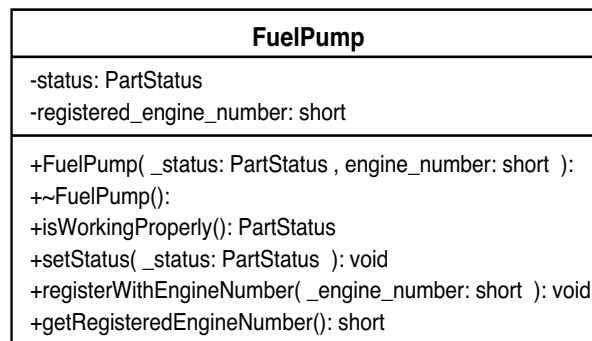


Figure 12-6: FuelPump Class

The public interface functions, indicated by the leading + sign in the class diagram, define the behavior you can expect from a FuelPump object. The constructor takes two parameters: `_status`, which is of type `PartStatus`, and `engine_number` which is of type `short`. The `short` type is a fundamental C++ data type. The `PartStatus` type is an enumeration declared for the purpose of this example. The `PartStatus` declaration is located in a file named `aircraftutils.h` shown in example 12.13.

```

1 #ifndef __AIRCRAFT_UTILITIES_H 12.13 aircraftutils.h
2 #define __AIRCRAFT_UTILITIES_H
3
4 enum PartStatus {NotWorking, Working};
5 enum EngineStatus {NotReady, Ready};
6
7 #endif

```

The `PartStatus` enum has two states: `NotWorking` and `Working`. When a `FuelPump` constructor is called the first argument must be of type `PartStatus` indicating whether the part is working or not. The second constructor parameter is the number of the engine to which the `FuelPump` object belongs.

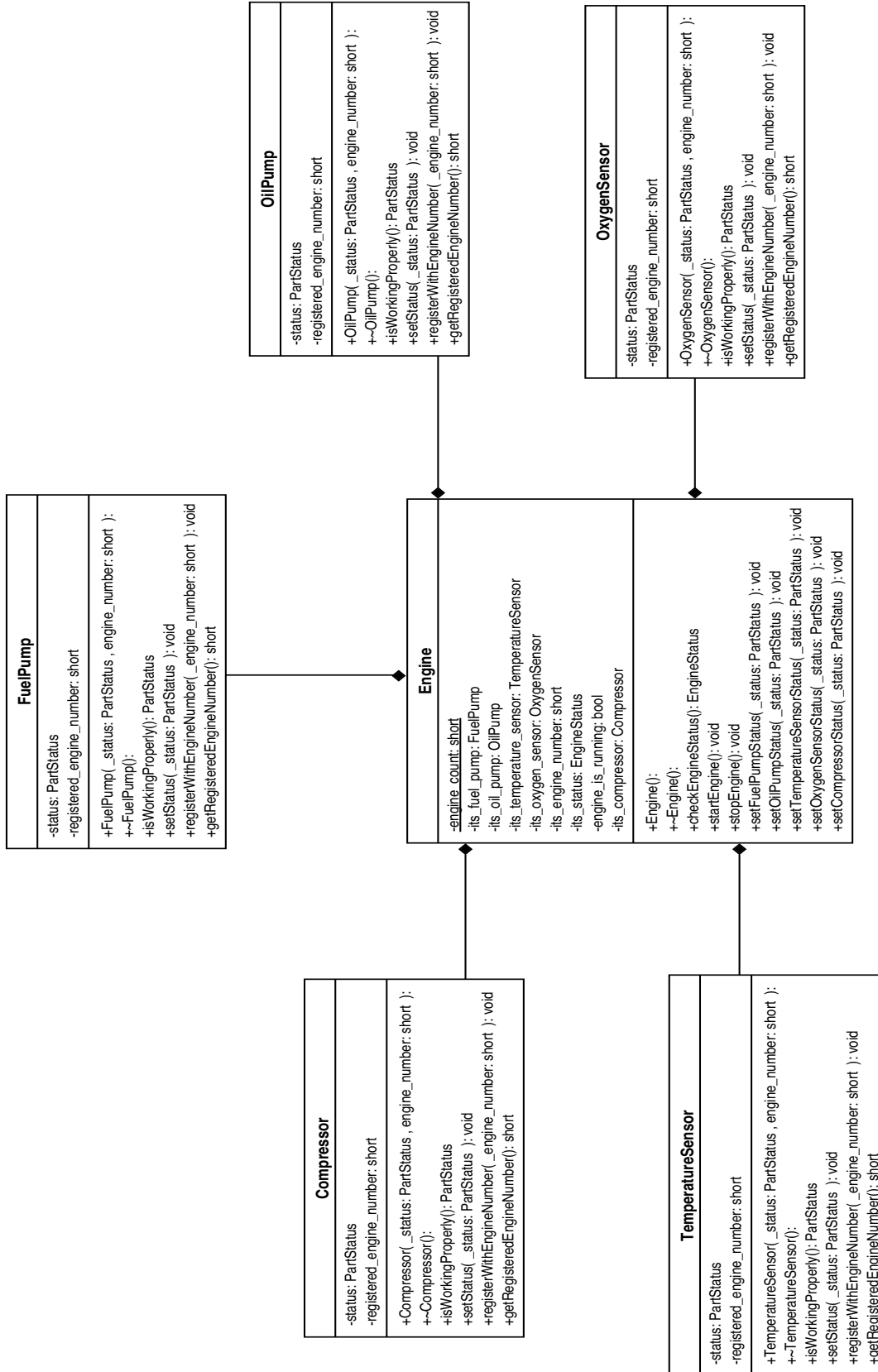


Figure 12-5: Engine Composite Aggregation Class Diagram

The other public functions of FuelPump indicate the other behaviors FuelPump objects can exhibit. You can determine if a fuel pump is working properly by sending it an `isWorkingProperly()` message; you can set a fuel pump's operational status by sending it a `setStatus()` message. You can register a fuel pump with a particular engine by sending it a `registerWithEngineNumber()` message, and lastly, you can determine to what engine number a fuel pump is registered to by sending it a `getRegisteredEngineNumber()` message. Essentially the public interface is a set of accessor and mutator functions that provide some rudimentary behavior.

All the other classes used in the Engine design have the same interface and therefore the same behavior. You may examine them more closely if you wish by studying figure 12-5.

The class declaration for the FuelPump class resides in a file named `fuelpump.h` and is shown in example 12.14.

```

1  #ifndef FUEL_PUMP_H
2  #define FUEL_PUMP_H
3  #include "aircraftutils.h"
4
5  class FuelPump {
6      public:
7          FuelPump(PartStatus _status = Working, short engine_number = 0);
8          ~FuelPump();
9          PartStatus isWorkingProperly();
10         void setStatus(PartStatus _status);
11         void registerWithEngineNumber(short _engine_number);
12         short getRegisteredEngineNumber();
13     private:
14         PartStatus status;
15         short registered_engine_number;
16 };
17
18 #endif

```

12.14 *fuelpump.h*

THE ENGINE CLASS

The class diagram for the Engine class is shown in figure 12-7. It too presents a set of public interface functions

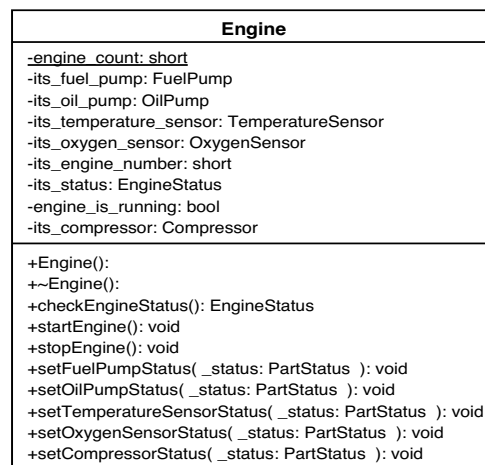


Figure 12-7: Engine Class Diagram

which define the type of behavior you can expect from an Engine object. Besides creating and destroying an Engine object, you can check its status, start, stop, and set the status of each of its parts. Since the Engine class is a composite aggregate it controls the life of each of its parts.

The Engine class contains one static class-scope data member named `engine_count`. Class-scope data members appear underlined in UML class diagrams. There are two more data members to note: `its_status` is of type `EngineStatus` which is an enumerated type declared in the `aircraftutils.h` file; `engine_is_running` is a boolean variable which is used to indicate whether the engine is running or not.

Note that all the Engine class data members are private. The only authorized way to manipulate engine objects is through the Engine class's public interface. Example 12.15 gives the source code for the Engine class declaration.

```

1  #ifndef ENGINE_H 12.15 engine.h
2  #define ENGINE_H
3  #include "aircraftutils.h"
4  #include "fuelpump.h"
5  #include "oxygensensor.h"
6  #include "oilpump.h"
7  #include "compressor.h"
8  #include "temperaturesensor.h"
9
10
11 class Engine{
12     public:
13         Engine();
14         ~Engine();
15         EngineStatus checkEngineStatus();
16         void startEngine();
17         void stopEngine();
18         void setFuelPumpStatus(PartStatus _status);
19         void setOilPumpStatus(PartStatus _status);
20         void setTemperatureSensorStatus(PartStatus _status);
21         void setOxygenSensorStatus(PartStatus _status);
22         void setCompressorStatus(PartStatus _status);
23
24     private:
25         static short      engine_count;
26         FuelPump          its_fuel_pump;
27         OilPump           its_oil_pump;
28         TemperatureSensor its_temperature_sensor;
29         OxygenSensor      its_oxygen_sensor;
30         Compressor        its_compressor;
31         short             its_engine_number;
32         EngineStatus      its_status;
33         bool              engine_is_running;
34 };
35 #endif

```

If you study the Engine class diagram in figure 12-7 you will find it matches closely what you see in the Engine class declaration above. A good UML design tool can do more than just draw pretty pictures; it can be used to flesh out the classes and their interfaces and to generate the framework code resulting from the design. Although the code shown above was not generated with a UML tool, it could easily have been.

Notice on lines 3 through 8 that all the required header files corresponding to each of the part classes are included.

Before we look at the Engine class implementation code let us take a look at the `main()` function and how Engine objects are created and used. Example 12.16 gives the source code for the `main.cpp` file used to run the aircraft engine program.

Two Engine objects are created on line 8, `e1` and `e2`. Next, the `checkEngineStatus()` message is sent to each Engine object. On line 13 the `stopEngine()` message is sent to the `e1` object. Since the engine has not yet been started this should result in a message saying something to that effect. Next, both engine objects are sent the `startEngine()` message.

```

1  #include <iostream>
2  using namespace std;
3
4  #include "engine.h"
5
6  int main()
7  {
8      Engine e1,e2;
9
10     e1.checkEngineStatus();
11     e2.checkEngineStatus();
12
13     e1.stopEngine();
14
15     e1.startEngine();
16     e2.startEngine();
17
18     e1.setCompressorStatus(NotWorking);
19     e1.checkEngineStatus();
20
21     return 0;
22 }

```

12.16 main.cpp

On line 18, the e1 object is sent the setCompressorStatus() message with the NotWorking enumeration state value as an argument. This should result in the status of the compressor associated with Engine object e1 being set to NotWorking. The result of this message is tested by sending a checkEngineStatus() message to the e1 engine object. When the program terminates all engine objects and their associated part objects are destroyed. Figure 12-8 shows the results of running this program.

```

Aircraft_Engine_Components.out.out
Fuel pump 1 working
Oil pump 1 working
Temperature sensor 1 working
Oxygen sensor 1 working
Compressor 1 working
Fuel pump 2 working
Oil pump 2 working
Temperature sensor 2 working
Oxygen sensor 2 working
Compressor 2 working
All engine number 1 components working properly.
All engine number 2 components working properly.
Engine 1 not running!
All engine number 1 components working properly.
Engine number 1 started.
All engine number 2 components working properly.
Engine number 2 started.
Engine number 1 malfunction!
Engine number 1 stopped.
Compressor 2 destroyed
Oxygen sensor 2 destroyed
Temperature sensor 2 destroyed
Oil pump 2 destroyed
Fuel pump 2 destroyed
Compressor 1 destroyed
Oxygen sensor 1 destroyed
Temperature sensor 1 destroyed
Oil pump 1 destroyed
Fuel pump 1 destroyed

```

Figure 12-8: Results of Running Example 12-16

Notice that the creation of each Engine object causes each part object to be created, as is indicated by the message each part object's constructor prints to the screen. You can trace each message produced in figure 12-8 as a result of sending messages to each Engine object in the main() function.

THE ENTIRE AIRCRAFT ENGINE SIMULATION PROJECT

The best way to study the workings of this project is to see all the code laid out before you. The Aircraft Engine simulation project contains a total of fourteen source files. First the header files: *aircraftutils.h*, *fuelpump.h*, *oilpump.h*, *tempraturesensor.h*, *oxygensensor.h*, *compressor.h*, and *engine.h*. Then, the class implementation files: *fuelpump.cpp*, *oilpump.cpp*, *temperaturesensor.cpp*, *oxygensensor.cpp*, *compressor.cpp*, and *engine.cpp*. Lastly, the *main.cpp* file contains the main() functions needed to get everything running. The source files will be presented in this order with no line numbering.

AIRCRAFTUTILS.H

aircraftutils.h

```
#ifndef __AIRCRAFT_UTILITIES_H
#define __AIRCRAFT_UTILITIES_H

enum PartStatus {NotWorking, Working};
enum EngineStatus {NotReady, Ready};

#endif
```

FUELPUMP.H

fuelpump.h

```
#ifndef FUEL_PUMP_H
#define FUEL_PUMP_H
#include "aircraftutils.h"

class FuelPump {
public:
    FuelPump(PartStatus _status = Working, short engine_number = 0);
    ~FuelPump();
    PartStatus isWorkingProperly();
    void setStatus(PartStatus _status);
    void registerWithEngineNumber(short _engine_number);
    short getRegisteredEngineNumber();
private:
    PartStatus status;
    short registered_engine_number;
};
#endif
```

oilpump.h*oilpump.h*

```

#ifndef OIL_PUMP_H
#define OIL_PUMP_H
#include "aircraftutils.h"

class OilPump {
public:
    OilPump(PartStatus _status = Working, short engine_number = 0);
    ~OilPump();
    PartStatus isWorkingProperly();
    void setStatus(PartStatus _status);
    void registerWithEngineNumber(short _engine_number);
    short getRegisteredEngineNumber();
private:
    PartStatus status;
    short registered_engine_number;
};
#endif

```

TEMPERATURESENSOR.H*temperaturesensor.h*

```

#ifndef TEMP_SENSOR_H
#define TEMP_SENSOR_H
#include "aircraftutils.h"

class TemperatureSensor {
public:
    TemperatureSensor(PartStatus _status = Working, short engine_number = 0);
    ~TemperatureSensor();
    PartStatus isWorkingProperly();
    void setStatus(PartStatus _status);
    void registerWithEngineNumber(short _engine_number);
    short getRegisteredEngineNumber();
private:
    PartStatus status;
    short registered_engine_number;
};
#endif

```

OXYGENSENSOR.H*oxygensensor.h*

```

#ifndef OXYGEN_SENSOR_H
#define OXYGEN_SENSOR_H
#include "aircraftutils.h"

class OxygenSensor {
public:
    OxygenSensor(PartStatus _status = Working, short engine_number = 0);
    ~OxygenSensor();
    PartStatus isWorkingProperly();
    void setStatus(PartStatus _status);
    void registerWithEngineNumber(short _engine_number);
    short getRegisteredEngineNumber();
};

```



```

    private:
        PartStatus status;
        short registered_engine_number;
};
#endif

```

COMPRESSOR.H*compressor.h*

```

#ifndef COMPRESSOR_H
#define COMPRESSOR_H
#include "aircraftutils.h"

class Compressor {
public:
    Compressor(PartStatus _status = Working, short engine_number = 0);
    ~Compressor();
    PartStatus isWorkingProperly();
    void setStatus(PartStatus _status);
    void registerWithEngineNumber(short _engine_number);
    short getRegisteredEngineNumber();
private:
    PartStatus status;
    short registered_engine_number;
};
#endif

```

ENGINE.H*engine.h*

```

#ifndef ENGINE_H
#define ENGINE_H
#include "aircraftutils.h"
#include "fuelpump.h"
#include "oxygensensor.h"
#include "oilpump.h"
#include "compressor.h"
#include "temperaturesensor.h"

class Engine{
public:
    Engine();
    ~Engine();
    EngineStatus checkEngineStatus();
    void startEngine();
    void stopEngine();
    void setFuelPumpStatus(PartStatus _status);
    void setOilPumpStatus(PartStatus _status);
    void setTemperatureSensorStatus(PartStatus _status);
    void setOxygenSensorStatus(PartStatus _status);
    void setCompressorStatus(PartStatus _status);

private:
    static short    engine_count;
    FuelPump        its_fuel_pump;
    OilPump         its_oil_pump;
    TemperatureSensor its_temperature_sensor;
};

```

```

    OxygenSensor    its_oxygen_sensor;
    Compressor      its_compressor;
    short           its_engine_number;
    EngineStatus    its_status;
    bool            engine_is_running;
};
#endif

```

fuelpump.cpp

fuelpump.cpp

```

#include "aircraftutils.h"
#include "fuelpump.h"
#include <iostream>
using namespace std;

FuelPump::FuelPump(PartStatus _status, short engine_number):status(_status),
    registered_engine_number(engine_number) {
    switch(status) {
        case NotWorking: cout<<"Fuel pump " <<registered_engine_number
            <<" malfunction"<<endl;
            break;
        case Working : cout<<"Fuel pump " <<registered_engine_number<<" working"<<endl;
            break;
        default : ;
    }
}

FuelPump::~FuelPump() {
    cout<<"Fuel pump " <<registered_engine_number<<" destroyed"<<endl;
}

PartStatus FuelPump::isWorkingProperly() { return status;}

void FuelPump::setStatus(PartStatus _status) { status = _status;}

void FuelPump::registerWithEngineNumber(short _engine_number) {
    registered_engine_number = _engine_number;
}

short FuelPump::getRegisteredEngineNumber() {return registered_engine_number;}

```

oilpump.cpp

oilpump.cpp

```

#include "aircraftutils.h"
#include "oilpump.h"
#include <iostream>
using namespace std;

OilPump::OilPump(PartStatus _status, short engine_number):status(_status),
    registered_engine_number(engine_number) {
    switch(status) {
        case NotWorking: cout<<"Oil pump " <<registered_engine_number<<" malfunction"<<endl;
            break;
        case Working : cout<<"Oil pump " <<registered_engine_number<<" working"<<endl;
            break;
    }
}

```

```

        default      : ;
    }
}

OilPump::~OilPump() {
    cout<<"Oil pump "<<registered_engine_number<<" destroyed"<<endl;
}

PartStatus OilPump::isWorkingProperly(){ return status;}

void OilPump::setStatus(PartStatus _status){ status = _status;}

void OilPump::registerWithEngineNumber(short _engine_number){
    registered_engine_number = _engine_number;
}

short OilPump::getRegisteredEngineNumber(){return registered_engine_number;}

```

TEMPERATURESENSOR.CPP*temperaturesensor.cpp*

```

#include "aircraftutils.h"
#include "temperaturesensor.h"
#include <iostream>
using namespace std;

TemperatureSensor::TemperatureSensor(PartStatus _status, short engine_number)
    :status(_status), registered_engine_number(engine_number){
    switch(status){
        case NotWorking: cout<<"Temperature sensor "<<registered_engine_number
            <<" malfunction"<<endl;
            break;
        case Working : cout<<"Temperature sensor "<<registered_engine_number
            <<" working"<<endl;
            break;
        default      : ;
    }
}

TemperatureSensor::~TemperatureSensor(){
    cout<<"Temperature sensor "<<registered_engine_number<<" destroyed"<<endl;
}

PartStatus TemperatureSensor::isWorkingProperly(){ return status;}

void TemperatureSensor::setStatus(PartStatus _status){ status = _status;}

void TemperatureSensor::registerWithEngineNumber(short _engine_number){
    registered_engine_number = _engine_number;
}

short TemperatureSensor::getRegisteredEngineNumber(){return registered_engine_number;}

```

OXYGENSENSOR.CPP*oxygensensor.cpp*

```

#include "aircraftutils.h"

```

```

#include "oxygensensor.h"
#include <iostream>
using namespace std;

OxygenSensor::OxygenSensor(PartStatus _status, short engine_number)
    :status(_status), registered_engine_number(engine_number){
    switch(status){
        case NotWorking: cout<<"Oxygen sensor "<<registered_engine_number
            <<" malfunction"<<endl;
            break;
        case Working    : cout<<"Oxygen sensor "<<registered_engine_number
            <<" working"<<endl;
            break;
        default         : ;
    }
}

OxygenSensor::~OxygenSensor(){
    cout<<"Oxygen sensor "<<registered_engine_number<<" destroyed"<<endl;
}

PartStatus OxygenSensor::isWorkingProperly(){ return status;}

void OxygenSensor::setStatus(PartStatus _status){ status = _status;}

void OxygenSensor::registerWithEngineNumber(short _engine_number){
    registered_engine_number = _engine_number;
}

short OxygenSensor::getRegisteredEngineNumber(){return registered_engine_number;}

```

COMPRESSOR.CPP*compressor.cpp*

```

#include "aircraftutils.h"
#include "compressor.h"
#include <iostream>
using namespace std;

Compressor::Compressor(PartStatus _status, short engine_number)
    :status(_status), registered_engine_number(engine_number){
    switch(status){
        case NotWorking: cout<<"Compressor "<<registered_engine_number
            <<" malfunction"<<endl;
            break;
        case Working    : cout<<"Compressor "<<registered_engine_number
            <<" working"<<endl;
            break;
        default         : ;
    }
}

Compressor::~Compressor(){
    cout<<"Compressor "<<registered_engine_number<<" destroyed"<<endl;
}

PartStatus Compressor::isWorkingProperly(){ return status;}

```

```

void Compressor::setStatus(PartStatus _status){ status = _status;}

void Compressor::registerWithEngineNumber(short _engine_number){
    registered_engine_number = _engine_number;
}

short Compressor::getRegisteredEngineNumber(){return registered_engine_number;}

```

ENGINE.CPP*engine.cpp*

```

#include "aircraftutils.h"
#include "fuelpump.h"
#include "oxygensensor.h"
#include "oilpump.h"
#include "compressor.h"
#include "temperaturesensor.h"
#include "engine.h"
#include <iostream>
using namespace std;

short Engine::engine_count = 1;

Engine::Engine():its_engine_number(engine_count++), engine_is_running(false),
    its_fuel_pump(Working, engine_count),
    its_oil_pump(Working, engine_count),
    its_temperature_sensor(Working, engine_count),
    its_oxygen_sensor(Working, engine_count),
    its_compressor(Working, engine_count) {}

Engine::~~Engine() {
    engine_count--;
}

EngineStatus Engine::checkEngineStatus() {
    if(its_fuel_pump.isWorkingProperly()
        && its_oil_pump.isWorkingProperly()
        && its_temperature_sensor.isWorkingProperly()
        && its_oxygen_sensor.isWorkingProperly()
        && its_compressor.isWorkingProperly()){
        its_status = Ready;
        cout<<"All engine number "<<its_engine_number
            <<" components working properly."<<endl;
    }
    else{
        its_status = NotReady;
        cout<<"Engine number "<<its_engine_number<<" malfunction!"<<endl;
        stopEngine();
    }

    return its_status;
}

void Engine::startEngine(){
    if(!engine_is_running && checkEngineStatus()){

```

```

        cout<<"Engine number "<<its_engine_number<<" started."<<endl;
        engine_is_running = true;
    }
    else cout<<"Engine number "<<its_engine_number<<" cannot start."<<endl;
}

void Engine::stopEngine() {
    if(engine_is_running) {
        engine_is_running = false;
        cout<<"Engine number "<<its_engine_number<<" stopped."<<endl;
    }
    else cout<<"Engine "<<its_engine_number<<" not running!"<<endl;
}

void Engine::setFuelPumpStatus(PartStatus _status) {
    its_fuel_pump.setStatus(_status);
}

void Engine::setOilPumpStatus(PartStatus _status) {
    its_oil_pump.setStatus(_status);
}

void Engine::setTemperatureSensorStatus(PartStatus _status) {
    its_temperature_sensor.setStatus(_status);
}

void Engine::setOxygenSensorStatus(PartStatus _status) {
    its_oxygen_sensor.setStatus(_status);
}

void Engine::setCompressorStatus(PartStatus _status) {
    its_compressor.setStatus(_status);
}

```

main.cpp*main.cpp*

```

#include <iostream>
using namespace std;

#include "engine.h"

int main() {
    Engine e1,e2;

    e1.checkEngineStatus();
    e2.checkEngineStatus();
    e1.stopEngine();
    e1.startEngine();
    e2.startEngine();
    e1.setCompressorStatus(NotWorking);
    e1.checkEngineStatus();

    return 0;
}

```

SUMMARY

Class behavior can be implemented by building upon the behavior of other classes. A class built from other user-defined classes is called an aggregate class or aggregation. There are two forms of aggregation: simple and composite. In simple aggregation the whole object does not control the lifetime of its part objects. This means that part objects could play a role in more than one aggregate relationship. In composite aggregation the whole object controls the lifetime of its part objects. This means it controls their creation and destruction. Composite aggregates can be formed from simple part objects, part objects allocated dynamically, or from a combination of both. Composite aggregates control the lifetimes of their part objects; simple aggregates do not.

The UML class diagram can be used to illustrate aggregate relationships. A line between classes denotes an association. A diamond is placed at one end of the association line denoting aggregation. A hollow diamond indicates simple aggregation; a solid diamond indicates composite aggregation.

The UML sequence diagram is used to illustrate message passing between objects. Whereas the class diagram shows the static relationship between classes in an object-oriented application, a sequence diagram shows the dynamic interaction between objects in the application.

Skill Building Exercises

1. **Procure UML Design Tool:** Procure a UML design tool. The features you're looking for in addition to diagramming are reverse engineering and code generation. Many good shareware tools exist but it may be a good idea to download a commercial tool with a limited demo license.
2. **Obtain UML Reference:** Procure a good UML reference and/or tutorial. Many good UML tutorials exist on the Internet so I recommend starting your search there. UML has evolved since it was first introduced so it is a good idea to get a recent reference.
3. **Simple Aggregate Class:** Write a program that uses three simple classes named Whole, PartA, and PartB. Tailor the constructors and destructors of PartA and PartB to print out messages specific to those class types indicating their construction and destruction. In addition to a constructor and destructor provide one additional public interface function for each part class named showBehavior(). Tailor the behavior of the showBehavior() function to print out a different message to the screen for each part class. Create a simple aggregate version of Whole class where the Whole class has two private attributes; one of type pointer to PartA and the other of type pointer to PartB. Write a main() function to test your aggregate class. Create instances of PartA and PartB and test their interface functions. Then, using the code of example 12.12 as a guide, create an object of type Whole using the addresses of the part objects. Hint: The Whole class constructor should take two parameters: one that is a pointer to a PartA object and the other that is a pointer to a PartB object.
4. **Reverse Engineer:** Use the UML design tool you procured in exercise 1 above to reverse engineer the code you wrote for Exercise 3.
5. **Composite Aggregate Class:** Revise skill building exercise 3 to make the Whole class a composite aggregate. Leave the data members as pointer types but create each of the part objects in the Whole class constructor. Hint: Use the code in examples 12.6 and 12.7 as a guide.
6. **Composite Aggregate Class:** Revise skill building exercise 3 again to make the Whole class a composite aggregate without using pointers.
7. **Simple/Composite Class:** Revise skill building exercise 3 again to make the Whole class a combination simple and composite aggregation.

8. **Reverse Engineer:** Use your UML tool to reverse engineer the code resulting from skill building exercise 7. How is it different from the diagram generated in skill building exercise 4?
9. **Design Simulation:** Use your UML tool to design a simple simulation like the aircraft engine simulator from scratch. Perhaps a tank comprised of an engine and a gun. Both the engine and the gun are themselves comprised of several parts. Think of the interface the tank would need and how a message sent to a tank object would call a message on its engine object or gun object. When you think your design is complete generate the source code and study the results. Compare them to the aircraft engine simulation code. Evaluate the performance of your UML tool with regards to how close it comes to generating good header files and relatively complete class function implementation stubs. (*see chapter 20*)
10. **Implement Simulation:** Write the code required to get the code generated in the previous exercise working.

SUGGESTED PROJECTS

1. **Aircraft Simulation:** Expand on the aircraft engine simulation project. Add a class named EngineComputer that allows you to periodically monitor engine status and set the status of engine components. The EngineComputer might display a text-based user interface that allows the user to interactively monitor engine status. The EngineComputer class should be able to control as many Engine objects as required.
2. **Aircraft Simulation:** Expand on the aircraft engine simulation project. Create a class named Aircraft that contains one EngineComputer and four Engine objects. Make Aircraft a composite aggregation. Write a small program to test the functionality of your Aircraft class.
3. **Computer Simulator:** Rewrite the computer simulator presented in chapter 8, suggested project 4, as an aggregate class. Implement the following components as separate classes: Memory, Processor, ComputerSystem. The Memory class will encapsulate the memory array and provide a set of functions to read from and write to specific memory locations. The Processor will encapsulate the accumulator and implement the instructions that manipulate the accumulator. The ComputerSystem will be comprised of a Memory and a Processor and contain any code necessary to get the two objects to work together. The ComputerSystem class is also responsible for the presentation of a user interface.
4. **Research:** Do some research on the Hubble space telescope. Design a simple application that lets you control and aim the telescope and check its operational status. There should be two primary composite classes: Telescope and GroundStation. A GroundStation object should be able to control as many Telescope objects as necessary. Identify several key component Telescope parts and several GroundStation parts and implement them as classes. The Telescope should be a composite aggregation and the GroundStation perhaps a combination simple/composite aggregate. Layout the design of your Hubble space telescope control system using your UML modeling tool.
5. **Generate Code:** Generate the source files for the Hubble space telescope control system and implement the remaining code. Test and run your program.

SELF TEST QUESTIONS

1. A class built from other class types is referred to as an _____.
2. List the two types of aggregation.
3. Discuss each of the types of aggregation you listed above in terms of what role part objects play in each.

4. In this type of aggregation, part objects belong solely to the whole or containing class. Name the type of aggregation.
5. In this type of aggregation, part object lifetimes are not controlled by the whole or containing class. Name the type of aggregation.
6. What does the line drawn between classes in a UML class diagram denote?
7. What type of aggregation does a solid diamond indicate when attached to one end of an association line?
8. What type of aggregation does a hollow diamond indicate when attached to one end of an association line?
9. In a UML class diagram, the aggregation diamond is drawn closest to which class, the whole or the part?
10. What is the purpose of a UML sequence diagram?

REFERENCES

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994. ISBN: 0-8053-5340-2

Robert C. Martin. *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice Hall, Englewood Cliffs, New Jersey, 1994. ISBN: 0-13-203837-4

Grady Booch, et. al. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, 1998. ISBN: 0-201-57168-4

NOTES

CHAPTER 13



Colorful Port

EXTENDING CLASS FUNCTIONALITY THROUGH INHERITANCE

LEARNING OBJECTIVES

- STATE THE PURPOSE AND USE OF INHERITANCE IN C++ CLASS DESIGN
- EXPLAIN HOW TO APPLY THE THREE ACCESS SPECIFIERS, PUBLIC, PROTECTED, AND PRIVATE
- EXPLAIN HOW TO HIDE BASE CLASS FUNCTIONS WITH DERIVED CLASS FUNCTIONS
- EXPLAIN HOW TO CALL A BASE CLASS CONSTRUCTOR FROM A DERIVED CLASS INITIALIZER LIST
- EXPLAIN THE USE OF THE VIRTUAL KEY WORD AS IT RELATES TO DESTRUCTORS AND CLASS MEMBER FUNCTIONS
- EXPLAIN HOW TO OVERRIDE VIRTUAL BASE CLASS FUNCTIONS
- EXPLAIN HOW TO IMPLEMENT PURE VIRTUAL FUNCTIONS
- EXPLAIN HOW TO DECLARE AND USE ABSTRACT BASE CLASSES
- EXPLAIN HOW TO SUBSTITUTE DERIVED CLASS OBJECTS WHERE BASE CLASS OBJECTS ARE SPECIFIED
- EXPLAIN HOW TO IMPLEMENT MULTIPLE INHERITANCE
- STATE THE PURPOSE AND USE OF A VIRTUAL BASE CLASS
- EXPLAIN HOW TO SAFELY USE INHERITANCE IN YOUR APPLICATION DESIGN
- EXPLAIN HOW TO EXTEND THE UML CLASS DIAGRAM TO ILLUSTRATE CLASS INHERITANCE HIERARCHIES
- DEMONSTRATE YOUR ABILITY TO EXPRESS INHERITANCE WITH A UML CLASS DIAGRAM
- DEMONSTRATE YOUR ABILITY TO UTILIZE INHERITANCE IN THE DESIGN OF COMPLEX C++ PROGRAMMING PROJECTS

INTRODUCTION

In this chapter you will learn how to create new class types that derive some or all of their behavior by inheriting that behavior from one or more pre-existing class types. A class that inherits the functionality of another class is referred to as a subclass or derived class; the class whose behavior is inherited is referred to as the superclass or base class. Inheritance is a powerful tool that, when combined with compositional design, opens seemingly endless design possibilities.

We have a lot to talk about in this chapter. The three access specifiers `public`, `protected`, and `private` will now be used to specify how base class members are to be accessed from derived class objects. You use the access specifiers to specify `public`, `protected`, or `private` inheritance.

You will learn how to call a base class constructor from a derived class initializer list, how to apply the `virtual` keyword to functions, and how to implement virtual functions that can be overridden in derived classes. You will learn how to create abstract base classes that contain pure virtual functions. Once you learn how to create abstract base classes I will show you how to inherit and implement these abstract base class interfaces in derived classes and use pointers to abstract base classes to call functions on derived class objects. You will learn the difference between function overloading and function overriding. And just when you think you have had enough I will show you how to inherit from more than one class!

In support of the material presented here I will show you how to extend the UML class diagram to express inheritance relationships.

PURPOSE AND USE OF INHERITANCE

Inheritance allows you to adopt or extend the behavior of an existing class or set of classes. Creating new classes via inheritance offers many benefits. First, and perhaps primarily, any class that inherits the behavior of another class is said to implement an “is a...” relationship. For instance, if class B inherits behavior from class A then a class B object is also a class A object.

Another benefit of inheritance is code reuse. When designing with inheritance, base classes should declare or define behavior common to all subclasses. If you find yourself repeating code in a set of subclasses that share a common base class you should migrate that code up the inheritance hierarchy and put it in the base class in which it belongs.

The most powerful benefit you gain from using inheritance is the ability to define abstract base classes and defer implementation of their pure virtual functions to their subclasses. Designing with abstract base classes allows you to define a stable application architecture in terms of class interfaces and the behavior they declare. Given a well-designed application architecture, new application features can be added by extending the architecture through inheritance rather than modifying the architecture itself. This is the idea behind an advanced object-oriented design principle known as the open-closed principle (OCP). (*open for extension, closed for modification*)

Understanding how to implement and use inheritance prepares you for learning two other advanced object-oriented design principles: Liskov substitution principle (LSP) and dependency inversion principle (DIP). Chapter 19 formally presents all three of these design principles.

EXPRESSING INHERITANCE WITH A UML CLASS DIAGRAM

The UML class diagram can be extended to show class generalization. An association line tipped with an open arrowhead is drawn from the subclass to the base class. Figure 13-1 shows a class diagram for two classes named `BaseClass` and `DerivedClassOne`. These classes will be used in the next several sections to demonstrate some of the features associated with inheritance.

`BaseClass` is referred to as the base class and `DerivedClassOne` is referred to as the derived class or subclass. Since `DerivedClassOne` directly inherits from `BaseClass`, `BaseClass` can also be referred to as a direct base class.

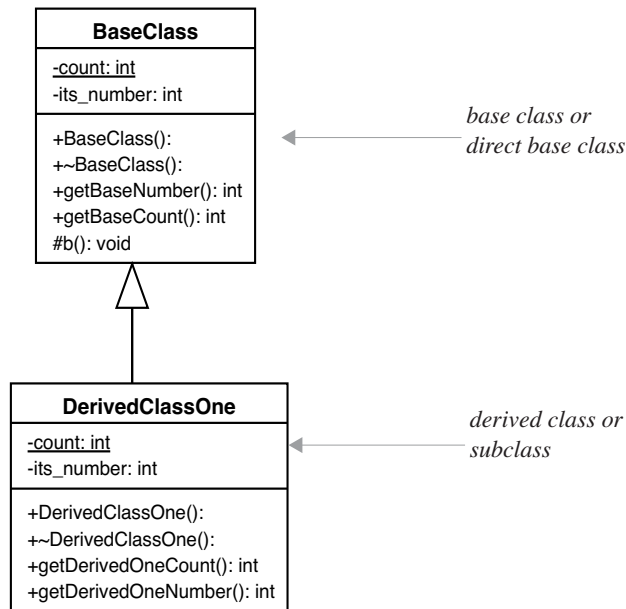


Figure 13-1: UML Class Diagram Showing Generalization

BaseClass contains two private attributes. One is a static, class-wide variable named count, and the other is an integer variable named its_count. BaseClass contains four public functions and one protected function. You will soon see how protected members are treated in an inheritance relationship. DerivedClassOne has its own set of private attributes with the same names as the ones found in BaseClass. DerivedClassOne has four public interface functions that provide basic functionality.

IMPLEMENTING BASECLASS AND DERIVEDCLASSONE

Examples 13.1 through 13.4 give the source code for BaseClass and DerivedClassOne contained in four files named baseclass.h, derivedclassone.h, baseclass.cpp, and derivedclassone.cpp.

```

1  #ifndef BASE_CLASS_H
2  #define BASE_CLASS_H
3
4  class BaseClass{
5  public:
6      BaseClass();
7      virtual ~BaseClass();
8      int getBaseNumber();
9      int getBaseCount();
10
11  protected:
12      void b();
13
14  private:
15      static int count;
16      int its_number;
17  };
18  #endif

```

13.1 baseclass.h

The behavior of BaseClass is declared by the limited set of public interface functions. All you can do with a BaseClass object is create it, destroy it, get the base number, and get the base count. The protected BaseClass::b()

```

1  #ifndef DERIVED_CLASS_H 13.2 derivedclassone.h
2  #define DERIVED_CLASS_H
3  #include "baseclass.h"
4
5  class DerivedClassOne : public BaseClass{
6  public:
7      DerivedClassOne();
8      virtual ~DerivedClassOne();
9      int getDerivedOneCount();
10     int getDerivedOneNumber();
11
12     private:
13         static int count;
14         int its_number;
15 };
16 #endif

```

```

1  #include "baseclass.h" 13.3 baseclass.cpp
2  #include <iostream>
3  using namespace std;
4
5  int BaseClass::count = 0;
6
7  BaseClass::BaseClass():its_number(++count){
8      cout<<"BaseClass object number "<<its_number<<" created."<<endl;
9      cout<<"There are "<<count<<" BaseClass objects."<<endl;
10 }
11
12 BaseClass::~BaseClass(){
13     cout<<"BaseClass object number "<<its_number<<" destroyed."<<endl;
14     cout<<"There are "<<--count<<" BaseClass objects remaining."<<endl;
15 }
16
17 int BaseClass::getBaseNumber(){ return its_number;}
18
19 int BaseClass::getBaseCount(){return count;}
20
21 void BaseClass::b(){
22     cout<<"BaseClass protected function called!"<<endl;
23 }

```

method can only be called from within the class itself or from within the code of a derived class. The `BaseClass::b()` method simply prints a message to the screen.

It should be noted that the private attributes of `BaseClass` remain private to `BaseClass`. They are not available for horizontal access or for access by derived classes. The protected function `BaseClass::b()` is also closed to horizontal access but is available for inheritance by derived classes.

The keyword `virtual` is used in front of the destructor. This is done to ensure derived class destructors get called when necessary.

Refer to the declaration for `DerivedClassOne` shown in example 13.2 above. The class declaration includes a colon followed by the keyword `public` followed by the name of the class to be used as a base class. In this case the name of the base class is `BaseClass`. The rest of the `DerivedClassOne` class declaration is what you have seen in previous chapters. The keyword `virtual` is applied to the `DerivedClassOne` destructor although according to the C++ standard doing so is redundant.

Study examples 13.3 through 13.4 to get a feel for what type of behavior these two simple classes will exhibit. When objects of each type are created the constructors will print simple messages to the screen. The same holds true when objects are destroyed. The static data members named `count` in each class will be used to keep track of how

```

1  #include "derivedclassone.h"
2  #include "baseclass.h"
3  #include <iostream>
4  using namespace std;
5
6  int DerivedClassOne::count = 0;
7
8  DerivedClassOne::DerivedClassOne():its_number(++count){
9      cout<<"DerivedClassOne object number "<<its_number<<" created."<<endl;
10     cout<<"There are "<<count<<" DerivedClassOne objects."<<endl;
11 }
12
13 DerivedClassOne::~DerivedClassOne(){
14     cout<<"DerivedClassOne object number "<<its_number<<" destroyed."<<endl;
15     cout<<"There are "<<--count<<" DerivedClassOne objects remaining."<<endl;
16 }
17
18 int DerivedClassOne::getDerivedOneCount(){return count;}
19
20 int DerivedClassOne::getDerivedOneNumber(){return its_number;}

```

13.4 derivedclassone.cpp

many objects of each type exist. Each class has several accessor functions to get the values of both the static and local data members. The BaseClass::b() function just prints a message to the screen saying that a protected function was called.

Although these two classes do not do much, the simple messages contained in the constructors and destructors will teach us a lot about inheritance. Let us take a look at how these classes would be used in a main() function and examine the results obtained from running such a program. Example 13.5 gives the code showing BaseClass and DerivedClassOne objects being created used in a main() function. Figure 13-2 shows the output when the program is run.

Compare the code shown in example 13.5 with the output shown in figure 13-2. On line 8 in example 13.5 a BaseClass object named b1 is created. This causes the constructor message to be printed to the console. On the next several lines the two BaseClass functions named getBaseCount() and getBaseNumber() are called on the b1 object. A DerivedClassOne object is created on line 15. Because it inherits the public functions of BaseClass those functions can be called through the DerivedClassOne object which is done on lines 17 and 19. On line 24 a BaseClass pointer named base_ptr is declared and assigned the address of a dynamically allocated DerivedClassOne object. Because the pointer is of a BaseClass type, only the functions declared in BaseClass can be called. On line 33 a DerivedClassOne pointer named derived_ptr is declared and initialized to the address of a dynamically allocated DerivedClassOne object. Since the pointer is of a derived class type it can be used to call both base class and derived class functions. The pointers are deleted on lines 41 and 42 before the program exits. The statically allocated objects will be deleted when the program terminates.

Quick Review

A class type can exhibit the behavior of another class type through the mechanism of inheritance. The class whose behavior is inherited is called the base class; the class inheriting the behavior is called the derived class. A public base class function can be called via a derived class object. A base class pointer can be assigned the addresses of a derived class object, however, since the pointer is of a base class type, only the public functions declared in the base class can be called via the pointer. (*True for now until you learn how to override base class functions in a later section.*) A derived class pointer holding a derived class object address can call both base class and derived class public functions. The previous section presented a quick overview of inheritance but left out a lot of cool stuff. The following sections explore the topic in greater detail.

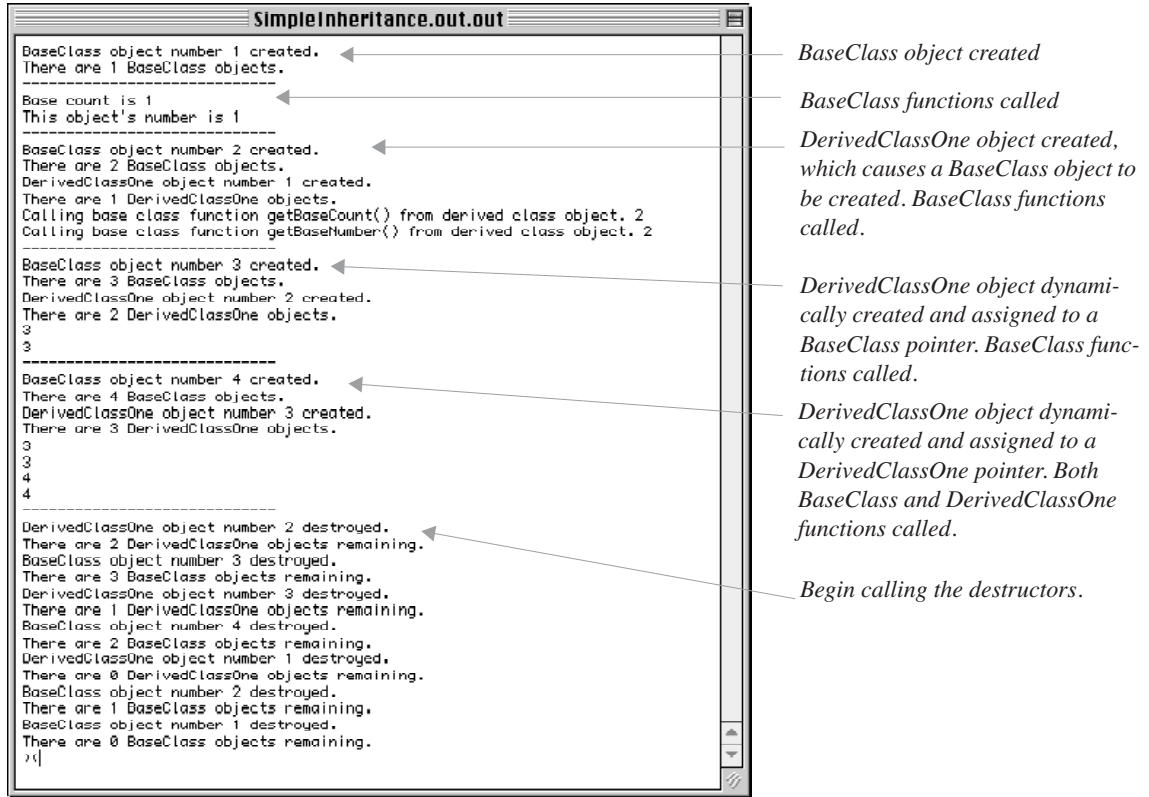


Figure 13-2: Results of Running Example 13.5

ACCESS SPECIFIERS AND VERTICAL ACCESS

The three access specifiers `public`, `protected`, and `private` are used to specify what type of access base class members will have from the derived class when those members are inherited.

Figure 13-3 shows how each access specifier affects the inheritance mechanism.

Inheritance	base class	derived class
<i>public</i>	public: ● protected: ● private:	public: ● protected: ● private:
<i>protected</i>	public: ● protected: ● private:	public: protected: ● private:
<i>private</i>	public: ● protected: ● private:	public: protected: private: ●

Figure 13-3: Effects of Using Different Inheritance Specifiers

```

1  #include "baseclass.h"
2  #include "derivedclassone.h"
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {   /* create a base class object and call some functions */
8      BaseClass b1;
9      cout<<"-----"<<endl;
10     cout<<"Base count is "<<b1.getBaseCount()<<endl;
11     cout<<"This object's number is "<<b1.getBaseNumber()<<endl;
12     cout<<"-----"<<endl;
13
14     /* create a derived class object and call some base class functions */
15     DerivedClassOne d1;
16     cout<<"Calling base class function getBaseCount() "
17         <<"from derived class object. "<< d1.getBaseCount()<<endl;
18     cout<<"Calling base class function getBaseNumber() "
19         <<"from derived class object. "<<d1.getBaseNumber()<<endl;
20     cout<<"-----"<<endl;
21
22     /* create a base class pointer and assign
23        derived class object address */
24     BaseClass *base_ptr = new DerivedClassOne;
25
26     /* call some base class functions */
27     cout<<base_ptr->getBaseCount()<<endl;
28     cout<<base_ptr->getBaseNumber()<<endl;
29     cout<<"-----"<<endl;
30
31     /* now create derived class pointer and call some
32        derived class functions and some base class functions */
33     DerivedClassOne *derived_ptr = new DerivedClassOne;
34     cout<<derived_ptr->getDerivedOneCount()<<endl;
35     cout<<derived_ptr->getDerivedOneNumber()<<endl;
36     cout<<derived_ptr->getBaseCount()<<endl;
37     cout<<derived_ptr->getBaseNumber()<<endl;
38     cout<<"-----"<<endl;
39
40     /* destroy the pointers before exiting program */
41     delete base_ptr;
42     delete derived_ptr;
43     return 0;
44 }

```

13.5 main.cpp

Public Inheritance

Use the public access specifier to specify public inheritance. Referring to figure 13-3 above and figure 13-4 below, if the public access specifier is used in the derived class, public data members and functions in the base class are inherited by the derived class and remain public. Protected data members and functions are also inherited by the derived class and remain protected.

Public inheritance is used to implement “is a” relationships between base and derived classes. The use of public inheritance ensures base class interface functions will continue to be inherited if the inheritance hierarchy is extended.

Protected Inheritance

Use the protected access specifier to specify protected inheritance. If protected access is specified, the public and protected data members and functions inherited by the derived class become protected.

Private Inheritance

Use the private access specifier to specify private inheritance. If private access is specified then the public and protected data members and functions of the base class will become private to the derived class. This effectively prevents any further inheritance of base class data members and functions since private members and functions are not inherited.

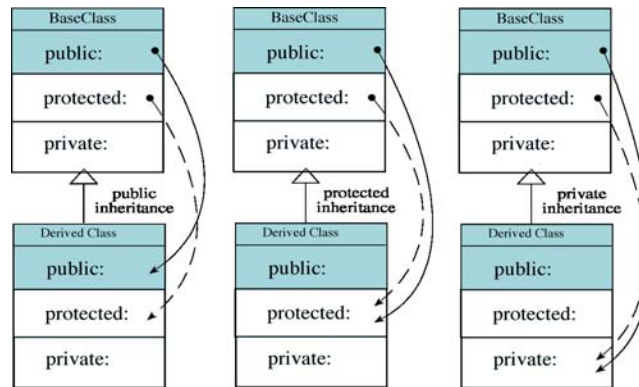


Figure 13-4: Public, Protected, & Private Inheritance

Figure 13-5 shows the effects of public inheritance from a horizontal access perspective.

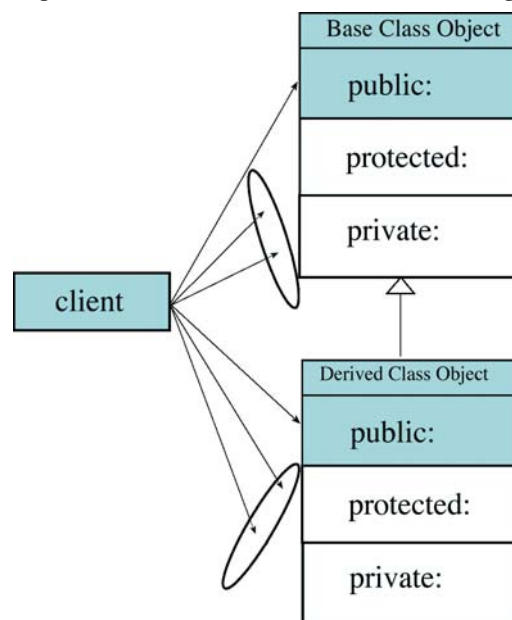


Figure 13-5: Public Inheritance from a Horizontal Access Perspective

The client cannot horizontally access the protected or private members of either the base class or derived class objects.

Quick Review

The access specifiers `public`, `protected`, and `private` are used to control vertical access between base and derived classes and to specify the type of access base class members will have from the derived class. Public inheritance is achieved with the `public` access specifier. Public inheritance is used to implement an “is a” relationship between a base and derived class. The `protected` access specifier is used to specify protected inheritance. Protected inheritance is seldom used in practice. Use the `private` access specifier to specify private inheritance. Private inheritance is used to prevent further implementation of “is a” relationships.

CALLING BASE CLASS CONSTRUCTORS

When a derived class object is created it must call its base class constructor to ensure its base class object is properly instantiated. Base class constructors are called in the constructor initializer list. Let us take a look at some example code to see how this is done. This example will also illustrate how code can be reused and extended in an object-oriented design. Figure 13-6 is a class diagram showing the class `Person`, originally presented in chapter 11, being extended by a derived class named `Student`.

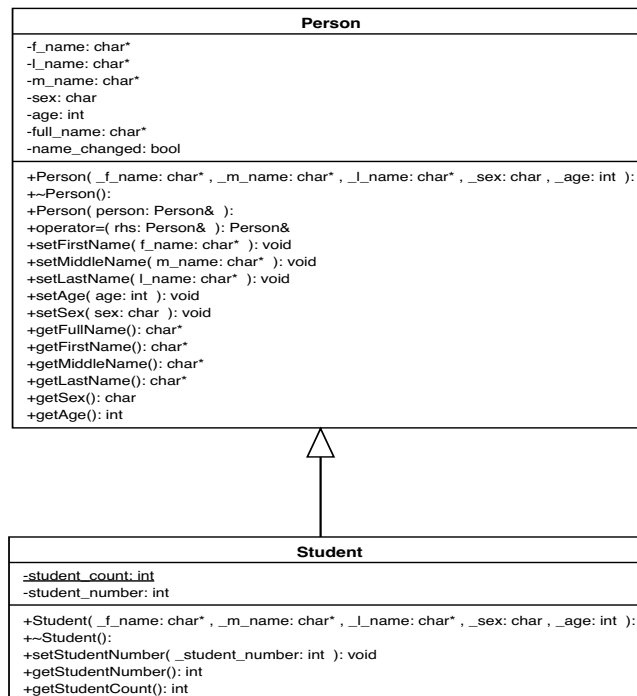


Figure 13-6: Person/Student Class Diagram

By extending `Person`, a `Student` will have all the behavior of `Person`, plus any additional behavior specific to a `Student`. Examination of figure 13-6 reveals the addition of a `student_count` static variable and an instance variable named `student_number`. Accessor and mutator functions were added to manipulate these variables. Other than that, a `Student` is a `Person`. (of course!)

The source code for `Person` is given in its entirety in chapter 11 so I will not repeat it here with the exception of the `person.h` file. This is really all we need in that a programmer tasked with writing the `Student` class may only have access to the `Person` class header file. That would be all they need to determine what constructors and other public interface functions are available for their use.

Examples 13.6 through 13.9 give the source code for `person.h`, `student.h`, `student.cpp`, and a `main.cpp` file showing these classes in action. Figure 13-7 shows the results of running example 13.9.

Refer to the `student.cpp` file shown in example 13.8. The `Student` class constructor definition begins on line 8 and continues through line 11. The `Student` constructor declares parameters that match those of the `Person` class. This means that for a `Student` object to be created it needs a first name, middle name, last name, sex, and age. However, these attributes are not found in the `Student` class. The `Student` constructor must use its constructor parameters as arguments to the `Person` constructor. This is shown on line 9 of example 13.8. The colon starts the `Student` constructor initializer list and the first thing that appears to the right of the colon is the call to the `Person` base class constructor. The rest of the `Student` constructor goes on to initialize the `student_number` using the static `student_count` variable.

```

1 #ifndef __Person_H
2 #define __Person_H
3
4
5 class Person{
6     public:
7         Person(char* _f_name = "John", char* _m_name = "M",
8               char* _l_name = "Doe", char _sex = 'M', int _age = 18);
9         virtual ~Person();
10        Person(Person& person);
11        Person& operator=(Person& rhs);
12        void setFirstName(char* f_name);
13        void setMiddleName(char* m_name);
14        void setLastName(char* l_name);
15        void setAge(int age);
16        void setSex(char sex);
17        char* getFullName();
18        char* getFirstName();
19        char* getMiddleName();
20        char* getLastName();
21        char getSex();
22        int getAge();
23
24    private:
25        char* f_name;
26        char* l_name;
27        char* m_name;
28        char sex;
29        int age;
30        char* full_name;
31        bool name_changed;
32 };
33 #endif

```

13.6 person.h

```

1 #ifndef STUDENT_H
2 #define STUDENT_H
3 #include "person.h"
4
5 class Student : public Person{
6     public:
7         Student(char* _f_name = "Joe", char* _m_name = "M",
8               char* _l_name = "Student", char _sex = 'M', int _age = 18);
9         ~Student();
10        void setStudentNumber(int _student_number);
11        int getStudentNumber();
12        int getStudentCount();
13
14    private:
15        static int student_count;
16        int student_number;
17 };
18 #endif

```

13.7 student.h

```

1 #include "student.h"
2 #include "person.h"
3 #include <iostream>
4 using namespace std;
5
6 int Student::student_count = 0;
7
8 Student::Student(char* _f_name, char* _m_name, char* _l_name, char _sex, int _age)
9     :Person(_f_name, _m_name, _l_name, _sex, _age){
10        student_number = ++student_count;
11 }
12
13 Student::~Student(){
14        student_count--;
15 }
16
17 void Student::setStudentNumber(int _student_number){
18        student_number = _student_number;
19 }
20
21 int Student::getStudentNumber(){ return student_number;}
22
23 int Student::getStudentCount(){ return student_count;}

```

13.8 student.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "student.h"
4
5  int main() {
6
7      Student s1, s2("Coralie", "Sarah", "Miller");
8
9      cout<<s1.getFirstName()<<" "<<s1.getLastName()<<endl;
10     cout<<s2.getFirstName()<<" "<<s2.getLastName()<<endl;
11
12     return 0;
13 }

```

13.9 main.cpp

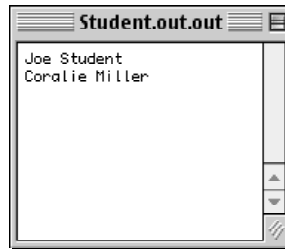


Figure 13-7: Results of Running Example 13.9

Quick Review

When a derived class must initialize base class attributes it must make an explicit base class constructor call in its initializer list. The derived class constructor can use its parameters as arguments to its base class constructor. If a default constructor exists for both the base and derived class then some or all of the constructor arguments can be omitted when objects are instantiated.

FUNCTION NAME HIDING: THIS IS NOT FUNCTION OVERRIDING!

If a derived class function has the same function signature as a base class function, the derived class function hides the base class function. Function hiding is a lot like variable scoping. If you declare a variable in a local scope it will hide a variable with the same name in an enclosing scope. Function hiding only affects inherited public and protected functions since derived classes do not have access and therefore cannot see private base class functions.

FUNCTION HIDING vs. FUNCTION OVERLOADING

Function hiding differs from function overloading in several ways. There can be several overloaded functions in the same class. One overloaded function differs from another overloaded function in the number and type of its parameters.

A hiding function has the same name and the same number and types of parameters. A hiding function will only appear in a derived class to hide a function with the same signature in its base class.

Another way to think of it is like this: When you overload a class function you are creating multiple versions of a function with the same name. Each different version of the overloaded function must be unique in its parameter list. When you call an overloaded function the compiler determines which overloaded function you're referring to by what arguments are used to make the function call.

A hiding function appears in a derived class. A hiding function in a derived class hides its matching function in its base class. There is a one-for-one mapping between a base class function and its hiding function in a derived class.

Take a look at figure 13-8.

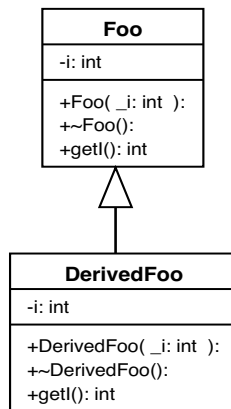


Figure 13-8: Foo and DerivedFoo Class Diagram

Figure 13-8 shows a class diagram for a class named Foo and a subclass of Foo named DerivedFoo. In addition to constructors and destructors each class implements a function named getI(). Notice that the function signature of DerivedFoo::getI() matches that of Foo::getI(). The code for both of these classes is given in examples 13.10 through 13.14 below.

```

1  #ifndef FOO_H                                13.10 foo.h
2  #define FOO_H
3
4  class Foo{
5  public:
6      Foo(int _i = 0);
7      virtual ~Foo();
8      int getI();
9  private:
10     int i;
11 };
12 #endif
  
```

```

1  #ifndef DERIVED_FOO_H                        13.11 derivedfoo.h
2  #define DERIVED_FOO_H
3  #include "foo.h"
4
5  class DerivedFoo : public Foo{
6  public:
7      DerivedFoo(int _i = 0);
8      virtual ~DerivedFoo();
9      int getI();
10 private:
11     int i;
12 };
13 #endif
  
```

```

1  #include "foo.h"                              13.12 foo.cpp
2  #include <iostream>
3  using namespace std;
4
5  Foo::Foo(int _i):i(_i){
6      cout<<"Foo object created"<<endl;
7  }
8
9  Foo::~~Foo(){
10     cout<<"Foo object destroyed!"<<endl;
11 }
12
13 int Foo::getI(){
14     cout<<"Foo getI(): ";
15     return i;
16 }
  
```

The important behavior to note here is the result of function calls on different objects. For instance, when the getI() function is invoked on a Foo object, the Foo::getI() is called. This is as you would expect. When getI() is

```

1  #include "derivedfoo.h"
2  #include <iostream>
3  using namespace std;
4
5  DerivedFoo::DerivedFoo(int _i):Foo(_i), i(_i){
6      cout<<"DerivedFoo object created."<<endl;
7  }
8
9  DerivedFoo::~~DerivedFoo(){
10     cout<<"Derived Foo object destroyed!"<<endl;
11 }
12
13 int DerivedFoo::getI(){
14     cout<<"DerivedFoo getI(): ";
15     return i;
16 }

```

13.13 derivedfoo.cpp

Base class version of getI() called.

Derived class version of getI() called

Figure 13-9: Results of Running Example 13.14

```

1  #include <iostream>
2  using namespace std;
3  #include "foo.h"
4  #include "derivedfoo.h"
5  #include "ddfoo.h"
6
7  int main(){
8      Foo f1(1);
9      cout<<"getI() called on base class object: "<<endl;
10     cout<<f1.getI()<<endl;
11     cout<<"-----"<<endl;
12
13     DerivedFoo d1(2);
14     cout<<"getI() called on derived class object: "<<endl;
15     cout<<d1.getI()<<endl;
16     cout<<"-----"<<endl;
17
18     Foo* foo_ptr = new DerivedFoo(3);
19     cout<<"getI() called via base class pointer: "<<endl;
20     cout<<foo_ptr->getI()<<endl;
21     cout<<"-----"<<endl;
22
23     DerivedFoo* derived_foo_ptr = new DerivedFoo(4);
24     cout<<"Derived getI() called via derived class pointer: "<<endl;
25     cout<<derived_foo_ptr->getI()<<endl;
26     cout<<"-----"<<endl;
27
28     delete foo_ptr;
29     delete derived_foo_ptr;
30     return 0;
31 }

```

13.14 main.cpp

invoked on a DerivedFoo object, the DerivedFoo::getI() is called. This happens because DerivedFoo::getI() hides Foo::getI(). If DerivedFoo did not declare and implement a function named getI() then Foo::getI() would have been called because it was publicly inherited from the Foo class by the DerivedFoo class.

Note the behavior of pointers. On line 18 of example 13.14 a `Foo` pointer is declared and initialized to the address of a `DerivedFoo` object. When `getI()` is called via the base class pointer it results in the base class version of the function being called, as is shown in figure 13-9.

On line 23 of example 13.14 a `DerivedFoo` pointer is declared and initialized to the address of a `DerivedFoo` object. When `getI()` is called via the derived class pointer it results in `DerivedFoo::getI()` being called.

I want to revisit the base class pointer topic for a moment. The base class, in this case `Foo`, published a set of public interface methods. These were its constructor, destructor, and `getI()`. Disregarding the constructor and destructor, if an implementation of `getI()` exists in the base class it will be called via the base class pointer even though the pointer points to a derived class object. You can, however, change this behavior by declaring functions to be virtual, using the `virtual` keyword. I will show you how to do this in the next section.

Quick Review

A function appearing in a derived class having the same function signature of a base class function hides that function. Be careful when dealing with base class pointers. When calling a function via a base class pointer the base class version of the method will be called, even though the base class pointer contains the address of a derived class object, unless the base class function is declared to be virtual.

WHAT THEN IS FUNCTION OVERRIDING?

Good question. The difference between simply hiding base class functions and overriding base class functions is the invocation of the virtual calling mechanism via a base class pointer when using virtual functions. As you saw in figure 13-9, when the `getI()` function is called via the `Foo` base class pointer, the base class version of the method is invoked even though the base class pointer contains the address of a derived class object. Therefore, `DerivedFoo::getI()` did not override `Foo::getI()`. An overriding function will be called via a base class pointer whereas a function that simply hides the name of a base class function will not. That is the difference! Understanding this difference is the key to understanding the heart and soul of object-oriented programming.

When you override a virtual base class function in a derived class and call the function via a base class pointer you achieve polymorphic behavior. Simple function name hiding does not achieve polymorphic behavior.

Now, let me show you how to achieve polymorphic behavior using virtual functions and function overriding.

CREATING VIRTUAL FUNCTIONS: THE VIRTUAL KEYWORD

PURPOSE OF VIRTUAL FUNCTIONS

The following excerpt comes straight from the ANSI C++ standard, Section 10.3 on page 168:

“Virtual functions support dynamic binding and object-oriented programming. A class that declares or inherits a virtual function is called a polymorphic class.”

DECLARING AND USING VIRTUAL FUNCTIONS

To declare a virtual function simply add the keyword `virtual` to the function declaration. Example 13.15 gives the source code for a slightly modified `Foo` class declaration originally given in example 13.10.

The only difference between this version of `Foo` and the last is the addition of the keyword `virtual` to the `getI()` function declaration. This now makes `getI()` a virtual function and the `Foo` class a polymorphic class. No other changes are required to the previous set of example files and the `main.cpp` file shown in example 13.14 can be recompiled and run to produce the output shown in figure 13-10. Notice now that when the `getI()` function is called via the base class pointer that the `Derived` version of `getI()` executes. This is an example of polymorphic behavior.

```

1  #ifndef FOO_H
2  #define FOO_H
3
4  class Foo{
5  public:
6      Foo(int _i = 0);
7      virtual ~Foo();
8      virtual int getI();
9  private:
10     int i;
11 };
12 #endif

```

13.15 foo.h

```

VirtualOverriding.out.out
Foo object created
getI() called on base class object:
Foo getI(): 1
-----
Foo object created
DerivedFoo object created.
getI() called on derived class object:
DerivedFoo getI(): 2
-----
Foo object created
DerivedFoo object created.
getI() called via base class pointer:
DerivedFoo getI(): 3
-----
Foo object created
DerivedFoo object created.
Derived getI() called via derived class pointer:
DerivedFoo getI(): 4
-----
Derived Foo object destroyed!
Foo object destroyed!
Derived Foo object destroyed!
Foo object destroyed!
Derived Foo object destroyed!
Foo object destroyed!
Derived Foo object destroyed!
Foo object destroyed!
|

```

The DerivedFoo version of getI() is now called via the base class pointer. This is polymorphic behavior.

Figure 13-10: Results of Running Example 13.14 After Modifying foo.cpp

VIRTUAL DESTRUCTORS

You have seen the keyword `virtual` added to destructors in this book before being formally discussed here. Virtual destructors are necessary in that they ensure that derived class object destructors will be called when the base class pointer is deleted. The best way to demonstrate the importance of making base class destructors virtual is to show you what happens when they are not. Example 13.16 shows the `Foo` class declaration once again modified so that the `virtual` keyword is removed from the destructor declaration. When example 13.14 is again compiled and run it produces different results as shown in figure 13-11.

```

1  #ifndef FOO_H
2  #define FOO_H
3
4  class Foo{
5  public:
6      Foo(int _i = 0);
7      ~Foo();
8      virtual int getI();
9  private:
10     int i;
11 };
12 #endif

```

13.16 foo.h

virtual keyword removed from destructor declaration

```

BadDestructor.out.out
-----
Foo object created
getI() called on base class object:
Foo getI(): 1
-----
Foo object created
DerivedFoo object created.
getI() called on derived class object:
DerivedFoo getI(): 2
-----
Foo object created
DerivedFoo object created.
getI() called via base class pointer:
DerivedFoo getI(): 3
-----
Foo object created
DerivedFoo object created.
Derived getI() called via derived class pointer:
DerivedFoo getI(): 4
-----
Foo object destroyed!
Derived Foo object destroyed!
Foo object destroyed!
Derived Foo object destroyed!
Foo object destroyed!
Foo object destroyed!

```

There's a DerivedFoo object destructor call missing. This results in a memory leak.

Figure 13-11: Results of Running Example 13.14 After Removing the virtual Keyword from Foo Class Destructor Declaration

Compare figures 13-10 and 13-11 carefully while studying example 13.14. Before the program exits the pointers are deleted. The first pointer to be deleted is the Foo base class pointer. When the Foo destructor is declared virtual the DerivedFoo object's destructor is called before the Foo object's destructor. This is shown happening in figure 13.10.

However, when the Foo destructor is not declared to be virtual, the DerivedFoo object's constructor is not called when the Foo pointer is deleted. This is shown happening in figure 13-11. Since a DerivedFoo object was created in memory and its destructor was not called when the base class pointer was deleted, this results in memory leak.

Quick Review

Virtual functions support dynamic binding and object-oriented programming. If a base class function is declared to be virtual and a derived class declares a function with the same function signature as the virtual base class function, the derived class function overrides the base class function and will be called via a base class pointer if the base class pointer points to a derived class object.

PURE VIRTUAL FUNCTIONS

A pure virtual function is a function with no implementation. In the Foo examples used in the previous sections, the getI() function is implemented in the Foo class. It is a virtual function and if a derived class does not provide an overriding version of getI(), the Foo version will be used.

However, in object-oriented design, base classes are used primarily to declare an interface only and the functions declared will have no implementation in the base class. The implementation of the interface, meaning, the implementation of all the interface functions declared in the base class, is left to derived classes.

DECLARING PURE VIRTUAL FUNCTIONS

A pure virtual function is declared by setting the function declaration to zero. This is done via the assignment operator as shown in the following example.

```
virtual void f() = 0;
```

Here, a pure virtual function named f() is declared. The declaration of a pure virtual function can be thought of as saying to the compiler "Don't expect to find a definition of this function in this class. Look to derived classes for the implementation!"

Now, what is the use of a pure virtual function? With pure virtual functions you can create abstract base classes which can then be used to generalize your object-oriented designs. This topic is discussed in the next section.

ABSTRACT CLASSES

An abstract class is a class that contains one or more pure virtual function declarations. There can be no instances of an abstract class but you can create an abstract class pointer and assign to it the address of a derived class object. Are you beginning to notice a theme to this object-oriented programming stuff? It is base class pointers pointing to derived class objects in the quest to achieve polymorphic behavior.

By no instances I mean you cannot create an abstract class object. For example, if you have an abstract class named `AbstractClass` then you cannot do the following:

```
AbstractClass abs_object; //ERROR!
```

Here, an `AbstractClass` object named `abs_object` is declared, however, your compiler will complain. On the other hand, you can declare an `AbstractClass` pointer:

```
AbstractClass* abs_ptr; //OK
```

However, if you try to dynamically create an `AbstractClass` object and assign its address to the pointer your compiler will complain.

```
abs_ptr = new AbstractClass; //ERROR
```

If you create another class named `DerivedClassOne` that inherits from `AbstractClass` and it provides an implementation for `AbstractClass`'s pure virtual functions then you can do the following:

```
abs_ptr = new DerivedClassOne; //OK
```

Now the base class pointer is being assigned the address of a non-abstract derived class object. Another word for non-abstract is concrete. In this example, `DerivedClassOne` represents a concrete implementation of `AbstractClass`.

Let us take a look at an extended example showing abstract classes in use. Figure 13-12 shows a class diagram for a small navy fleet simulation application. The fleet simulation uses three abstract classes, `Vessel`, `Plant`, and `Weapon`. The purpose of each of the abstract classes is to declare an interface that derived classes must implement. The source code for each of the abstract classes appears in examples 13.17 through 13.19.

```

1  #ifndef MY_VESSEL_H 13.17 vessel.h
2  #define MY_VESSEL_H
3  class Plant;
4  class Weapon;
5
6  class Vessel{
7      public:
8          Vessel(Plant &thePlant, Weapon &theWeapon);
9          virtual ~Vessel();
10         virtual void LightOff_Plant() = 0;
11         virtual void ShutDown_Plant() = 0;
12         virtual void Train_Weapon() = 0;
13         virtual void Fire_Weapon() = 0;
14         virtual bool Get_Plant_Status() = 0;
15
16     protected:
17         Plant &GetPlant() {return itsPlant;}
18         Weapon &GetWeapon() {return itsWeapon;}
19
20     private:
21         Plant &itsPlant;
22         Weapon &itsWeapon;
23         static int count;
24     };
25 #endif

```

Example 13.17 uses several language features you have not seen before. First, on lines 3 and 4 the classes `Plant` and `Weapon` are forward declared. You can forward declare a class when you just need the name of the class to be

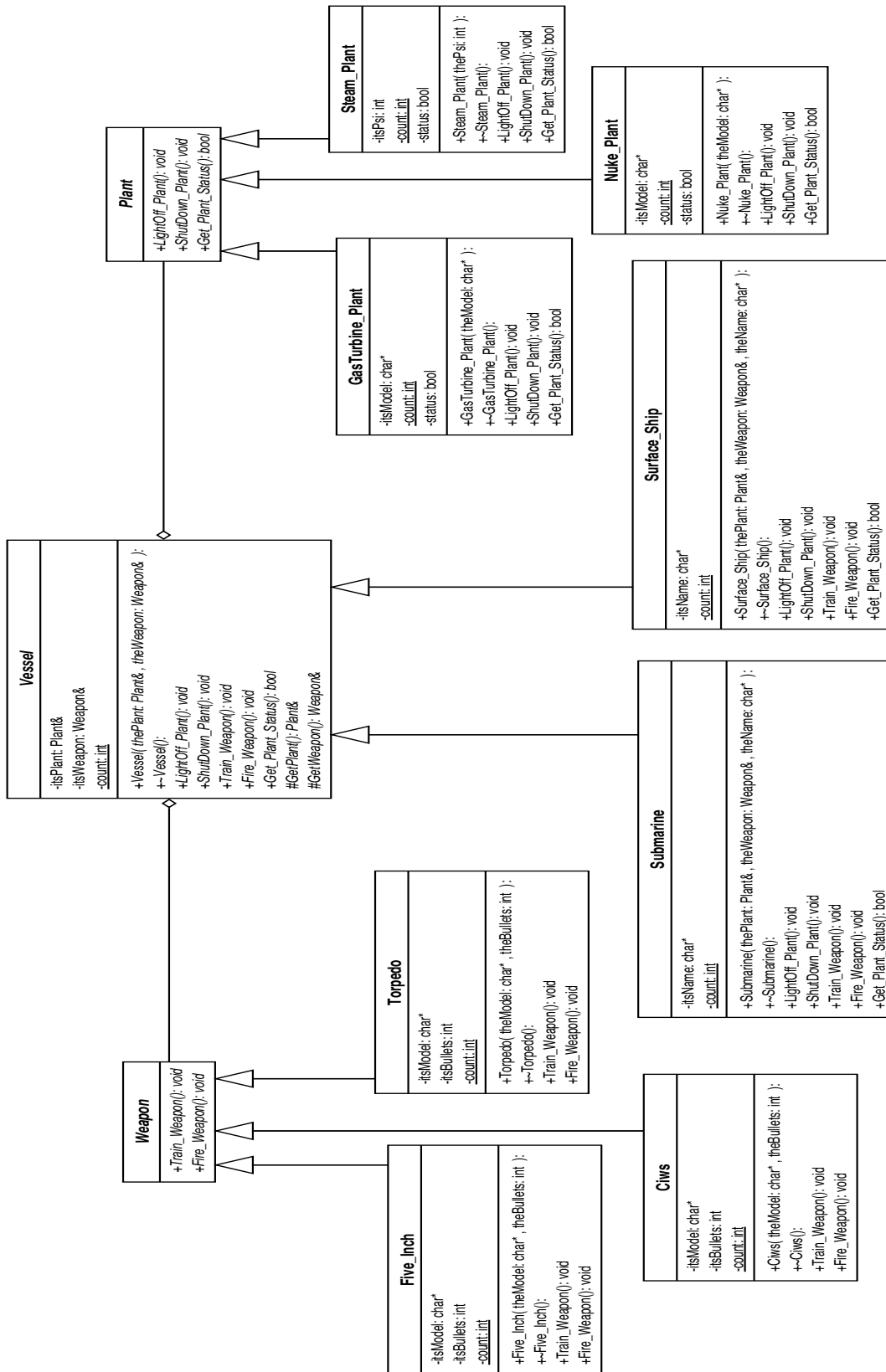


Figure 13-12: Fleet Simulation Class Diagram

```

1  #ifndef MY_PLANT_H
2  #define MY_PLANT_H
3
4  class Plant{
5  public:
6      virtual void LightOff_Plant() = 0;
7      virtual void ShutDown_Plant() = 0;
8      virtual bool Get_Plant_Status() = 0;
9  };
10 #endif

```

13.18 plant.h

```

1  #ifndef MY_WEAPON_H
2  #define MY_WEAPON_H
3
4  class Weapon{
5  public:
6      virtual void Train_Weapon() = 0;
7      virtual void Fire_Weapon() = 0;
8  };
9  #endif

```

13.19 weapon.h

known, rather than all its interface functions. In the Vessel class declaration, the class names Plant and Weapon are simply used to declare constructor parameters and return types. The second language feature used is protected members. The two functions named GetPlant() and GetWeapon() are declared to have protected accessibility. When a derived class inherits from Vessel, it will inherit both of these functions. The private members of Vessel will not be inherited.

All the public interface functions declared in Vessel, with the exception of the constructor, are declared to be virtual. The functions LightOff_Plant(), ShutDown_Plant(), Train_Weapon(), Fire_Weapon(), and Get_Plant_Status() are declared to be pure virtual functions as denoted by their being assigned the value of zero.

Both the Plant and Weapon classes declare an interface only. Notice the lack of an explicitly declared constructor or destructor in each of these classes.

Referring to the fleet simulation class diagram shown in figure 13-12, you can see that a Vessel is comprised of a Plant and a Weapon. The abstract classes declare the type of behavior each of the derived classes must implement. Consider the Weapon class. It declares two interface functions named Train_Weapon() and Fire_Weapon(). In the Weapon class these are pure virtual functions and must be implemented in classes that derive from Weapon. According to the fleet simulation class diagram three classes extend the functionality of Weapon: CIWS (Close In Weapon System), Five_Inch (5" 54 caliber naval gun), and Torpedo. Each of these weapon systems will be aimed and fired in different ways and the source code required to implement that behavior must appear in those concrete classes.

Before discussing how the fleet simulation source code works examine the main.cpp file given in example 13.20 showing the fleet simulation classes in action.

The results of running this program are shown in figure 13-13. The rest of the source code for the fleet simulation follows with line numbers omitted.

```

FleetSimOne.out.out
Constructor: There is 1 vessel.
Constructor: There is 1 submarine.
Constructor: There are 2 vessels.
Constructor: There is 1 surface ship.
Reactor Critical!
Torpedo is locked on target!
Fish In The Water!
Nuke Plant Shut Down!
Steam Plant Lit Off!
Stack Clear!
Five Inch Gun is locked on target!
Blam!
Steam Plant Shut Down!
Destructor: There are 0 surface ships.
Destructor: There is 1 vessel.
Destructor: There are 0 submarines.
Destructor: There are 0 vessels.
|

```

Figure 13-13: Results of Running Example 13.20

13.20 main.cpp

```

1  #include <iostream>
2  #include "vessel.h"
3  #include "submarine.h"
4  #include "Surface_Ship.h"
5  #include "weapon.h"
6  #include "ciws.h"
7  #include "torpedo.h"
8  #include "five_inch.h"
9  #include "plant.h"
10 #include "nuke_plant.h"
11 #include "steam_plant.h"
12 #include "gasturbine_plant.h"
13
14 int main(){
15     /*****
16     Make an array of pointers to base classes
17     *****/
18     Vessel *myNavy[2];
19
20     /*****
21     Make some plants to power the vessels...
22     *****/
23     Nuke_Plant      Nukel("Liquid Metal");
24     Steam_Plant     Steam1(1200);
25     GasTurbine_Plant GasTurbine("LM5000");
26
27     /*****
28     Make some weapons...
29     *****/
30     Torpedo  Torpedol("MK87", 25);
31     Ciws     Ciws1("MK1001", 5000);
32     Five_Inch FiveInchl("Super Shot", 400);
33
34     /*****
35     Construct various vessels...
36     *****/
37     Submarine Sub1(Nukel, Torpedol, "SSN 714");
38     Surface_Ship Ship1(Steam1, FiveInchl, "Skimmer");
39
40     /*****
41     load array with addresses of derived class objects
42     *****/
43     myNavy[0] = &Sub1;
44     myNavy[1] = &Ship1;
45
46     /*****
47     call polymorphic functions via base class pointers
48     *****/
49
50     for(int i = 0; i<2; i++){
51         myNavy[i]->LightOff_Plant();
52         myNavy[i]->Train_Weapon();
53         myNavy[i]->Fire_Weapon();
54         myNavy[i]->ShutDown_Plant();
55     }
56     return 0;
57 }

```

FLEET SIMULATION SOURCE CODE

ciws.h

ciws.h

```

#ifndef MY_CIWS_H
#define MY_CIWS_H
#include "weapon.h"

class Ciws : public Weapon{
public:
    Ciws(char *theModel, int theBullets);

```

```

        virtual ~Ciws();
        virtual void Train_Weapon();
        virtual void Fire_Weapon();

    private:
        char *itsModel;
        int itsBullets;
        static int count;
};
#endif

```

five_inch.h*five_inch.h*

```

#ifndef MY_FIVE_INCH_H
#define MY_FIVE_INCH_H
#include "weapon.h"

class Five_Inch : public Weapon{
public:
    Five_Inch(char *theModel, int theBullets);
    virtual ~Five_Inch();
    virtual void Train_Weapon();
    virtual void Fire_Weapon();

private:
    char *itsModel;
    int itsBullets;
    static int count;
};
#endif

```

torpedo.h*torpedo.h*

```

#ifndef MY_TORPEDO_H
#define MY_TORPEDO_H
#include "weapon.h"

class Torpedo : public Weapon{
public:
    Torpedo(char *theModel, int theBullets);
    virtual ~Torpedo();
    virtual void Train_Weapon();
    virtual void Fire_Weapon();

private:
    char *itsModel;
    int itsBullets;
    static int count;
};
#endif

```

GASTURBINE.H*gasturbine_plant.h*

```

#ifndef MY_GAS_TURBINE_PLANT
#define MY_GAS_TURBINE_PLANT
#include "plant.h"

class GasTurbine_Plant : public Plant{
public:
    GasTurbine_Plant(char *theModel);
    virtual ~GasTurbine_Plant();
    virtual void LightOff_Plant();
    virtual void ShutDown_Plant();
    virtual bool Get_Plant_Status();
private:
    char *itsModel;
    static int count;
    bool status;
};
#endif

```


NUKE_PLANT.H*nuke_plant.h*

```
#ifndef MY_NUKE_PLANT_H
#define MY_NUKE_PLANT_H
#include "plant.h"

class Nuke_Plant : public Plant{
public:
    Nuke_Plant(char *theModel);
    virtual ~Nuke_Plant();
    virtual void LightOff_Plant();
    virtual void ShutDown_Plant();
    virtual bool Get_Plant_Status();
private:
    char *itsModel;
    static int count;
    bool status;
};
#endif
```

STEAM_PLANT.H*steam_plant.h*

```
#ifndef MY_STEAM_PLANT_H
#define MY_STEAM_PLANT_H
#include "plant.h"

class Steam_Plant : public Plant{
public:
    Steam_Plant(int thePsi);
    virtual ~Steam_Plant();
    virtual void LightOff_Plant();
    virtual void ShutDown_Plant();
    virtual bool Get_Plant_Status();
private:
    int itsPsi;
    static int count;
    bool status;
};
#endif
```

SUBMARINE.H*submarine.h*

```
#ifndef MY_SUBMARINE_H
#define MY_SUBMARINE_H
#include "vessel.h"

class Submarine : public Vessel{
public:
    Submarine(Plant &thePlant, Weapon &theWeapon, char *theName);
    virtual ~Submarine();
    virtual void LightOff_Plant();
    virtual void ShutDown_Plant();
    virtual void Train_Weapon();
    virtual void Fire_Weapon();
    virtual bool Get_Plant_Status();
private:
    char *itsName;
    static int count;
};
#endif
```

surface_ship.h*surface_ship.h*

```

#ifndef MY_Surface_Ship_H
#define MY_Surface_Ship_H
#include "vessel.h"

class Surface_Ship : public Vessel{
public:
    Surface_Ship(Plant &thePlant, Weapon &theWeapon, char *theName);
    virtual ~Surface_Ship();
    virtual void LightOff_Plant();
    virtual void ShutDown_Plant();
    virtual void Train_Weapon();
    virtual void Fire_Weapon();
    virtual bool Get_Plant_Status();
private:
    char *itsName;
    static int count;
};
#endif

```

ciws.cpp*ciws.cpp*

```

#include "Ciws.h"
#include <iostream.h>

int Ciws::count = 0;

Ciws::Ciws(char *theModel, int theBullets) : Weapon(), itsBullets(theBullets),
    itsModel(theModel){ Ciws::count++;}

Ciws::~Ciws(){ Ciws::count--;}

void Ciws::Train_Weapon(){
    cout<<"Ciws is locked on target!"<<endl;
}

void Ciws::Fire_Weapon(){
    if(itsBullets){
        cout<<"Sabots Away!"<<endl;
        itsBullets--;
    }
    else
        cout<<"Ciws Out Of Ammo!"<<endl;
}

```

five_inch.cpp*five_inch.cpp*

```

#include "five_inch.h"
#include <iostream.h>

int Five_Inch::count = 0;

Five_Inch::Five_Inch(char *theModel, int theBullets) : Weapon(), itsBullets(theBullets),
    itsModel(theModel){ Five_Inch::count++;}

Five_Inch::~Five_Inch(){ Five_Inch::count--;}

void Five_Inch::Train_Weapon(){
    cout<<"Five Inch Gun is locked on target!"<<endl;
}

void Five_Inch::Fire_Weapon(){
    if(itsBullets)
    {
        cout<<"Blam!"<<endl;
        itsBullets--;
    }
}

```

```

    }
    else
        cout<<"Five Inch Gun Out Of Ammo!"<<endl;
}

```

GASTURBINE_PLANT.CPP

gasturbine_plant.cpp

```

#include <iostream.h>
#include "gasturbine_plant.h"

int GasTurbine_Plant::count = 0;

GasTurbine_Plant::GasTurbine_Plant(char *theModel) : itsModel(theModel), status(false){
    GasTurbine_Plant::count++;
}

GasTurbine_Plant::~GasTurbine_Plant(){
    GasTurbine_Plant::count--;
}

void GasTurbine_Plant:: LightOff_Plant(){
    cout<<"Gas Turbine Plant Lit Off!"<<endl;
    cout<<"All Indications Normal!"<<endl;
    status = true;
}

void GasTurbine_Plant::ShutDown_Plant(){
    cout<<"GasTurbine Plant Shut Down!"<<endl;
    status = false;
}

bool GasTurbine_Plant::Get_Plant_Status(){
    if(status)
        cout<<"The "<<itsModel<<" is running fine!"<<endl;
    else
        cout<<"Gas Turbine Plant Secured!"<<endl;
    return status;
}

```

NUKE_PLANT.CPP

nuke_plant.cpp

```

#include <iostream.h>
#include "nuke_plant.h"

int Nuke_Plant::count = 0;

Nuke_Plant::Nuke_Plant(char *theModel) : itsModel(theModel), status(false){
    Nuke_Plant::count++;
}

Nuke_Plant::~Nuke_Plant(){
    Nuke_Plant::count--;
}

void Nuke_Plant:: LightOff_Plant(){
    cout<<"Reactor Critical!"<<endl;
    status = true;
}

void Nuke_Plant::ShutDown_Plant(){
    cout<<"Nuke Plant Shut Down!"<<endl;
    status = false;
}

bool Nuke_Plant::Get_Plant_Status(){
    if(status)
        cout<<"The "<<itsModel<<" is running fine!"<<endl;
    else
        cout<<"Nuke plant secured!"<<endl;
    return status;
}

```

STEAM_PLANT.CPP*steam_plant.cpp*

```

#include <iostream.h>
#include "steam_plant.h"

int Steam_Plant::count = 0;

Steam_Plant::Steam_Plant(int thePsi) :itsPsi(thePsi), status(false){
    Steam_Plant::count++;
}

Steam_Plant::~Steam_Plant(){
    Steam_Plant::count--;
}

void Steam_Plant:: LightOff_Plant(){
    cout<<"Steam Plant Lit Off!"<<endl;
    cout<<"Stack Clear!"<<endl;
    status = true;
}

void Steam_Plant::ShutDown_Plant(){
    cout<<"Steam Plant Shut Down!"<<endl;
    status = false;
}

bool Steam_Plant::Get_Plant_Status(){
    if(status)
        cout<<"Plant's running fine! There's "<<itsPsi<<" psi at the main stop valves!"<<endl;
    else
        cout<<"Steam plant secured!"<<endl;
    return status;
}

```

SUBMARINE.CPP*submarine.cpp*

```

#include "submarine.h"
#include <iostream.h>
#include "plant.h"
#include "weapon.h"

int Submarine::count = 0;

Submarine::Submarine(Plant &thePlant, Weapon &theWeapon, char *theName)
    : Vessel(thePlant, theWeapon), itsName(theName){
    if(++Submarine::count) == 1)
        cout<<"Constructor: There is "<<Submarine::count<<" submarine."<<endl;
    else
        cout<<"Constructor: There are "<<Submarine::count<<" submarines."<<endl;
}

Submarine::~Submarine(){
    if(--Submarine::count) == 1)
        cout<<"Destructor: There is "<<Submarine::count<<" submarine."<<endl;
    else
        cout<<"Destructor: There are "<<Submarine::count<<" submarines."<<endl;
}

void Submarine::LightOff_Plant(){
    GetPlant().LightOff_Plant();
}

void Submarine::ShutDown_Plant(){
    GetPlant().ShutDown_Plant();
}

void Submarine::Train_Weapon(){
    GetWeapon().Train_Weapon();
}

void Submarine::Fire_Weapon(){

```

```

    GetWeapon().Fire_Weapon();
}

bool Submarine::Get_Plant_Status() {
    return GetPlant().Get_Plant_Status();
}

```

surface_ship.cpp

surface_ship.cpp

```

#include "Surface_Ship.h"
#include <iostream.h>
#include "plant.h"
#include "weapon.h"

int Surface_Ship::count = 0;

Surface_Ship::Surface_Ship(Plant &thePlant, Weapon &theWeapon, char *theName)
    : Vessel(thePlant, theWeapon), itsName(theName){
    if(++Surface_Ship::count == 1)
        cout<<"Constructor: There is "<<Surface_Ship::count<<" surface ship."<<endl;
    else
        cout<<"Constructor: There are "<<Surface_Ship::count<<" surface ships."<<endl;
}

Surface_Ship::~Surface_Ship(){
    if(--Surface_Ship::count == 1)
        cout<<"Destructor: There is "<<Surface_Ship::count<<" surface ship."<<endl;
    else
        cout<<"Destructor: There are "<<Surface_Ship::count<<" surface ships."<<endl;
}

void Surface_Ship::LightOff_Plant(){
    GetPlant().LightOff_Plant();
}

void Surface_Ship::ShutDown_Plant(){
    GetPlant().ShutDown_Plant();
}

void Surface_Ship::Train_Weapon(){
    GetWeapon().Train_Weapon();
}

void Surface_Ship::Fire_Weapon(){
    GetWeapon().Fire_Weapon();
}

bool Surface_Ship::Get_Plant_Status(){
    return GetPlant().Get_Plant_Status();
}

```

torpedo.cpp

torpedo.cpp

```

#include "torpedo.h"
#include <iostream.h>

int Torpedo::count = 0;

Torpedo::Torpedo(char *theModel, int theBullets) : Weapon(), itsBullets(theBullets),
    itsModel(theModel){Torpedo::count++;}

Torpedo::~Torpedo(){Torpedo::count--;}

void Torpedo::Train_Weapon(){
    cout<<"Torpedo is locked on target!"<<endl;
}

void Torpedo::Fire_Weapon(){
    if(itsBullets)
    {
        cout<<"Fish In The Water!"<<endl;
        itsBullets--;
    }
}

```

```

    else
        cout<<"Fresh Out Of Torpedos Captain!"<<endl;
}

```

vessel.cpp

vessel.cpp

```

#include "vessel.h"
#include <iostream.h>

int Vessel::count = 0;

Vessel::Vessel(Plant &thePlant, Weapon &theWeapon) : itsPlant(thePlant),
    itsWeapon(theWeapon) {
    if(++Vessel::count) == 1)
        cout<<"Constructor: There is "<<Vessel::count<<" vessel."<<endl;
    else
        cout<<"Constructor: There are "<<Vessel::count<<" vessels."<<endl;
}

Vessel::~Vessel(){
    if(--Vessel::count) == 1)
        cout<<"Destructor: There is "<<Vessel::count<<" vessel."<<endl;
    else
        cout<<"Destructor: There are "<<Vessel::count<<" vessels."<<endl;
}

```

Multiple Inheritance

A class can inherit the functionality from more than one class which means it will have more than one base class. The act of inheriting the functionality from multiple classes is referred to as multiple inheritance. I will demonstrate multiple inheritance by creating a simple payroll application. Figure 13-14 gives the class diagram.

Referring to figure 13-14, the Person class is reused code from chapter 11. There are two abstract classes: Payable and Employee. Payable is abstract because it declares a pure virtual function named pay(). In fact, the only thing the Payable class does is declare this virtual function. In this regard, Payable is publishing an interface to which subclasses either must implement or let further subclasses implement.

The Employee class inherits from Person and Payable. So, an Employee is a Person and is a Payable. Because the Person class provides all the functionality of storing a person's name and other such information the Employee class need not worry about replicating that functionality. Since an Employee is a Person, any class that inherits from Employee in the future will also be a Person and inherit the functionality of Person.

The relationship between Employee and Payable is somewhat different than that between Employee and Person. This is due to the pay() function being a pure virtual function. The Employee class must either implement pay() or not. If it does not, which is the case here, the Employee class becomes abstract, meaning there can be no instances of Employee and the pay() function must be implemented by classes derived from Employee.

The HourlyEmployee and SalariedEmployee classes inherit from Employee. This makes both of them an Employee, a Person, and a Payable. Both HourlyEmployee and SalariedEmployee have all the functionality of Employee and Person, however, since the pay() function was not implemented in the Employee class they each must declare and define what the pay() function means to them. The assumption made here is that an HourlyEmployee's pay will be calculated differently from a SalariedEmployee's pay.

In addition to multiple inheritance this simple payroll application demonstrates virtual function overriding and dynamic polymorphic behavior. Let us take a closer look at this application starting with the Payable class whose code is given in example 13.21.

The Payable class simply declares the pure virtual function pay(). The code for the Employee class is given in Example 13.22.

Line 6 of example 13.22 begins the Employee class declaration. Notice that additional base classes are separated by a comma. The Employee class implementation code is given in example 12.23.

You may be surprised at the small size of this file. Since most of the work is done in the Person class there is little

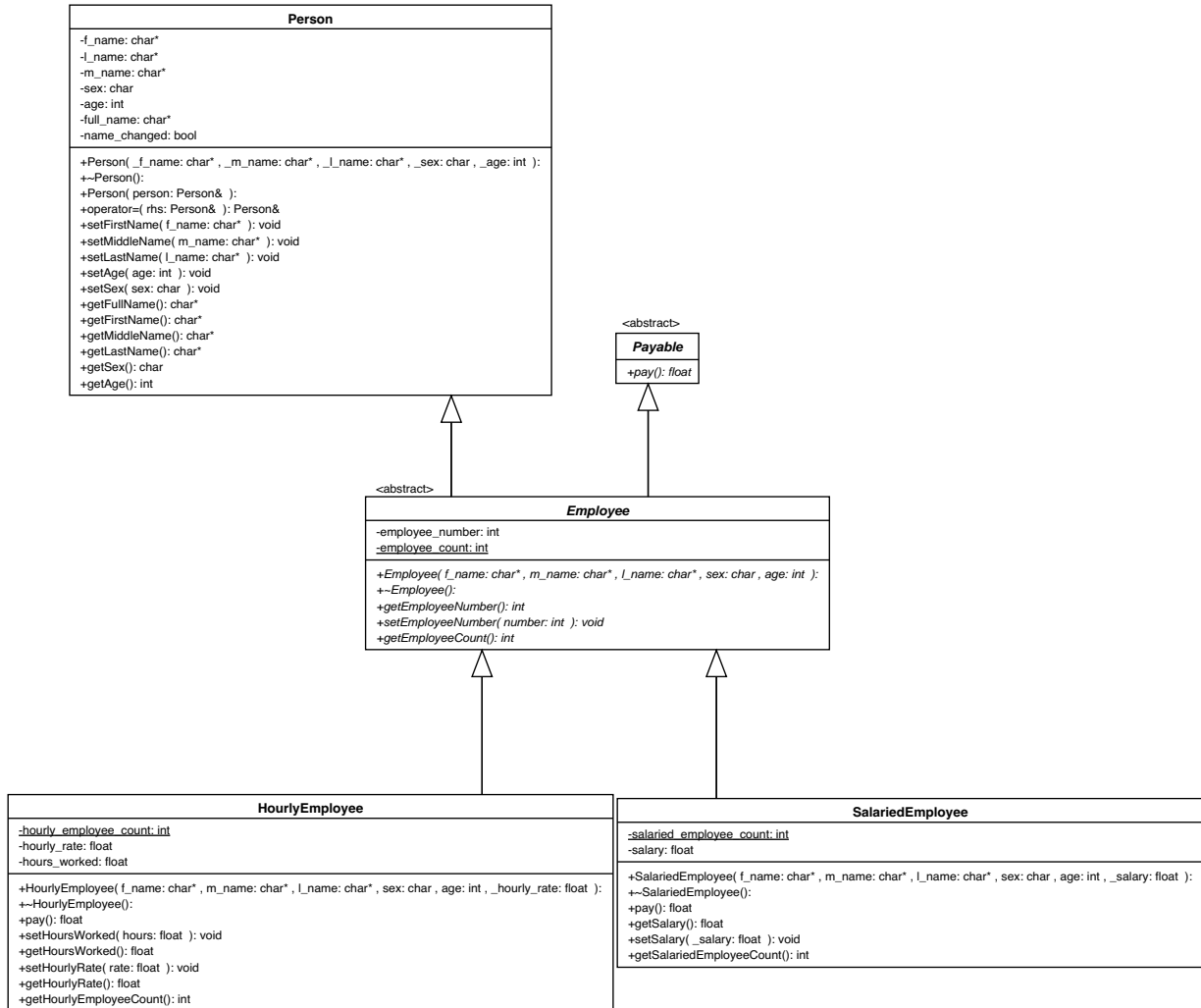


Figure 13-14: Payroll Application Class Diagram

```

1  #ifndef PAYABLE_H
2  #define PAYABLE_H
3
4  class Payable {
5      public:
6          virtual float pay() = 0;
7  };
8  #endif
  
```

13.21 payable.h

for the Employee class to do except implement the three public functions `getEmployeeNumber()`, `setEmployeeNumber()` and `getEmployeeCount()`. The Employee class does not implement the `pay()` function; that will be left to the derived classes `HourlyEmployee` and `SalariedEmployee`. The class declarations for each of these files is given in examples 13.24 and 13.25.

Notice how each of these classes have only to inherit from `Employee`. Since an `Employee` is a `Person` and is a `Payable`, any class that inherits from `Employee` will be an `Employee`, a `Person`, and a `Payable`, with access to all the functionality the inheritance relationship offers. Also notice that both `HourlyEmployee` and `SalariedEmployee`

```

1  #ifndef EMPLOYEE_H
2  #define EMPLOYEE_H
3  #include "person.h"
4  #include "payable.h"
5
6  class Employee : public Person, public Payable{
7      public:
8          Employee(char* f_name = "Brand", char* m_name = "New", char* l_name = "Employee",
9                  char sex = 'M', int age = 18);
10         virtual ~Employee();
11         int getEmployeeNumber();
12         void setEmployeeNumber(int number);
13         int getEmployeeCount();
14
15     private:
16         int employee_number;
17         static int employee_count;
18 };
19
20 #endif

```

13.22 *employee.h*

```

1  #include "person.h"
2  #include "employee.h"
3
4
5  int Employee::employee_count = 0;
6
7  Employee::Employee(char* f_name, char* m_name, char* l_name, char sex,
8                    int age):Person(f_name, m_name, l_name, sex, age){
9      employee_number = ++employee_count;
10 }
11
12 Employee::~~Employee(){}
13
14 int Employee::getEmployeeNumber(){return employee_number;}
15
16 void Employee::setEmployeeNumber(int number){
17     employee_number = number;
18 }
19
20 int Employee::getEmployeeCount(){ return employee_count;}

```

13.23 *employee.cpp*

```

1  #ifndef HOURLY_EMPLOYEE_H
2  #define HOURLY_EMPLOYEE_H
3  #include "employee.h"
4
5  class HourlyEmployee : public Employee{
6      public:
7          HourlyEmployee(char* f_name = "Brand", char* m_name = "New", char* l_name = "Employee",
8                        char sex = 'M', int age = 18, float _hourly_rate = 0);
9          virtual ~HourlyEmployee();
10         float pay();
11         void setHoursWorked(float hours);
12         float getHoursWorked();
13         void setHourlyRate(float rate);
14         float getHourlyRate();
15         int getHourlyEmployeeCount();
16     private:
17         static int hourly_employee_count;
18         float hourly_rate;
19         float hours_worked;
20 };
21 #endif

```

13.24 *hourlyemployee.h*

declare the `pay()` function. This function will override the pure virtual `pay()` function in `Employee`, which is the same pure virtual `pay()` function declared in `Payable`.

The implementation code for the `HourlyEmployee` and `SalariedEmployee` classes is given in examples 13.26 and 13.27.

Since most of the work of being a `Person` and an `Employee` is already done, each derived class need only implement any additional functions they have declared, including the overriding `pay()` function. Most of the work for each of these classes takes place in their respective constructors which must call the `Employee` base class constructor in the

13.25 *salariedemployee.h*

```

1 #ifndef SALARIED_EMPLOYEE_H
2 #define SALARIED_EMPLOYEE_H
3 #include "employee.h"
4
5 class SalariedEmployee : public Employee{
6     public:
7         SalariedEmployee(char* f_name = "New", char* m_name = "Salaried",
8                         char* l_name = "Employee", char sex = 'M', int age = 18,
9                         float _salary = 0);
10        virtual ~SalariedEmployee();
11        float pay();
12        float getSalary();
13        void setSalary(float _salary);
14        int getSalariedEmployeeCount();
15    private:
16        static int salaried_employee_count;
17        float salary;
18 };
19 #endif

```

13.26 *hourlyemployee.cpp*

```

1 #include "employee.h"
2 #include "hourlyemployee.h"
3
4
5 int HourlyEmployee::hourly_employee_count = 0;
6
7 HourlyEmployee::HourlyEmployee(char* f_name, char* m_name, char* l_name,
8                               char sex, int age, float _hourly_rate):
9     Employee(f_name, m_name, l_name, sex, age),
10    hourly_rate(_hourly_rate),
11    hours_worked(0){
12        hours_worked(0){
13            hourly_employee_count++;
14        }
15
16    HourlyEmployee::~~HourlyEmployee() {hourly_employee_count--;}
17
18    float HourlyEmployee::pay() {return (hourly_rate * hours_worked);}
19
20    void HourlyEmployee::setHoursWorked(float hours) {
21        hours_worked = hours;
22    }
23
24    float HourlyEmployee::getHoursWorked() {return hours_worked;}
25
26    void HourlyEmployee::setHourlyRate(float rate) {
27        hourly_rate = rate;}
28
29    float HourlyEmployee::getHourlyRate() {return hourly_rate;}
30
31    int HourlyEmployee::getHourlyEmployeeCount() {return hourly_employee_count;}

```

initializer list using its parameters. Example 13.28 gives a main.cpp file showing the classes of the payroll application in action. Figure 13-15 shows the results of running example 13.28.

```

MultipleInheritance.out.out
80 8 640
67000 2576.92

#      First      Middle      Last      Pay
-----
1      Bob        J          Jones     $   640
2      Sue        Mae        Lind      $ 2576.92

```

Figure 13-15: Results of Running Example 13.28

13.27 *salariedemployee.cpp*

```

1  #include "employee.h"
2  #include "salariedemployee.h"
3
4  int SalariedEmployee::salaried_employee_count = 0;
5
6  SalariedEmployee::SalariedEmployee(char* f_name, char* m_name, char* l_name,
7      char sex, int age, float _salary):
8      Employee(f_name, m_name, l_name, sex, age), salary(_salary){
9      salaried_employee_count++;
10 }
11
12 SalariedEmployee::~SalariedEmployee() {salaried_employee_count--;}
13
14 float SalariedEmployee::pay() { return (salary/26);}
15
16 float SalariedEmployee::getSalary() {return salary;}
17
18 void SalariedEmployee::setSalary(float _salary){
19     salary = _salary;
20 }
21
22 int SalariedEmployee::getSalariedEmployeeCount() {
23     return salaried_employee_count;
24 }

```

13.28 *main.cpp*

```

1  #include <iostream>
2  using namespace std;
3  #include <iomanip.h>
4  #include "employee.h"
5  #include "hourlyemployee.h"
6  #include "salariedemployee.h"
7
8  int main(){
9      Employee* employees[2];
10
11      HourlyEmployee Bob("Bob", "J", "Jones");
12      Bob.setHoursWorked(80);
13      Bob.setHourlyRate(8.00);
14      cout<<Bob.getHoursWorked()<<" "<<Bob.getHourlyRate()<<" "<<Bob.pay()<<endl;
15
16      SalariedEmployee Sue("Sue", "Mae", "Lind", 'F', 23, 67000.00);
17      cout<<Sue.getSalary()<<" "<<Sue.pay()<<endl;
18
19      employees[0] = &Bob;
20      employees[1] = &Sue;
21
22      cout<<endl<<endl<<endl;
23
24      cout<<"# "<<setw(15)<<"First"<<setw(15)<<"Middle"<<setw(15)<<"Last"<<setw(10)<<"Pay"<<endl;
25      cout<<"-----"<<endl;
26
27
28      for(int i=0; i<2; i++){
29          cout<<employees[i]->getEmployeeNumber()<<setw(15)<<employees[i]->getFirstName()
30              <<setw(15)<<employees[i]->getMiddleName()<<setw(15)<<employees[i]->getLastName()
31              <<"    "$<<setw(8)<<employees[i]->pay()<<endl;
32
33      }
34      return 0;
35 }

```

VIRTUAL BASE CLASSES: VIRTUAL INHERITANCE

Pause here for a moment and consider the multiple inheritance example presented in the previous section. When a derived class object of type `HourlyEmployee` or `SalariedEmployee` is created in memory so too are their related base class objects created in memory. Every instance of `HourlyEmployee` will have its very own `Employee` and `Person` base class objects. In the context of the previous example this is exactly the object behavior desired. But what happens when a subclass inherits from two separate classes who themselves inherit from a common base class? You may or may not want multiple copies of the base class created in memory. To control the creation of base classes in a

multiple inheritance design you use the virtual keyword to denote virtual inheritance so that only one copy of the base class object is created.

Figure 13-16 shows a class diagram with four classes: A, B, C, & D. Class D inherits from classes B and C.

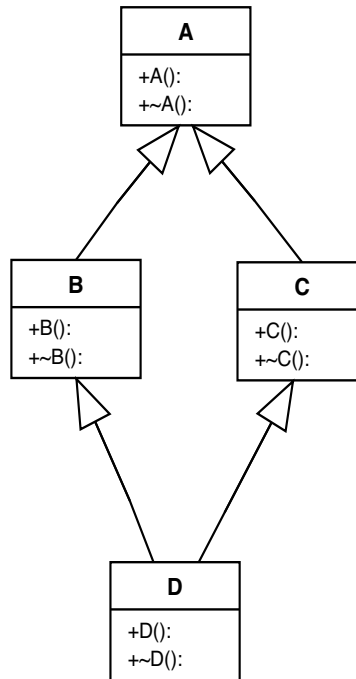


Figure 13-16: Class Diagram Showing Common Base Class Inheritance

Classes B and C inherit from class A. Using normal, non-virtual inheritance, when a class D object is created, a class B object will be created along with its class A object. A class C object will also be created along with its class A object. This will result in two class A objects existing in memory at the same time as shown in figure 13-17.

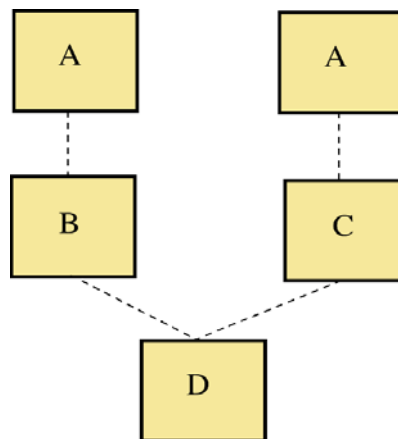


Figure 13-17: Non-Virtual Inheritance Will Result in Multiple Instances of Base Classes

In this example, when a D object is created, it would be desirable to have a B and C object created, but not necessarily desirable to have two copies of an A object. (*desirability being a matter of design*) The non-virtual inheritance implementation of the class diagram shown in figure 13-16 is given in examples 13-29 through 13-33. In these examples I have used inline functions to save space. All the constructors and destructors do is print simple messages so you can trace object creation.

Referring to the code for classes B and C, notice how they publicly inherit from class A in the usual fashion. This

```

1  #ifndef A_H
2  #define A_H
3  #include <iostream>
4  using namespace std;
5
6  class A {
7  public:
8      A(){cout<<"A constructor"<<endl;}
9      virtual ~A(){cout<<"A destructor"<<endl;}
10 };
11 #endif

```

13.29 a.h

```

1  #ifndef B_H
2  #define B_H
3  #include "a.h"
4  #include <iostream>
5  using namespace std;
6
7  class B : public A{
8  public:
9      B(){cout<<"B constructor"<<endl;}
10     virtual ~B(){cout<<"B destructor"<<endl;}
11 };
12 #endif

```

13.30 b.h

```

1  #ifndef C_H
2  #define C_H
3  #include <iostream>
4  using namespace std;
5  #include "a.h"
6
7  class C : public A{
8  public:
9      C(){cout<<"C constructor"<<endl;}
10     virtual ~C(){cout<<"C destructor"<<endl;}
11 };
12 #endif

```

13.31 c.h

```

1  #ifndef D_H
2  #define D_H
3  #include <iostream>
4  using namespace std;
5  #include "b.h"
6  #include "c.h"
7
8  class D : public B, public C {
9  public:
10     D(){cout<<"D constructor"<<endl;}
11     virtual ~D(){cout<<"D destructor"<<endl;}
12 };
13 #endif

```

13.32 d.h

```

1  #include <iostream>
2  #include "d.h"
3
4  int main () {
5      D d;
6      return 0;
7  }

```

13.33 main.cpp

is non-virtual inheritance and is the usual form of inheritance you will employ in your design. Class D multiply inherits from class B and C. The creation of a class D object results in an object creation chain reaction as is shown in figure 13-18.

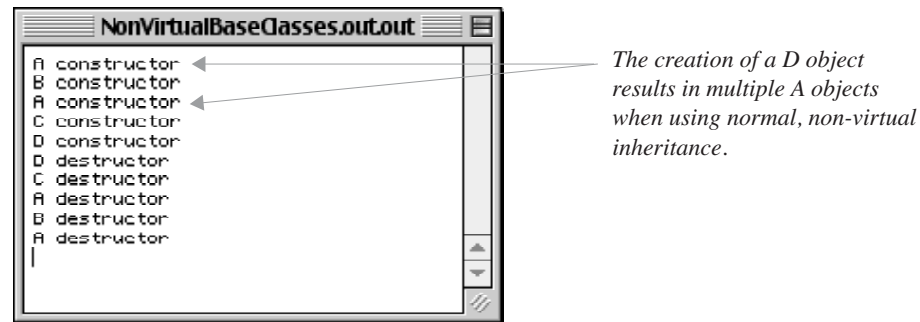


Figure 13-18: Results of Running Example 13.33.

To prevent multiple A instances classes B and C must virtually inherit from A. Only the B and C class declarations need be modified. Examples 13.34 and 13.35 give the modified code.

```

1  #ifndef B_H
2  #define B_H
3  #include "a.h"
4  #include <iostream>
5  using namespace std;
6
7  class B : virtual public A{
8  public:
9      B(){cout<<"B constructor"<<endl;}
10     virtual ~B(){cout<<"B destructor"<<endl;}
11 };
12 #endif

```

13.34 b.h

Add virtual keyword

```

1  #ifndef C_H
2  #define C_H
3  #include <iostream>
4  using namespace std;
5  #include "a.h"
6
7  class C : virtual public A{
8  public:
9      C(){cout<<"C constructor"<<endl;}
10     virtual ~C(){cout<<"C destructor"<<endl;}
11 };
12 #endif

```

13.35 c.h

Add virtual keyword

Now, when example 13.33 is run again it results in a different output as shown in figure 13-19.

Figure 13-20 shows graphically the relationship of A, B, C, and D objects in memory when virtual inheritance is used.

GETTING INHERITANCE RIGHT: SOME POINTS TO CONSIDER

Of all the object-oriented topics, the proper definition and use of inheritance causes the most controversy in both the academic and professional communities. Essentially, getting inheritance right is not easy.

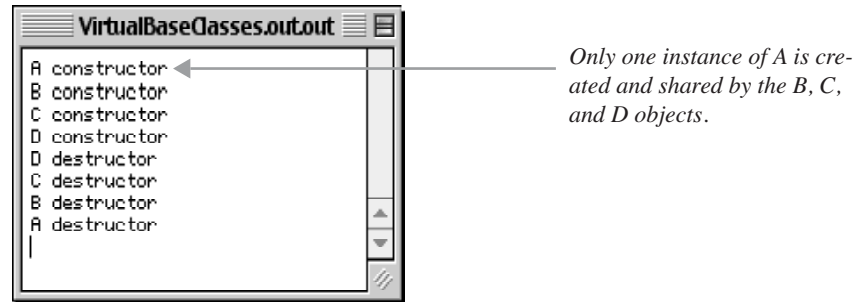


Figure 13-19: Results of Running 13.33 Showing Effects of Virtual Inheritance

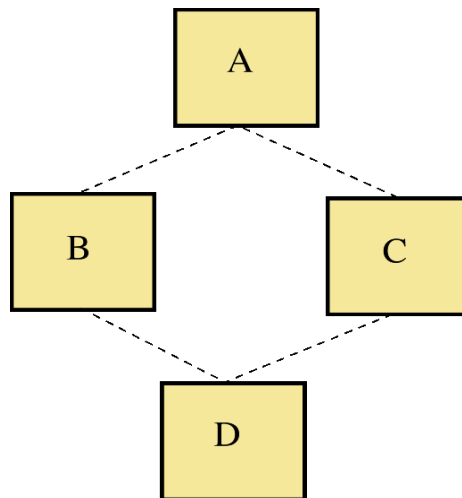


Figure 13-20: Virtual Inheritance Results in One Instance of A

TWO DIFFERENT USES OF INHERITANCE

There are two primary uses of inheritance: as an enabler of advanced, object-oriented reasoning about the structural aspects of a program, and second, as a way to incrementally evolve code.

REASONING ABOUT OBJECT-ORIENTED APPLICATION DESIGN

One use of inheritance is as an aid to the programmer helping them reason about the nature or structure of a program. When using inheritance in this fashion a programmer will think of class types and their subtypes and how these conceptual entities fulfill their role in an application's design. Using inheritance in this way is perhaps the quintessential difference between the object-oriented programmer and the procedural programmer. Inheritance is the enabler of object-oriented programming. Understanding the use of inheritance in this way results in the "light bulb" going on in a programmer's head.

INCREMENTAL CODE EVOLUTION

Another use of inheritance is to achieve incremental code evolution. For instance, when Employee inherits from Person, the only thing an Employee class must do is implement the functionality that differentiates itself from its superclass. Inheritance in this regard facilitates differential or incremental program development. The Person class represents a generalization, whereas the Employee class represents a specialization. Using inheritance in this fashion is fine so long as subclasses are in fact providing a strict specialization of the superclass. However, when you override a base class function in a derived class, the only "rule" C++ enforces is that the derived class function have the same

name (signature) as the base class function it overrides. This opens the possibility of unexpected behavior being introduced into an overriding derived class function that breaks the code that might depend on the base class implementation of that function.

PROTECT YOURSELF IN YOUR DESIGN

To generally ensure the good use of inheritance in your design you can adopt the following strategy: First, use abstract base classes containing nothing but pure virtual functions to specify the conceptual design of your application at the highest level. This is the approach taken in the fleet simulation application shown in figure 13-12. A pure virtual function has no behavior until it is implemented in a derived class. Thinking in terms of abstract base classes, their interfaces, and their relationships to other abstract base classes is applying inheritance as an aid to conceptual modeling. Implementing abstract base classes and inheriting from them to gain the subtype or “is a...” relationship only is referred to as interface inheritance.

Second, when inheriting from a concrete class such as `Person`, ensure the derived class is truly a specialization in that it only provides functionality not found in the base class.

Third, avoid, when possible, the urge to override a virtual concrete base class function unless you study the “rules” of doing so as set forth by the designer of the base class code.

SUMMARY

A class type can exhibit the behavior of another class type through the mechanism of inheritance. The class whose behavior is inherited is called the base class; the class inheriting the behavior is called the derived class. A public base class function can be called via a derived class object. A base class pointer can be assigned the addresses of a derived class object, however, since the pointer is of a base class type, only the public functions declared in the base class can be called via the pointer unless the base class function is declared to be virtual and there is an overriding function in a derived class.

The access specifiers `public`, `protected`, and `private` are used to control vertical access between base and derived classes.

When a derived class must initialize base class attributes it must make an explicit base class constructor call in its initializer list. The derived class constructor can use its parameters as arguments to its base class constructor. If a default constructor exists for both the base and derived class then some or all of the constructor arguments can be omitted when objects are instantiated.

A function appearing in a derived class having the same function signature as a base class function merely hides the base class function. Virtual functions support dynamic binding and object-oriented programming. If a base class function is declared to be virtual and a derived class declares a function with the same function signature as the virtual base class function, the derived class function overrides the base class function and will be called via a base class pointer if the base class pointer points to a derived class object.

A class can inherit from more than one class. This is called multiple inheritance. Be careful when using multiple inheritance, especially in complex inheritance hierarchies where one class may inherit from two different classes who share a common base class. Use virtual inheritance to control the number of base class instances created in a multiple inheritance hierarchy.

Dynamic polymorphic behavior is achieved through the use of virtual base class functions being overridden in derived classes. Base class pointers can be assigned the addresses of derived class objects and overriding functions called via the base class pointer.

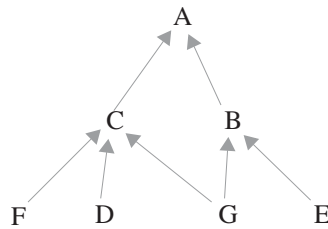
Skill Building Exercises

1. **Extend Person:** Create a class called `Faculty` that extends the functionality of `Person`. (See chapter 11 for the complete code for the `Person` class). Give the `Faculty` class some unique attributes such as faculty number, department code, etc. Declare and implement any necessary derived class accessor and mutator functions. Write a driver pro-

gram and test the Faculty class by creating Faculty objects and calling both derived class and base class functions via the Faculty objects. Use the Person/Student example presented in this chapter as a guide.

2. **Sensor Class:** Create an abstract class called Sensor. In Sensor declare a pure virtual function named `getReading()` that returns a float. Create two classes that derive from Sensor named `TemperatureSensor` and `OxygenSensor`. In each of these classes implement the `getReading()` method. In `TemperatureSensor` the `getReading()` method should return a float value between absolute zero and +4000 degrees centigrade. Use the standard C library `rand()` function to generate a random float value within this range. In `OxygenSensor` implement the `getReading()` function to return a float value between 0 and 100. Again, use the `rand()` function to randomly generate the necessary values each time the `getReading()` function is called. Write a driver to test your sensor objects. Create a few `TemperatureSensor` and `OxygenSensor` objects and test the `getReading()` function. Declare a Sensor base class pointer and dynamically assign the address of either a `TemperatureSensor` or `OxygenSensor` object. Call the `getReading()` polymorphically through the base class pointer.
3. **Research:** Study the C++ stream classes. Draw the inheritance hierarchy. How does each derived class build upon the functionality of its base class?
4. **Pop Quiz:** Explain the difference between function hiding and overriding.
5. **Research:** Continue your study of the topic of inheritance. Your objective is to gain a comfortable feeling for the different ways inheritance is used to achieve object-oriented design goals. Study how languages other than C++ allow programmers to implement their inheritance designs. A good place to start your research is by reading the papers listed in the reference section at the end of this chapter. If you are a member of the Association of Computing Machinery (ACM) they have an excellent online digital library. The web in general has lots of material on this topic.
6. **Program:**
 - a. Create a class named `MyBaseClass` and declare a constructor, and destructor, and two other public interface functions named `f()`, and `g()` that return void. The only functionality each function should have is to print a simple message to the screen using `iostream` output. Leave the keyword `virtual` off the destructor, `f()`, and `g()` for now. Write a `main()` function to test `MyBaseClass`.
 - b. Once `MyBaseClass` is tested, create a new class that derives from `MyBaseClass` named `MyDerivedClass`. Declare a constructor and destructor that print trace messages but no other functions. Test `MyDerivedClass` by declaring a `MyDerivedClass` object and calling the `f()` and `g()` functions.
 - c. In the `main()` function declare a `MyBaseClass` pointer and use the `new` operator to create a `MyDerivedClass` object and assign its address to the pointer. Call the functions `f()` and `g()` via the pointer. Do not forget to delete the pointer at the end of the program! Run the program and note the trace messages. specifically the destructor messages. Are they all there? If not, why?
 - d. Modify the `MyBaseClass` destructor and add the keyword `virtual`. Run the program again and note the trace messages. Are all the destructor messages there now?
 - e. Modify the `MyDerivedClass` by adding a new function named `f()`. Make `f()` print a message that is different from the `MyBaseClass` version of `f()`. Run the program again and note the trace messages. Are they now different? If so, how? If not, what would you have to do to get the derived class version of `f()` to be called via the base class pointer? (Hint: virtual base class functions)
7. **Pop Quiz:** Why can you not instantiate an abstract base class.
8. **Pop Quiz:** Explain the role abstract base classes play in designing with inheritance. If an abstract base class has only a set of public pure virtual functions, what is it good for? If you find it hard to answer this question confidently revisit skill building exercise #5.

9. **Program:** a. Create a set of classes named A, B, C, D, E, F, & G with the following inheritance relationship:



Give each class a constructor and destructor only that print simple trace messages to the screen. Use non-virtual inheritance. Ensure your destructors are declared to be virtual. Test each of the classes individually by creating objects of each type. Study the constructor and destructor trace messages.

b. Create a pointer to class A. Assign it the address of a dynamically created E object. Which objects are created?

c. Next, assign to the A pointer the address of a dynamically created G object. Which objects are created? How many A objects are created?

d. Next, assign to the A pointer the address of a dynamically created D object. Which objects are created? How many A objects are created?

e. Modify only the B class to virtually inherit from A. Assign to the A pointer the address of a dynamically created G object. Did this result in a different set of trace messages? If so, explain what has changed?

10. **Program:** a. Create an abstract class named AbstractClass that declares one pure virtual function named f(). Next, create three derived classes that inherit from AbstractClass named DerivedOne, DerivedTwo, and DerivedThree. In each of the derived classes implement the f() function to print a simple message to the screen. In a main() function create an array of three AbstractClass pointers and dynamically allocate one object of each derived class type and assign its address to an array element. Example:

```
abstract_array[0] = new DerivedOne;
```

b. Call the f() function via the pointers stored in each array element.

SUGGESTED PROJECTS

1. **Convert Engine Simulation:** Convert the aircraft engine simulation code presented in chapter 12 to use abstract base classes. Essentially, all components should derive from an abstract base class. The Engine itself could be abstract and be an aggregate of abstract component types. Refer to the fleet simulator example in this chapter for an example of designing with abstract base class aggregates.

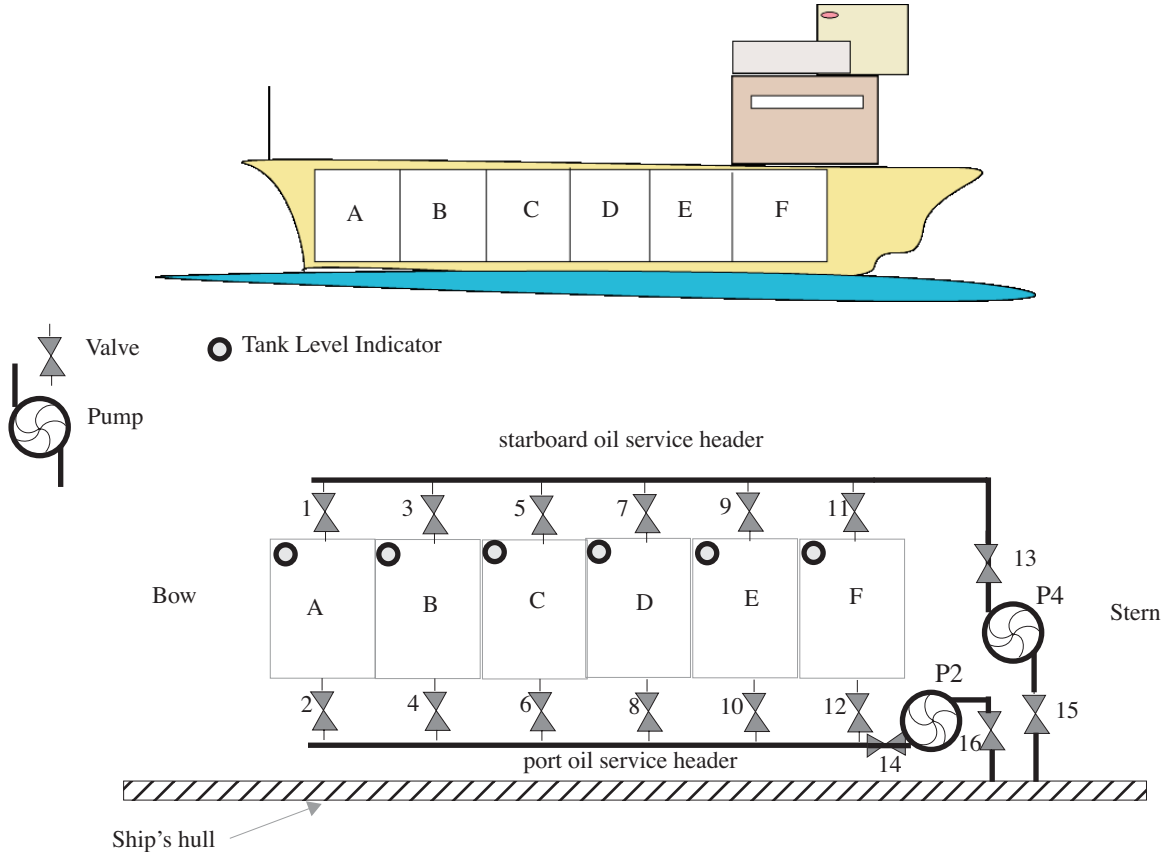
2. **Automobile Sensor System:** Design and create an automobile sensor system. The design may be similar to the aircraft engine design in the previous project. Add another component called Computer that monitors the various engine components via the sensors.

3. **Extend Fleet Simulation:** Extend the fleet simulation example presented in this chapter. Add several more weapon and plant types. Create several vessels using your new weapons and plants and test their functionality.

4. **Credit Card Validation System:** Design and implement a credit card validation system. Start by writing a descrip-

tion of the various components that may be required to implement the system. From this description pick out potential objects and the messages that would need to be sent between them. Model your design in UML if you have a UML design tool.

5. **Fuel Oil System:** Design and create an oil tanker ship tank fill, drain, and level control and indicator system. Assume your tanker ship has six oil cargo compartments as shown in the diagram below.



Each compartment can be filled and drained from either the starboard or port oil service header. The oil service system consists of 16 valves, 2 pumps, and 6 tank level indicators. The valves are remote controlled as are the pumps. The valves can be either open or shut. The pumps can be on or off. When a pump is on, it can be in one of two speed states: slow & fast, and one of two direction states: drain & fill. When running in the drain direction the pump is taking a suction from the tank side. When running in the fill direction it is taking a suction from the hull side.

Now, to fill oil cargo tank A quickly you could open valves 1, 2, 14, 16, 13, & 15, start pumps P2 and P4 in the fast/fill mode and monitor the tank level indicator on cargo tank A.

Assume a tank capacity of 75,000 gallons and pump flow rate of 1000 gal/min. in fast mode.

6. **University Enrollment System:** Create a university student enrollment tracking system. Identify the classes you will need. Reuse any code from this chapter you deem necessary. Research the `iostream` file input and output classes to save student and course data to disk for later use.

SELF TEST QUESTIONS

1. Describe the differences between horizontal and vertical access. Describe how to use the keywords `public`, `protected`, and `private` to control horizontal and vertical access.
2. When can you get away with forward declaring a class name vs. including the whole header file?
3. Describe, in your own words, the purpose of inheritance. In what two primary ways is inheritance used?
4. Describe how to hide a base class function with a derived class function.
5. How do you override a base class function in a derived class?
6. What is a virtual function?
7. What is a pure virtual function?
8. What is the purpose of a virtual destructor?
9. Describe how to achieve dynamic polymorphic behavior.
10. In what part of a derived class constructor do you call the base class constructor?

REFERENCES

Antero Taivalsaari. *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, No. 3, September 1996, pp. 438 - 479.

Clyde Ruby and Gary T. Levens. *Safely Creating Correct Subclasses without Seeing Superclass Code*. In OOP-SLA '00 Conference Proceedings.

Derek Rayside and Gerard T. Campbell. *An Aristotelian Understanding of Object-Oriented Programming*. OOP-SLA '00 Conference Proceedings.

International Standard, ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994. ISBN: 0-8053-5340-2

Robert C. Martin. *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice Hall, Englewood Cliffs, New Jersey, 1994. ISBN: 0-13-203837-4

Grady Booch, et. al. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, 1998. ISBN: 0-201-57168-4

NOTES

PART III: IMPLEMENTING POLYMORPHIC BEHAVIOR

CHAPTER 14



Family of the Old Country

Ad Hoc Polymorphism: Operator Overloading

LEARNING OBJECTIVES

- Define the term Ad Hoc Polymorphism
- Explain how to achieve ad hoc polymorphic behavior through operator overloading
- Identify which C++ operators can be overloaded
- Demonstrate your ability to overload the following arithmetic operators: +, -, *, /
- Demonstrate your ability to overload the following relational operators: <, >, <=, >=
- Demonstrate your ability to overload the following equality operators: ==, !=
- Demonstrate your ability to overload the following unary operators: prefix ++, postfix ++, prefix -, postfix -
- Demonstrate your ability to overload the subscript operator: []
- Demonstrate your ability to overload iostream operators
- Explain when and how to use friend functions to implement operator overloading
- Explain when overloaded operator functions should be class members
- Explain why and when operator overloading is right for your design

INTRODUCTION

In C++ you can add meaning to certain language operators so they behave in expected ways when applied to your user-defined type objects. Adding meaning to operators in this fashion is referred to as operator overloading.

You have already seen many examples of operator overloading in action. Take the declaration and use of pointers as an example. To declare a pointer you use the `*` operator. To dereference a pointer you use the same operator, which is the same operator you use to multiply two number types. The compiler knows, by examining the context in which the operator is used, which version of the operator to call.

Overloaded operators provide an elegant way to manipulate user-defined type objects. The decision regarding which operators should be overloaded to manipulate a particular class of objects is a function of your design. This chapter will help you understand how to overload C++ operators and show you when it is appropriate to overload the different types of operators in the context of your design.

Although I will show you all of the operators that can be overloaded in C++ I will not show you an example of how to overload every single operator. Many operators can be grouped together, like the binary arithmetic operators. Knowing how to overload one in the group leads to an understanding of how to overload the others. Most of the operators can be treated in this fashion. Others will be left to the student as an exercise for further independent study.

Operator overloading is one of my favorite C++ language features. It is one of the “cool” things a C++ programmer knows how to do. When you get the hang of operator overloading and get used to thinking of when and how to incorporate overloaded operators in your class design, you will miss not being able to overload operators when programming in a language like Java. Overloaded operators, used in the right context, lead to cleaner, easier to read and understand code.

Ad Hoc Polymorphism: FUNCTION OVERLOADING

Ad hoc polymorphism is function overloading. When you declare and define several versions of a function the function name remains the same but the function signature changes, meaning the number of function parameters and the types of those parameters varies from one version of the overloaded function to another. The decision regarding which version of the function to call is made by the compiler at compile time using the number and types of arguments passed in the function call. If the compiler finds a function matching the signature of the function call then it uses that version of the function. Failure to find a match results in a compiler error.

Operator overloading is function overloading and therefore works the same way. If an operator is applied to a user-defined type object, and that operator is overloaded to work in the context of that particular user-defined type, then the overloaded version of the operator is used. Failure on the part of the compiler to find an overloaded version of the operator results in a compiler error.

THE GOAL OF OPERATOR OVERLOADING

The goal of operator overloading is to give meaning to language operators in the context of user-defined data types. That is it! Understanding this goal will help guide you in implementing a particular overloaded operator for each of your user-defined data types.

OVERLOADABLE OPERATORS

Table 14.1 lists the overloadable operators. The right hand column gives an example prototype for one of the operators in the group for a user-defined type named `Foo`.

Operator Type	Operators	Example Prototype For Class Foo
Arithmetic (Additive & Multiplicative)	+, -, *, /, %	Foo& operator+(Foo& lhs, Foo& rhs);
Assignment	=	Foo& operator=(Foo& rhs);
Compound Assignment	*=, /=, %=, +=, -=, >>=, <<=, &=, ^=, =	Foo& operator+=(Foo& rhs);
Relational	<, >, <=, >=,	bool operator<(Foo& rhs);
Logical	!, &&,	bool operator!(Foo& rhs);
Equality	==, !=	bool operator==(Foo& rhs);
Shift	<<, >>	Foo& operator<<(int shift_by);
Bitwise	&, , ^	Foo& operator&(int mask_value);
Comma	,	Foo& operator,(Foo& rhs);
Pointer-to-Member	->, ->*	Foo* operator->();
IOStream Insertion & Ex- traction	<<, >>	iostream& operator<<(iostream& out, Foo&); iostream& operator>>(iostream& in, Foo&);
Increment & Decrement	++, --	Foo& operator++(); //postfix Foo& operator++(int); //prefix Foo& operator--(); //postfix Foo& operator--(int); //prefix
Function Call	()	Foo& operator()(int val1, int val2);
Subscript	[]	Foo& operator[](unsigned int);
Unary	*, &, +, -, !, ~	Foo& operator+();
Allocation & Deallocation	new, new[] delete, delete[]	Not covered in this chapter.

Table 14-1: Overloadable Operators

OVERLOADING OPERATORS

An overloaded operator is a function taking the form `operatorX()` where X is the operator being overloaded. For instance, `operator=()` is the function name of the assignment operator.

Overloaded operator function parameter lists vary in number and type based on three criteria: 1) where the function is declared, 2) on what types it is intended to operate, and 3) what operator is being overloaded. For example, an overloaded assignment operator declaration appearing in a Foo class declaration might take the following form:

```
Foo& operator=(Foo& rhs)
```

In this example the assignment operator is overloaded to operate on Foo objects taking a Foo object as an argument to the function. Assuming there are two Foo objects named f1 and f2, the overloaded assignment operator can be

called on f1 in the following fashion:

```
f1 = f2;
```

The f1 object appears on the left side of the assignment operator and the f2 object appears on the right hand side. Here is an alternative way to call the overloaded operator:

```
f1.operator=(f2);
```

Calling the overloaded operator by its function name is referred to as an explicit call. The function call is made via the left hand side object, f1, taking the f2 object as an argument.

The overloaded assignment operator is an example of an operator that must be declared as a non-static class member function. However, some operators can be declared globally, meaning they do not belong to any particular class. An overloaded operator of this type will usually take two arguments.

The following sections discuss the implementation of different groups of operators in more detail.

OVERLOADING IOSTREAM INSERTION AND EXTRACTION OPERATORS: <<, >>

The iostream insertion and extraction operators, << and >>, can be overloaded to properly handle user-defined data type objects. To illustrate iostream operator overloading let us revisit the Person class you have seen used in the past several chapters.

Up until now, to print information about a Person it was necessary to use an accessor function on a Person object in conjunction with iostream insertion as shown in the following lines of code:

```
Person P1("Richard", "Warren", "Miller");
cout<<P1.getFullName()<<endl;
```

In this example, a Person object named P1 is instantiated with first, middle, and last names. The getFullName() function is then called on the P1 object. The getFullName() returns a char* and the iostream insertion operator << knows how to handle char* types. But what would happen if you tried to insert only the P1 object into the output stream like so:

```
cout<<P1<<endl; //Oops! Compiler error
```

You will get a compiler error because the insertion operator has not been programmed to handle Person objects. To change this situation you will need to overload the insertion operator and define what it means to insert a Person object into the output stream. In this section I will also show you how to overload both the insertion and extraction operators to both print Person objects to the screen, save them to a file on disk, and extract them from a file.

The modified Person class declaration appears in example 14.1. Lines 25, 26, and 27 contain the declarations for the overloaded insertion and extraction operators. Each of these functions are declared to be friends of the Person class. Declaring a non-member function to be a friend of a class gives the function special access to the class's private attributes. This may or may not be desirable in the context of your design. The alternative to granting friend status to non-member functions is to provide adequate class accessor and mutator functions through which the necessary class attributes can be manipulated.

Study each of the three overloaded operator functions appearing on lines 25, 26, and 27 of the Person class declaration. The first operator is an overloaded insertion for an ofstream. This version of the insertion operator will be invoked by the compiler when inserting into an output file stream object. The insertion operator appearing on line 27 is declared to operate on ostream objects. Knowing a little about the iostream inheritance hierarchy will help decipher the behavior of each of these operators. The ofstream and ifstream classes represent output file and input file streams respectively. The ostream class represents an output stream. The inheritance hierarchy is shown in figure 14-1.

Example 14.2 gives the source code for the implementation of the three versions of the overloaded stream operators. The code in example 14.2 was added to the person.cpp file but since the overloaded stream insertion and extraction operators are not member functions of the Person class they could appear in any implementation file. However,

14.1 modified person.h

```

1  #ifndef __Person_H
2  #define __Person_H
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  class Person{
8  public:
9      Person(char* _f_name = "John", char* _m_name = "M",
10             char* _l_name = "Doe", char _sex = 'M', int _age = 18);
11      ~Person();
12      Person(Person& person);
13      Person&operator=(Person& rhs);
14      void    setFirstName(char* f_name);
15      void    setMiddleName(char* m_name);
16      void    setLastName(char* l_name);
17      void    setAge(int age);
18      void    setSex(char sex);
19      char*   getFullName();
20      char*   getFirstName();
21      char*   getMiddleName();
22      char*   getLastName();
23      char    getSex();
24      int     getAge();
25      friend ostream& operator<<(ostream& out, Person& p);
26      friend ifstream& operator>>(ifstream& in, Person& p);
27      friend ostream& operator<<(ostream& out, Person& p);
28  private:
29      char* f_name;
30      char* l_name;
31      char* m_name;
32      char sex;
33      int age;
34      char* full_name;
35      bool name_changed;
36  };
37  #endif

```

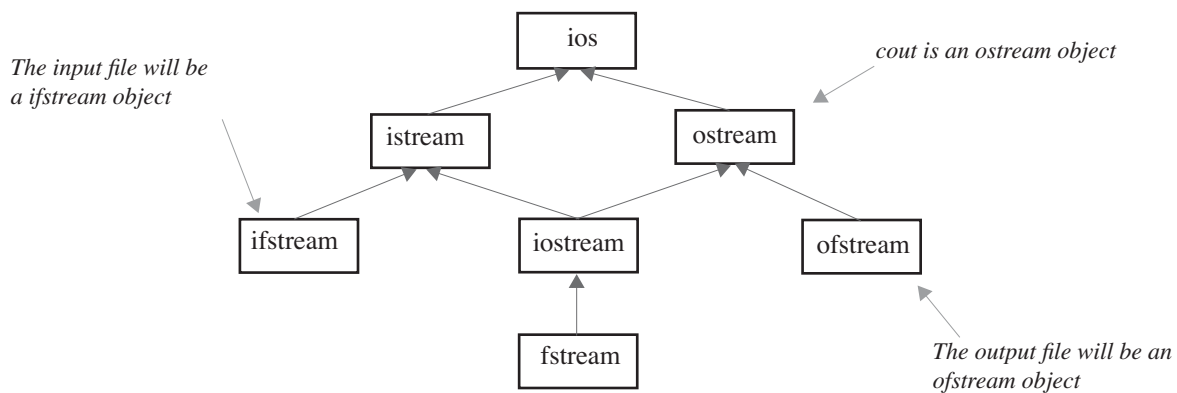


Figure 14-1: I/O Stream Class Hierarchy

since they are overloaded to handle `Person` objects it makes sense to place them in the `person.cpp` file where they can be readily found.

Referring to example 14.2 let us talk about the implementation of each overloaded operator. The definition of the first overloaded insertion operator begins on line 5. This version of the insertion operator is expecting an output file object. Since it has been declared as a friend function of the `Person` class the private attributes of `Person` objects can be directly accessed. The statement on line 6 does the hard work. It inserts various attributes of the `Person` object `p` into the `ofstream` object parameter `out`. The insertion of each `Person` attribute is followed by the insertion of a vertical bar character. The vertical bar will act as a delimiter between variable length fields in the file and will be used when the file is read by the overloaded extraction operator.

The overloaded extraction operator definition begins on line 10. It is overloaded to extract from an input file object into a `Person` object. A character array named `temp` in declared on line 11. This array will be used to temporarily hold the strings read from the input file before being copied into their corresponding `Person` object attribute

```

1  /*****
2  Overloaded IOStream Friend Functions
3  *****/
4
5  ostream& operator<<(ofstream& out, Person& p){
6      out<<p.f_name<<"|"<<p.m_name<<"|"<<p.l_name<<"|"<<p.sex<<"|"<<p.age<<"|"<<endl;
7      return out;
8  }
9
10 ifstream& operator>>(ifstream& in, Person& p){
11     char temp[31];
12     temp[0] = '\0';
13     in.getline(temp, 30, '|');
14
15     if(temp[0]){
16         delete[] p.f_name;
17         p.f_name = new char[strlen(temp)+1];
18         strcpy(p.f_name, temp);
19
20         in.getline(temp, 30, '|');
21         delete[] p.m_name;
22         p.m_name = new char[strlen(temp)+1];
23         strcpy(p.m_name, temp);
24
25         in.getline(temp, 30, '|');
26         delete[] p.l_name;
27         p.l_name = new char[strlen(temp)+1];
28         strcpy(p.l_name, temp);
29
30         in.getline(temp, 30, '|');
31         p.sex = temp[0];
32
33         in.getline(temp, 30, '|');
34         p.age = atoi(temp);
35
36         p.name_changed = true;
37         return in;
38     }else return in;
39 }
40 }
41
42 ostream& operator<<(ostream& out, Person& p){
43     out<<p.f_name<<" " <<p.m_name<<" " <<p.l_name<<" " <<p.sex<<" " <<p.age<<endl;
44     return out;
45 }

```

14.2 overloaded stream
operator implementation

arrays. On line 12 the first element in the temp array is set to the null character. On line 13 the first series of characters are read from the input file up to 30 characters or until the vertical bar delimiter is read. Line 15 checks to see if the first character of the temp array has changed, if so, the rest of the information from the current line of the input file is read and the Person object attributes are set accordingly.

The second overloaded insertion operator definition begins on line 42. This version of the insertion operator differs from the first in that the designated Person object's attributes will be inserted into the stream without the vertical bar delimiters.

Example 14.3 gives the source code for the main() function used to test these overloaded stream operators. Figure 14-2 shows the results of running example 14.3 and entering information for two people. The program begins by

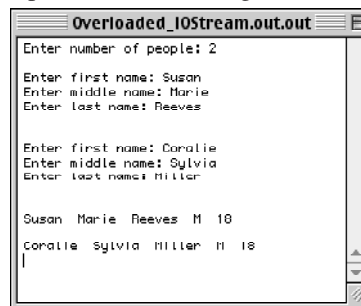


Figure 14-2: Results of Running Example 14.3

```

1  #include <iostream>
2  using namespace std;
3  #include "person.h"
4
5  int main(){
6
7      /*****variables*****/
8      int number = 0;
9      char fname[30];
10     char mname[30];
11     char lname[30];
12     Person **people;
13
14     /*****get number of people to enter *****/
15     cout<<"Enter number of people: ";
16     cin>>number;
17     cout<<endl;
18
19     /*****create people array*****/
20     people = new Person*[number];
21
22     /*****enter people names*****/
23     for(int i=0; i<number; i++){
24
25         cout<<"Enter first name: ";
26         cin>>fname;
27         cout<<"Enter middle name: ";
28         cin>>mname;
29         cout<<"Enter last name: ";
30         cin>>lname;
31         cout<<endl<<endl;
32
33         people[i] = new Person(fname, mname, lname);
34     }
35
36     /*****create file to save people *****/
37     ofstream outfile("people.txt");
38
39     for(int i=0; i<number; i++)
40         outfile<<(*people[i]);
41
42     outfile.close();
43
44     /****open file and read people info*****/
45     ifstream infile("people.txt");
46
47     for(int i=0; i<number; i++)
48         infile>>(*people[i]);
49
50     infile.close();
51
52     /*****print people to screen*****/
53     for(int i=0; i<number; i++)
54         cout<<(*people[i]);
55
56     /*****delete pointers*****/
57     for(int i=0; i<number; i++)
58         delete people[i];
59
60     delete[] people;
61
62     return 0;
63 }
64
65

```

prompting the user to enter the number of people they wish to enter. The people array is dynamically created on line 20 and on line 23 a for loop repeatedly prompts the user to enter the first, middle, and last name for each person. When all the Person objects have been created a file named people.txt is created and opened and each person object is inserted into the file. After the insertions the contents of the people.txt file looks like this:

```

Susan | Marie | Reeves | M | 18 |
Coralie | Sylvia | Miller | M | 18 |

```

When the insertions are complete the output file is closed.

On line 45 an input file named `people.txt` is opened and its contents extracted into the two `Person` objects contained in the `people` array. (The `people` array is an array of pointers to `Person`, hence the need to dereference each `Person` pointer to yield the `Person` object.) Each `Person` object is then inserted into the `cout` object yielding the output shown in figure 14-2. The `main()` function wraps up by deleting each `Person` pointer and the `people` array.

OVERLOADING THE ASSIGNMENT OPERATOR: =

The assignment operator works on two existing objects. Given two `Foo` objects `f1` and `f2` the `f1` object can be assigned the value of the `f2` object in the following manner:

```
f1 = f2;
```

To do this the assignment operator need not be overloaded if `Foo` objects are relatively simple and contain no pointers as data members. The compiler will provide a default assignment operator that will perform a member bit-wise copy. To demonstrate this observe the `Foo` header file shown in example 14.4

```

1  #ifndef _FOO_H
2  #define _FOO_H
3  #include <iostream>
4  using namespace std;
5
6  class Foo{
7  public:
8      Foo(int _i = 0):i(_i){}
9      friend ostream& operator<<(ostream& out, Foo rhs);
10 private:
11     int i;
12 };
13
14 ostream& operator<<(ostream& out, Foo rhs){
15     out<<rhs.i<<endl;
16     return out;
17 }
18 #endif

```

14.4 foo.h

This `Foo` class contains only one integer data member named `i`. The constructor has been inlined and the `ostream` insertion operator overloaded to make inserting `Foo` objects easier. However, notice that no assignment operator has been declared. Example 14.5 gives a `main()` function that creates and uses two `Foo` objects.

```

1  #include <iostream>
2  using namespace std;
3  #include "foo.h"
4
5  int main(){
6      Foo f1(1), f2(2);
7      cout<<f1<<f2<<endl;
8      f1 = f2;
9      cout<<f1<<f2<<endl;
10     return 0;
11 }

```

14.5 main.cpp

The value of `f1` is set to 1 and the value of `f2` is set to 2. Their values are printed to the console and then on line 8 the default assignment operator is used to perform a bitwise assignment resulting in `f1::i` being set to the value of `f2::i`. Figure 14-3 shows the results of running this program.

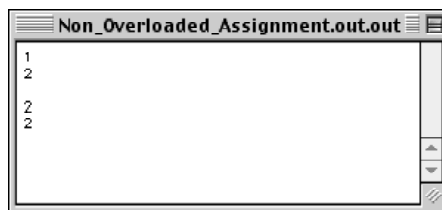


Figure 14-3: Results of Running Example 14.5

SHALLOW COPY vs. DEEP COPY

The default assignment operator will always perform a bitwise member copy of the right hand side object's data members to the left hand side object's data members. This is also referred to as a shallow copy. If the class contains only simple data members like the Foo class above this is acceptable behavior. However, in the real world of complex objects, a simple shallow copy is not enough.

The danger of performing a shallow copy on complex objects is illustrated in figures 14-4a and 14-4b.

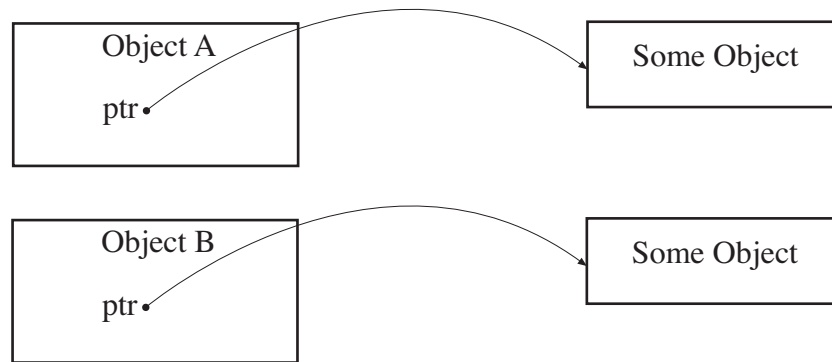


Figure 14-4a: Before Shallow Copy of Complex Objects

Oops! This object is still in memory!!

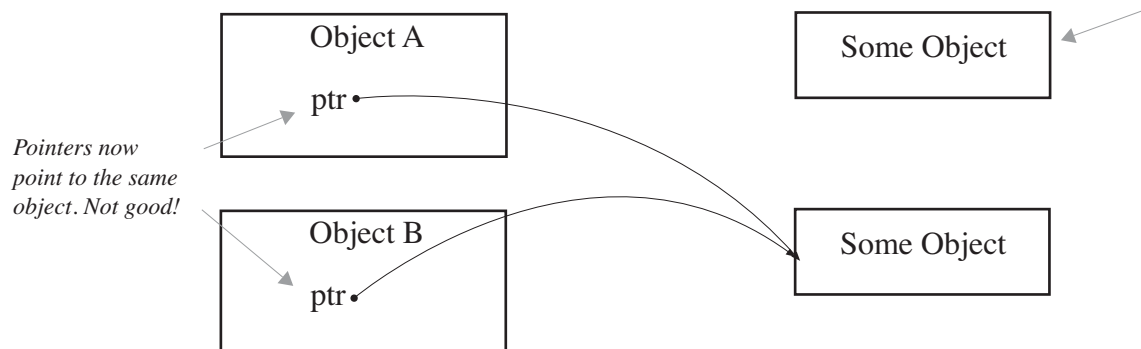


Figure 14-4b: After Shallow Copy of Complex Objects

Figure 14-4a shows two complex objects, A and B, before a shallow copy. Object A contains a pointer that points to some object. Object B, being of the same class, also contains a pointer but it points to a whole different object.

Along comes the shallow copy: `A = B`; Object B's pointer contents will be assigned to Object A's pointer, meaning Object A's pointer will now point to what Object B's pointer points to. All without deleting Object A's pointer. Memory leak!

When you have a complex class type you must overload the assignment operator to ensure the proper handling of class data members during assignment. To illustrate some of the complexities involved with proper assignment operator overloading I will once again draw your attention to the Person class. In fact, the assignment operator was overloaded for the Person class in chapter 11. You can refer to example 14.1 to see the assignment operator declaration as it appears in the Person class. Example 14.6 gives only the code for the assignment operator as it appears in the person.cpp file. The important thing to keep in mind when overloading the assignment operator is that it is working with two existing objects. When the left hand side object has dynamically allocated resources you must do any housekeeping required to ensure those resources are properly treated. In the case of Person class's overloaded assignment operator, it must delete the existing dynamically allocated character arrays and create new ones based on the length of the character arrays of the right hand side object. The characters from the right hand side Person object are then copied into the left hand side Person object's character arrays. You may notice that the Person class's assignment operator only copied the name information. You will have a chance to expand its capabilities in a skill building exercise at the end of the chapter.


```

1   Person& Person::operator=( Person& rhs ){
2       delete[] f_name;
3       f_name = new char[strlen(rhs.f_name)+1];
4       strcpy(f_name, rhs.f_name);
5
6       delete[] m_name;
7       m_name = new char[strlen(rhs.m_name)+1];
8       strcpy(m_name, rhs.m_name);
9
10      delete[] l_name;
11      l_name = new char[strlen(rhs.l_name)+1];
12      strcpy(l_name, rhs.l_name);
13
14      name_changed = true;
15
16      return *this;
17  }

```

14.6 overloaded assignment
operator implementation

OVERLOADING RELATIONAL OPERATORS: <, >, <=, >=

The relational operators are used to compare one object to another. The trick to implementing a relational operator is to pick one or more attributes from the set of class data members upon which the comparison will be made. Essentially, you as the programmer have to ask: “What does it mean for one object to be less than or greater than another?”

The relational operator function can be implemented as a class member function or as a non-member function. The member function version takes one argument and the non-member function version takes two arguments. The non-member version must either be declared to be a friend of the particular class for which it is being implemented or the class must supply adequate accessor and mutator functions. Example 14.7 gives the class declaration of the Person class yet again extended to include declarations for each of the relational operators.

The attribute selected for comparison was age. Example 14.8 gives the code showing the implementation of all four relational operators. These operators are implemented as Person class member functions and appear in the person.cpp file.

```

1   /*****
2       Overloaded Relational Operators
3       *****/
4   bool Person::operator<(Person& rhs){return (age < rhs.age);}
5
6   bool Person::operator>(Person& rhs){return (age > rhs.age);}
7
8   bool Person::operator<=(Person& rhs){return (age <= rhs.age);}
9
10  bool Person::operator>=(Person& rhs){return (age >= rhs.age);}

```

14.8 overloaded relational
operator implementation

Example 14.9 shows a main() function putting one of the overloaded relational operators to the test.

```

1   #include <iostream>
2   using namespace std;
3   #include "person.h"
4
5   int main() {
6
7       Person Rick("Rick", "Warren", "Miller", 'M', 41);
8       Person Bob("Bob", "J", "Jones", 'M', 25);
9
10      cout<<Rick<<endl;
11      cout<<Bob<<endl;
12
13      if(Bob < Rick)
14          cout<<Bob<<" is younger than "<<Rick<<endl;
15      else
16          cout<<Rick<<" is younger than "<<Bob<<endl;
17
18      return 0;
19  }

```

14.9 main.cpp

And Figure 14-5 shows the results of running example 14.9.

```

1  #ifndef __Person_H
2  #define __Person_H
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  class Person{
8      public:
9          Person(char* _f_name = "John", char* _m_name = "M",
10             char* _l_name = "Doe", char _sex = 'M', int _age = 18);
11          ~Person();
12          Person(Person& person);
13          Person&operator=(Person& rhs);
14          void setFirstName(char* f_name);
15          void setMiddleName(char* m_name);
16          void setLastName(char* l_name);
17          void setAge(int age);
18          void setSex(char sex);
19          char* getFullName();
20          char* getFirstName();
21          char* getMiddleName();
22          char* getLastName();
23          char getSex();
24          int getAge();
25          friend ostream& operator<<(ostream& out, Person& p);
26          friend ifstream& operator>>(ifstream& in, Person& p);
27          friend ostream& operator<<(ostream& out, Person& p);
28          bool operator<(Person& rhs);
29          bool operator>(Person& rhs);
30          bool operator<=(Person& rhs);
31          bool operator>=(Person& rhs);
32      private:
33          char* f_name;
34          char* l_name;
35          char* m_name;
36          char sex;
37          int age;
38          char* full_name;
39          bool name_changed;
40  };
41  #endif

```

14.7 person.h

```

Overloaded_Relational.out.out
Rick Warren Miller M 41
Bob J Jones M 25
Bob J Jones M 25
is younger than Rick Warren Miller M 41

```

Figure 14-5: Results of Running Example 14.9

Notice on line 13 of Example 14.9 how the two Person objects Rick and Bob are used in the if statement. Overloading relational operators, in situations where it makes sense to do so, results in cleaner, more natural code. Before moving on to the next section I should speak a little more to example 14.8. Notice in each of the relational operator functions how they all boil down to the use of a relational operator on a fundamental data type. The result of the comparison is simply returned.

OVERLOADING EQUALITY OPERATORS: ==, !=

The equality operators are used to compare the equality of two objects. Like the relational operators, the concept of equality is left to the discretion of the programmer. Example 14.10 gives the Person class extended to include both equality operators. Example 10.11 gives the source code for the implementation of both operators. Notice how the implementation of these operators differ from the relational operators. I have made the choice here to compare the this pointer of the left hand side object to the address of the right hand side object. If the addresses are the same, they must be the same object, otherwise they are different objects. I could just as easily have chosen a class attribute, such

14.10 extended Person class

```

1  #ifndef __Person_H
2  #define __Person_H
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  class Person{
8      public:
9          Person(char* _f_name = "John", char* _m_name = "M",
10             char* _l_name = "Doe", char _sex = 'M', int _age = 18);
11          ~Person();
12          Person(Person& person);
13          Person&operator=(Person& rhs);
14          void    setFirstName(char* f_name);
15          void    setMiddleName(char* m_name);
16          void    setLastName(char* l_name);
17          void    setAge(int age);
18          void    setSex(char sex);
19          char*   getFullName();
20          char*   getFirstName();
21          char*   getMiddleName();
22          char*   getLastName();
23          char    getSex();
24          int    getAge();
25          friend ostream& operator<<(ostream& out, Person& p);
26          friend ifstream& operator>>(ifstream& in, Person& p);
27          friend ostream& operator<<(ostream& out, Person& p);
28          bool    operator<(Person& rhs);
29          bool    operator>(Person& rhs);
30          bool    operator<=(Person& rhs);
31          bool    operator>=(Person& rhs);
32          bool    operator==(Person& rhs);
33          bool    operator!=(Person& rhs);
34      private:
35          char*   f_name;
36          char*   l_name;
37          char*   m_name;
38          char    sex;
39          int    age;
40          char*   full_name;
41          bool    name_changed;
42  };
43  #endif

```

14.11 equality operator implementation

```

1  bool Person::operator==(Person& rhs){
2      return (this == (&rhs));
3  }
4
5  bool Person::operator!=(Person& rhs){
6      return (this != (&rhs));
7  }

```

as age, as I did for the relational operators, to make the comparison. Example 14.12 shows the equality operators in action.

14.12 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "person.h"
4
5  int main(){
6
7      Person Rick("Rick", "Warren", "Miller", 'M', 41);
8      Person Bob("Bob", "J", "Jones", 'M', 25);
9
10     if(Rick == Rick)
11         cout<<"Same person!"<<endl;
12     else cout<<"Different person!"<<endl;
13
14     if(Rick != Bob)
15         cout<<"Different person!"<<endl;
16     else cout<<"Same person!"<<endl;
17     return 0;
18 }

```

Figure 14-6 shows the results of running example 14.12.



Figure 14-6: Results of Running Example 14.12

OVERLOADING ARITHMETIC OPERATORS: +, -, *, /, %

The arithmetic operators are overloaded as non-member functions. They can be declared to be friends of the class of objects upon which they operate. To demonstrate overloading arithmetic operators I will revert to my favorite class named `Foo`. Example 14.13 gives the code for the `Foo` class declaration.

```

1  #ifndef _FOO_H
2  #define _FOO_H
3
4  class Foo{
5  public:
6      Foo(int ival = 0);
7      ~Foo();
8      int getI();
9      Foo& operator=(int _i);
10     friend int operator+(Foo& lhs, Foo& rhs);
11     friend int operator-(Foo& lhs, Foo& rhs);
12 private:
13     int i;
14 };
15 #endif

```

14.13 *foo.h*

In this example two arithmetic operators are overloaded: `operator+()` and `operator-()`. Notice that each operator function takes two arguments, both of type `Foo`. So, what is being said here is that two `Foo` objects can be added together and that one `Foo` object can be subtracted from another. Each function returns an integer value that results from the operation. Example 14.14 gives the code for the `Foo.cpp` file showing the implementation of each of these operators.

```

1  #include "foo.h"
2
3  Foo::Foo(int ival):i(ival){}
4
5  Foo::~~Foo(){};
6
7  int Foo::getI(){return i;}
8
9  Foo& Foo::operator=(int _i){
10     i = _i;
11     return *this;
12 }
13
14 /*** Friend +, - functions ***/
15 int operator+(Foo& lhs, Foo& rhs){
16     return lhs.i + rhs.i;
17 }
18
19 int operator-(Foo& lhs, Foo& rhs){
20     return lhs.i - rhs.i;
21 }

```

14.14 *foo.cpp*

Because of their friend status to class `Foo`, the overloaded operator functions enjoy free access to a `Foo` object's private attributes. Example 14.15 gives a `main()` function showing the overloaded arithmetic operators in use on some `Foo` objects. Figure 14-7 shows the results of running example 14.15.

14.15 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "foo.h"
4
5  int main(){
6
7      Foo f1(1), f2(2), f3(3);
8
9      cout<<f1.getI()<<endl;
10     cout<<f2.getI()<<endl;
11     cout<<f3.getI()<<endl;
12
13     f1 = f2 + f3;
14     f2 = f1 + f3;
15     f3 = f2 + f1;
16
17     cout<<f1.getI()<<endl;
18     cout<<f2.getI()<<endl;
19     cout<<f3.getI()<<endl;
20
21     f1 = f1 - f1;
22
23     cout<<f1.getI()<<endl;
24
25     return 0;
26 }

```

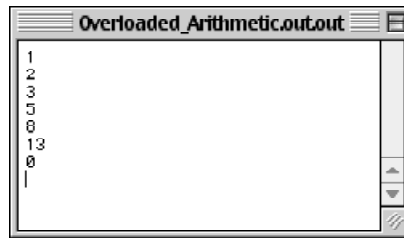


Figure 14-7: Results of Running Example 14.15

A FEW WORDS ABOUT ERROR CHECKING

Example 14.14 demonstrates a very simplistic implementation of two overloaded arithmetic operators. In reality, you should check to ensure the result of the operation does not exceed the capacity of the returned type without taking appropriate action.

OVERLOADING THE SUBSCRIPT OPERATOR: []

In cases where your class contains an array, you can make interaction with the array more natural by overloading the array subscript operator. The subscript operator must be declared as a class member function.

Example 14.16 gives the declaration for a class named `DynamicArray`. The `DynamicArray` class implements a dynamically allocated array of integers. It will automatically expand as necessary to accommodate insertions past its current size.

The declaration of the overloaded subscript operator appears on line 8. This version of the operator declares one parameter, an unsigned integer named `i`. The parameter will actually be the value that appears between the brackets when the implicit call to the subscript operator is made. In the case of `DynamicArray` objects, this call will look like what you have seen with ordinary arrays. For example, given a `DynamicArray` object named `d1`, the integer value 23 can be inserted into `d1`'s array position 0 with the following code:

```
d1[0] = 23;
```

```

1  #ifndef _DYNAMIC_ARRAY_H
2  #define _DYNAMIC_ARRAY_H
3
4  class DynamicArray{
5  public:
6      DynamicArray(int _size = 5);
7      virtual ~DynamicArray();
8      int& operator[](unsigned i);
9      int getSize();
10 private:
11     int* its_array;
12     int size;
13 };
14 #endif

```

14.16 *dynamicarray.h*

Example 14.17 shows the `dynamicarray.cpp` file.

```

1  #include "dynamicarray.h"
2
3  DynamicArray::DynamicArray(int _size):size(_size){
4      its_array = new int[_size];
5      for(int i=0; i<size; i++)
6          its_array[i] = 0;
7  }
8
9  DynamicArray::~DynamicArray(){
10     delete[] its_array;
11 }
12
13 int& DynamicArray::operator[](unsigned i){
14     if(i >= (size)){
15         int newsize = size+10;
16         int* temp = new int[newsize];
17         for(int j = 0; j<size; j++){
18             temp[j] = its_array[j];
19         }
20         delete[] its_array;
21         its_array = new int[newsize];
22         for(int j = 0; j<size; j++){
23             its_array[j] = temp[j];
24         }
25
26         for(int j=size; j<newsize; j++){
27             its_array[j] = 0;
28         }
29         delete[] temp;
30         size = newsize;
31
32         return its_array[i];
33     } else return its_array[i];
34 }
35
36 int DynamicArray::getSize(){ return size;}

```

14.17 *dynamicarray.cpp*

Note the implementation of the subscript operator and the use of the parameter `i`. The parameter is not the value being inserted into the array. The parameter is used as the index into the array. If you take away all the dynamic sizing code all the subscript operator really does is return a reference to the value located at the array position indicated by the parameter value. If the value of `i` is greater than or equal to the current size of the array the array is increased in size and then value is returned, other wise, the value is simply returned.

In this example the parameter is an integer value. However, the parameter could be anything. For instance, the parameter could be a string of characters used to locate entries in an associative array. Example 14.18 gives a `main()` function showing a `DynamicArray` object in action. Figure 14-8 shows the results of running example 14.18.

On line 6 of example 14.18 an instance of `DynamicArray` is instantiated with its default array size of 5. The for statement on line 8 inserts 6 integer values into the array. When the 6th integer is inserted the array resizes itself allowing the insertion.

Notice how the overloaded subscript operator makes handing `DynamicArray` objects easier. The code on line 13 shows how the subscript operator can be used on the right hand side of the assignment operator as well as on the left hand side.

14.18 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "dynamicarray.h"
4
5  int main(){
6      DynamicArray d1;
7
8      for(int i=0; i<6; i++){
9          d1[i] = i;
10     }
11
12     int j = d1[3];
13     cout<<"j: "<<j<<endl;
14
15     for(int i=0; i<d1.getSize(); i++){
16         cout<<d1[i]<<endl;
17     }
18 }
19
20 return 0;
21 }

```

```

Overloaded_Subscript.out.out
j: 3
0
1
2
3
4
5
0
0
0
0
0
0
0
0
0
0
0
0
0
|

```

Figure 14-8: Results of Running Example 14.18

OVERLOADING COMPOUND ASSIGNMENT OPERATORS: +=, -=, *=, ETC.

The important thing to remember about the compound operators is that they perform both a binary operation and an assignment all in one stroke. This gives a hint regarding their implementation. The Foo class will once again be used to demonstrate the implementation of a compound operator. Example 14.19 gives the declaration of class Foo with the operator+=() declared.

14.19 foo.h

```

1  #ifndef _FOO_H
2  #define _FOO_H
3
4  class Foo{
5      public:
6          Foo(int _i = 0);
7          void setI(int _i);
8          int getI();
9          Foo& operator=(Foo& rhs);
10         Foo& operator+=(Foo& rhs);
11     private:
12         int i;
13 };
14 #endif

```

The foo.cpp file is shown in example 14.20. The operator+=() definition begins on line 14 with the real work happening on line 15. The function ends by returning a reference to the current object. Example 14.21 gives a main() function showing the compound operator in action. Figure 14-9 shows the results of running 14.21.

```

1  #include "foo.h"
2
3  Foo::Foo(int _i):i(_i){}
4
5  void Foo::setI(int _i){ i = _i;}
6
7  int Foo::getI(){return i;}
8
9  Foo& Foo::operator=(Foo& rhs){
10     i = rhs.i;
11     return *this;
12 }
13
14 Foo& Foo::operator+=(Foo& rhs){
15     i = i+rhs.i;
16     return *this;
17 }

```

14.20 *foo.cpp*

```

1  #include <iostream>
2  using namespace std;
3  #include "foo.h"
4
5  int main(){
6
7      Foo f1(3), f2(4);
8
9      f1+=f2;
10     cout<<f1.getI()<<endl;
11     return 0;
12 }

```

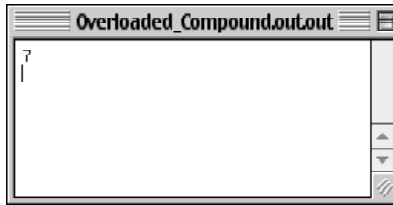
14.21 *main.cpp*

Figure 14-9: Results of Running Example 14.21

OVERLOADING INCREMENT & DECREMENT OPERATORS: ++, --

There are two forms of increment and decrement operator: the postfix form and the prefix form. The function signature of each form changes depending on whether you declare it as a member or non-member function. Example 14.22 gives a `Foo` class with both increment and decrement operators overloaded as member functions in both postfix and prefix form.

The difference between the prefix and postfix form of each operator lies in the number of parameters each takes. The prefix form takes no parameters while the postfix form takes one parameter, which is a dummy integer placeholder so the compiler can distinguish between the prefix and postfix form.

Example 14.23 gives the source code for the `foo.cpp` file containing the implementation of these operators. Example 14.24 shows a `main()` function putting all four operators to use, including two overloaded arithmetic operators. Figure 14-10 shows the results of running example 14.24.

14.22 *foo.h*

```

1  #ifndef _FOO_H
2  #define _FOO_H
3
4  class Foo{
5  public:
6      Foo(int ival = 0);
7      ~Foo();
8      int getI();
9      Foo& operator=(int _i);
10     Foo& operator++(); //prefix increment
11     Foo& operator++(int); //postfix increment
12     Foo& operator--(); //prefix decrement
13     Foo& operator--(int); //postfix decrement
14     friend int operator+(Foo& lhs, Foo& rhs);
15     friend int operator-(Foo& lhs, Foo& rhs);
16 private:
17     int i;
18 };
19 #endif

```

14.23 *foo.cpp*

```

1  #include "foo.h"
2
3  Foo::Foo(int ival):i(ival){}
4
5  Foo::~~Foo(){};
6
7  int Foo::getI(){return i;}
8
9  Foo& Foo::operator=(int _i){
10     i = _i;
11     return *this;
12 }
13
14 Foo& Foo::operator++(){
15     ++i;
16     return *this;
17 }
18
19 Foo& Foo::operator++(int){
20     i++;
21     return *this;
22 }
23
24 Foo& Foo::operator--(){
25     --i;
26     return *this;
27 }
28
29 Foo& Foo::operator--(int){
30     i--;
31     return *this;
32 }
33
34 /*** Friend +, - functions ***/
35 int operator+(Foo& lhs, Foo& rhs){
36     return lhs.i + rhs.i;
37 }
38
39 int operator-(Foo& lhs, Foo& rhs){
40     return lhs.i - rhs.i;
41 }

```

14.24 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "foo.h"
4
5  int main(){
6      Foo f1(1), f2(2), f3(3);
7
8      cout<<f1.getI()<<" ";
9      cout<<f2.getI()<<" ";
10     cout<<f3.getI()<<endl;
11
12     f1 = f2 + f3;
13     f2 = f1 + f3;
14     f3 = f2++ + f1++;
15
16     cout<<f1.getI()<<" ";
17     cout<<f2.getI()<<" ";
18     cout<<f3.getI()<<endl;
19
20     f1 = f1 - f1;
21
22     cout<<f1.getI()<<endl;
23
24     for(int i=0; i<5; i++){
25         cout<<f1.getI()<<" ";
26         f1++;
27     }
28     cout<<endl;
29
30     for(int i=0; i<5; i++){
31         cout<<f1.getI()<<" ";
32         ++f1;
33     }
34     cout<<endl;
35
36     for(int i=0; i<5; i++){
37         cout<<f1.getI()<<" ";
38         --f1;
39     }
40
41     return 0;
42 }

```

```

Overloaded_Increment.out.out
1, 2, 3
6, 9, 15
0
0, 1, 2, 3, 4,
5, 6, 7, 8, 9,
10, 9, 8, 7, 6, |

```

Figure 14-10: Results of Running Example 14.24

OVERLOADING VARIOUS OTHER OPERATORS: (), +, -, <<, ->, ETC.

This section will give you a hint as to how to overload various other operators that were not covered in previous sections. In your programming career you will be hard pressed to come up with a good reason to overload most of these operators.

The trusty Foo class will once again be used as a test class. Example 14.25 gives the function declaration with all the various operators declared. I will talk a little about each one.

Referring to example 14.25, lines 9 and 10 declare the assignment and compound addition/assignment operators which you have seen before. The comments on lines 12 through 28 indicate the other operators being overloaded.

Example 14.26 gives the foo.cpp file with the implementations of these functions. Example 14.27 gives a main()

```

1      #ifndef _FOO_H
2      #define _FOO_H
3
4      class Foo{
5      public:
6          Foo(int _i = 0);
7          void setI(int _i);
8          int getI();
9          Foo& operator=(Foo& rhs);
10         Foo& operator+=(Foo& rhs);
11         Foo& operator<<(int shift_by);           //left shift
12         Foo& operator&(unsigned mask_value);   //bitwise and
13         Foo& operator()(int val1, int val2);   //function
14         Foo& operator+();                       //unary positive
15         Foo& operator-();                       //unary negate
16         Foo* operator->();                      //member of
17         Foo& operator,(Foo& rhs);             //comma
18         Foo& operator,(int);                 //comma
19     private:
20         int i;
21     };
22
23     Foo& operator+(Foo& lhs, Foo& rhs);
24
25     #endif

```

14.25 foo.h

function showing these operators in action. Figure 14-11 shows the results of running Example 14.27

```

1      #include <iostream>
2      using namespace std;
3      #include "foo.h"
4
5      int main(){
6
7          Foo f1(3), f2(4);
8
9          f1+=f2;
10         cout<<f1.getI()<<endl;
11
12         f1<<5;
13
14         cout<<f1.getI()<<endl;
15
16         f1&0xfab5;
17
18         cout<<f1.getI()<<endl;
19
20         f1 = f1(3, 4);
21
22         cout<<f1.getI()<<endl;
23
24         f1 = f1 + f2;
25         cout<<f1.getI()<<endl;
26
27         f1 = (+f1);
28
29         cout<<f1.getI()<<endl;
30
31         f1 = (-f1);
32
33
34         cout<<f1.getI()<<endl;
35
36         int i = f1->getI();
37
38         cout<<i<<endl;
39
40         f1,1,f2;
41
42         return 0;
43     }

```

14.27 main.cpp

14.26 foo.cpp

```

1  #include "foo.h"
2  #include <iostream>
3  using namespace std;
4
5  Foo::Foo(int _i):i(_i){
6
7  void Foo::setI(int _i){ i = _i;}
8
9  int Foo::getI(){return i;}
10
11 Foo& Foo::operator=(Foo& rhs){
12     i = rhs.i;
13     return *this;
14 }
15
16 Foo& Foo::operator+=(Foo& rhs){
17     i = i+rhs.i;
18     return *this;
19 }
20
21 Foo& Foo::operator<<(int shift_by){
22     i <<= shift_by;
23     return *this;
24 }
25
26 Foo& Foo::operator&(unsigned mask_value){
27     i &= mask_value;
28     return *this;
29 }
30
31 Foo& Foo::operator()(int val1, int val2){
32     i += (val1 + val2);
33     return *this;
34 }
35
36 Foo& Foo::operator+(){
37     i = (+i);
38     return *this;
39 }
40
41 Foo& Foo::operator-(){
42     i = (-i);
43     return *this;
44 }
45
46 Foo* Foo::operator->(){
47     return this;
48 }
49
50 Foo& Foo::operator,(Foo& rhs){
51     cout<<"Foo::operator, followed by Foo object"<<endl;
52     rhs; //eliminates compiler warning - rhs parameter not used
53     return *this;
54 }
55
56 Foo& Foo::operator,(int){
57     cout<<"Foo::operator, followed by int object"<<endl;
58     return *this;
59 }
60
61 /*****
62     Global additive operator
63     *****/
64
65 Foo& operator+(Foo& lhs, Foo& rhs){
66     lhs.setI(lhs.getI() + rhs.getI());
67     return lhs;
68 }

```

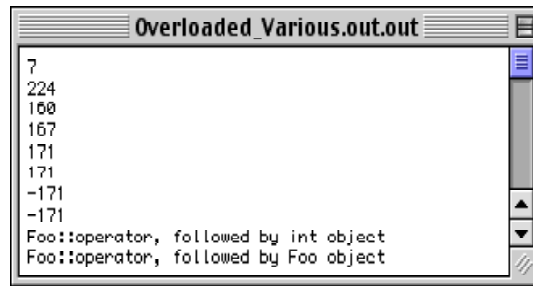


Figure 14-11: Results of Running Example 14.27

Most of the operators in this section are implemented in a straightforward fashion. However, the `operator()()`, `operator->()`, and `operator,()` functions need some discussion.

THE FUNCTION OPERATOR: `operator()()`

The `operator()()` is the function operator. It is generally overloaded to do things the subscript operator cannot do. In this example, referring to the code shown in example 14.26 on lines 31, 32, and 33, the function operator is overloaded to take two integer parameters and add their value to the Foo object's `i` attribute. The function operator can take as many parameters as necessary and its implementation is limited only by your imagination.

THE MEMBER OPERATOR: `operator->()`

The member operator implementation starts on line 46 of example 14.26. It is overloaded here to return a reference to the current Foo object. It could also be used to return references to other objects contained or pointed to by a Foo object.

THE COMMA OPERATOR: `operator,()` - A.K.A. THE SEQUENCING OPERATOR

Example 14.26 implements two versions of the comma operator. The version starting on line 50 implements the following Foo object version and the version starting on line 56 implements the following integer object version. The comma operator is very rarely overloaded!

VIRTUAL OVERLOADED OPERATORS

Overloaded operators can be overridden in derived classes like ordinary virtual functions. I will use two classes named Foo and Bar to demonstrate how to override the equality operator in a class deriving from Foo. Example 14.28 gives the Foo class declaration containing an overloaded equality operator.

```

1  #ifndef _FOO_H
2  #define _FOO_H
3
4  class Foo{
5      public:
6          Foo(int _i = 0);
7          virtual ~Foo();
8          virtual bool operator==(Foo& rhs);
9          int getI();
10     private:
11         int i;
12     };
13 #endif

```

14.28 foo.h

Notice the use of the keyword `virtual` in the declaration of the equality operator on line 8. Example 14.29 gives the declaration of class Bar. Bar extends Foo and declares two equality operators. The one declared on line 9 overloads the equality operator to take Bar objects. The version declared on line 10 overrides Foo's equality operator because it has the same function signature.

```

1  #ifndef _BAR_H
2  #define _BAR_H
3  #include "foo.h"
4
5  class Bar : public Foo{
6  public:
7      Bar(int _i = 0);
8      ~Bar();
9      virtual bool operator==(Bar& rhs);
10     virtual bool operator==(Foo& rhs);
11 private:
12     int i;
13 };
14 #endif

```

14.29 *bar.h*

Examples 14.30 and 14.31 give the source code for *foo.cpp* and *bar.cpp* respectively.

```

1  #include "foo.h"
2  #include <iostream>
3  using namespace std;
4
5  Foo::Foo(int _i):i(_i){}
6
7  Foo::~Foo(){}
8
9  bool Foo::operator==(Foo& rhs){
10     cout<<"Foo == called."<<endl;
11     return i == rhs.i;
12 }
13
14 int Foo::getI(){return i;}

```

14.30 *foo.cpp*

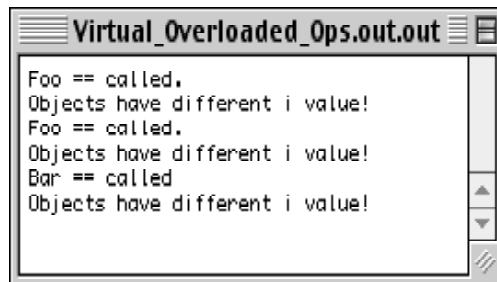
```

1  #include "bar.h"
2  #include <iostream>
3  using namespace std;
4
5  Bar::Bar(int _i):Foo(_i), i(_i){}
6
7  Bar::~Bar(){}
8
9  bool Bar::operator==(Bar& rhs){
10     cout<<"Bar == called"<<endl;
11     return i == rhs.i;
12 }
13
14 bool Bar::operator==(Foo& rhs){
15     cout<<"Bar (Foo) == called."<<endl;
16     return i == rhs.getI();
17 }

```

14.31 *bar.cpp*

Example 14.32 on the following page gives a *main()* function showing the equality operator being used on *Foo* and *Bar* objects. Figure 14-12 below shows the results of running example 14.32.



```

Virtual_Overloaded_Ops.out.out
Foo == called.
Objects have different i value!
Foo == called.
Objects have different i value!
Bar == called
Objects have different i value!

```

Figure 14-12: Results of Running Example 14.32

14.32 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "foo.h"
4  #include "bar.h"
5
6  int main(){
7      Foo f1(1), f2(2);
8      if(f1==f2){
9          cout<<"Objects have same i value!"<<endl;
10         }
11         else cout<<"Objects have different i value!"<<endl;
12
13         Foo* fptr = new Bar(5);
14
15         if(f1== (*fptr)){
16             cout<<"Objects have same i value!"<<endl;
17         }
18         else cout<<"Objects have different i value!"<<endl;
19
20         Bar b1(1), b2(2);
21
22         if(b1==b2){
23             cout<<"Objects have same i value!"<<endl;
24         }
25         else cout<<"Objects have different i value!"<<endl;
26
27         return 0;
28     }

```

SUMMARY

The goal of operator overloading is to give meaning to C++ operators in the context of user-defined data types. What an overloaded operator does to properly handle user-defined types is left to the discretion of the programmer. By overloading an operator for a particular class you answer the question “What does this operation mean in the context of this class?”

Most operators belong to a family or group of related operators. Knowing how to overload one leads to an understanding of how to overload the others in the group.

Most operators can be declared either as class member functions or non-member functions. Some of the operators, like the assignment operator, must be declared as a class member function.

Non-member operator functions can be declared to be friend functions of the class on which they will operate. However, granting friendship status to such functions grants them access to a class’s private attributes. This may or may not be in keeping with your class design goals.

The most often overloaded operator is the assignment operator. The purpose of the overloaded assignment operator is to properly handle deep copy operations. The compiler will provide a default assignment operator for your user-defined types but the default behavior is a bitwise member copy. Relying on the default assignment operator to properly handle complex class types usually results in disaster. Beware the default assignment operator and the effects of shallow vs. deep object copying.

Skill Building Exercises

1. **Expand Person Class:** Expand the capabilities of the Person class’s overloaded assignment operator to include copying sex and age data.
2. **Foo Class:** Create your own Foo class. Give it a few private data attributes. I will leave their types to your discretion. Overload the stream insertion and extraction operators to write the private members to a file, extract them from a file, and print them to the console. Use the overloaded insertion and extraction operators found in the Person class as a guide. Write a main() function to test the operators.

3. **Expand Foo Class:** Expand the capabilities of the Foo class you created above to include the following overloaded arithmetic operators: `*`, `/`, `%`. Write a `main()` function to test the operators.
4. **Expand Foo Class:** Expand the capabilities of the Foo class to include the following overloaded compound operators: `-=`, `*=`, `/=`, `%=`. Write a `main()` function to test the operators.
5. **Expand Foo Class:** Continue to expand the capabilities of the Foo class. Overload the following relational operators: `>`, `<=`, `>=`. Write a `main()` function to test these operators.

SUGGESTED PROJECTS

1. **Payroll Application:** Revisit the Payroll application shown in figure 13-14 in chapter 13. Overload the relational operators for both the `HourlyEmployee` and `SalariedEmployee` classes. The challenge here will be deciding what attribute in each class to use for the comparison. The comparison could also be the result of a calculation.
2. **Overload IOSTream Insertion and Extraction Operators:** Using the Payroll application from the previous chapter again, overload the stream insertion and extraction operators to save employees to a file, extract them from a file, and print them to the console.

SELF TEST QUESTIONS

1. In your own words describe the goal of operator overloading.
2. Describe the difference between a shallow copy and a deep copy.
3. What type of copy does a default, compiler-provided assignment operator perform?
4. Describe how operator overloading is related to function overloading.
5. (T/F) Operators overloaded in base classes can be overridden in derived classes.
6. What is the purpose of the dummy integer parameter in the postfix version of the increment and decrement operators?
7. What level of access does a friend function of a class enjoy?
8. How would non-member operator functions gain access to private class data members?
9. When overloading arithmetic operators, to what type of error checking should you pay particular attention?
10. What is the ultimate result and benefit of operator overloading?

REFERENCES

International Standard, ISO/IEC 14882, *Programming Languages — C++*, First Edition 1998-09-01

Bjarne Stroustrup. *The C++ Programming Language*, Third Edition. Addison-Wesley, Reading, Massachusetts. ISBN: 0-201-88954-4

Paul J. Lucas. *The C++ Programmer's Handbook*, Prentice Hall P T R, Englewood Cliffs, New Jersey. ISBN: 0-13-118233-1

Colin Flanagan. *Introduction To C++ Operators*, <http://www.ul.ie/~flanagan/ee6721/basoper/basoper.html>

NOTES

CHAPTER 15



DINNER IS SERVED

STATIC POLYMORPHISM: TEMPLATES

LEARNING OBJECTIVES

- Explain how to achieve static polymorphic behavior through the use of templates
- Explain how to write generic code using templates
- Describe the concept of a template class
- Explain how to declare and implement function templates
- Explain how to declare and implement class templates
- Explain how to declare and implement class member function templates
- Demonstrate your ability to declare and implement single parameter template classes
- Demonstrate your ability to declare and implement multiple parameter template classes
- Explain how to use components of the C++ Standard Template Library in your C++ programming projects
- State the purpose and use of STL iterators, algorithms, and containers
- Demonstrate your ability to utilize class and function templates to create generic code in support of your C++ programming projects

INTRODUCTION

Static polymorphic behavior is realized at compile time. A class template acts as a generic class definition from which new class types can be defined by a simple change of parameter types. Whereas a normal class is used to create objects, a class template is used to define new class types, from which objects are then created. The compiler uses the types you supply, combined with the class template, to create a wholly new class type.

Understanding how to declare, implement, and use class and function templates will pay big dividends in many ways. Primarily, it provides you with a mechanism to write generic code. Learning to write generic code can potentially save you a lot of work. Another benefit to learning templates is that it opens up to you the world of the C++ Standard Template Library (STL). The STL is a collection of class template components ready for your use. All you have to know to use STL components is an understanding of how class templates work.

In this chapter I will show you how to declare, implement, and use class templates. I will also introduce you to several important STL components and show how you can integrate them into your programs.

DEFINITION OF TEMPLATE

A template defines a related set of classes or functions. The related set of classes or functions defined by a template share the same code structure and functionality. The class or function template can then be used to declare a new class or function type.

FUNCTION TEMPLATES

A function template is a generic function declaration and definition from which different versions of the function can be created by the compiler based on the argument types used to call the function. If you think this sounds a lot like overloaded functions you are right. Function templates and overloaded functions are related as you will soon see.

CLASS TEMPLATES

A class template is a generic class declaration and definition from which different, but related, class types can be created by the compiler based on type parameters.

STRUCTURE TEMPLATES

A structure template is like a class template but using structures instead.

HOW TEMPLATES WORK: AN ANALOGY

When you declare and define a template you are creating a generic version of whatever piece of code you are writing, be it a function or a class. In the declaration and definition of the template you will use one or more identifiers as type placeholders. These type placeholders are similar in function to the placeholders in a form letter generated with a word processor.

Figure 15.1 illustrates a simple mail merge operation. A master letter is created with placeholders for certain data elements. The structure of the data source is mapped to the master letter by the placeholder identifiers name and age. When the mail merge function is executed, the data source is merged with the master letter to yield the finished letters. The master letter is a generic document that can be reused to generate many specific letter instances.

Class and function templates work in similar fashion. A generic function or class is declared and defined. Placeholders are inserted into the code to reserve spots for actual data types. When specific versions of a template function are required the type substitutions are made based on the types of the arguments used to call the function. In the case of template classes, a special syntax is used when a new template class is declared.

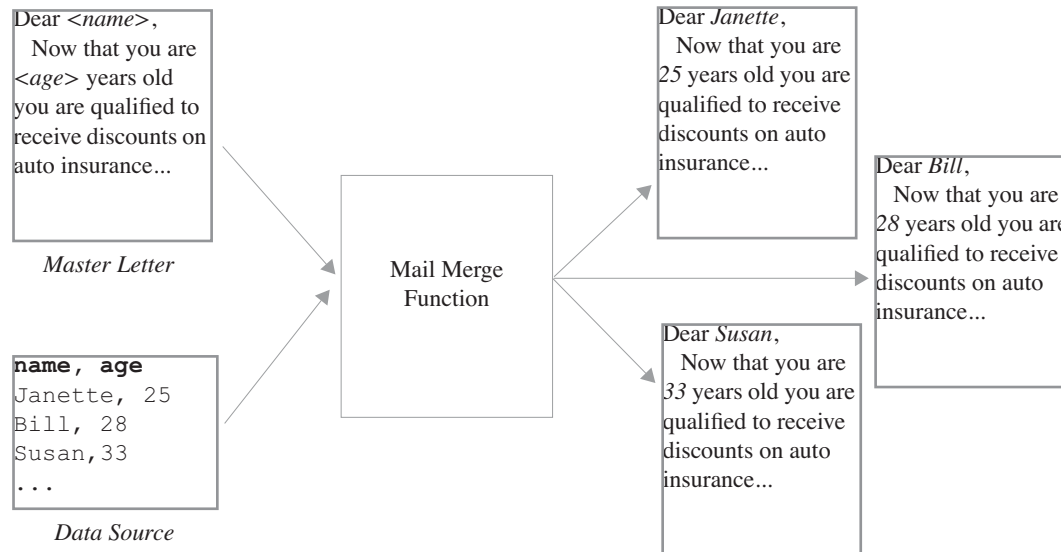


Figure 15-1: Placeholder Use In Mail Merge

DECLARING AND USING FUNCTION TEMPLATES

Up until now, if you wanted to create different versions of the same function to operate on different data types you would overload the function. For instance, if you wanted to declare a function named `Sum()` that took two arguments, added them together, and returned the result you could create several versions of the function like so:

```
int Sum(int val1, int val2);
float Sum(float val1, float val2);
char Sum(char val1, char val2);
```

These three functions can be replaced with one function template. Example 15.1 gives the header file for the function template named `Sum()`.

```

1  #ifndef SUM_TEMPLATE_H
2  #define SUM_TEMPLATE_H
3
4  template<class T>
5  T Sum(T val1, T val2){
6      return val1 + val2;
7  }
8
9  #endif

```

15.1 *sumtemplate.h*

The `Sum()` function is declared to be a template by the keyword `template` appearing on line 4. Following the keyword `template` in angle brackets is the keyword `class` followed by a placeholder identifier named `T`. The `class` keyword as it is used here essentially means “any type”. The placeholder `T` will then appear somewhere in the function. It can appear in more than one place, as it does here in the parameter list. You can use any valid identifier as a placeholder name, not just `T`. You can also declare more than one placeholder as you will see later.

SEPARATING DECLARATION FROM IMPLEMENTATION: SOME BACKGROUND

Notice in example 15.1 that both the declaration of the `Sum()` function template and its definition appear in the header file. Older UNIX compilers based on AT&T’s Cfront would allow you to split the declaration from the defini-

tion as we have been doing with ordinary functions throughout this book. However, splitting template declaration and definition had its problems and this method of doing business was largely abandoned by PC C++ compiler vendors in favor of grouping the declaration and definition of a class or function template in one header file. Eventually, some UNIX compiler vendors adopted this grouping practice as well. The ANSI C++ standard supports the separating approach but a lot of compiler vendors still only offer the grouping approach. So, in this chapter I will use the grouping approach as the examples should work for you regardless of your compiler.

WHEN IN DOUBT REFER TO YOUR COMPILER DOCUMENTATION

If you are using a brand new C++ compiler released within the last year you may be able to separate a template's declaration from its definition. Refer to your compiler's documentation to learn how this is done.

EXAMPLE 15.1 CONTINUED

OK, with the background discussion complete let us continue with the Sum() function template example. To use the Sum() function template just include the header file as you would normally do. The function is called the same way as normal functions are called. Example 15.2 shows a main() function using the Sum() function on different data types. Figure 15-2 shows the results of running Example 15.2.

```

1  #include <iostream>
2  #include "sumtemplate.h"
3  using namespace std;
4
5  int main(){
6      cout<<Sum(3, 25)<<endl;
7      cout<<Sum(3.456, 5.786)<<endl;
8      cout<<Sum('a', 'b')<<endl;
9      return 0;
10 }
```

15.2 main.cpp

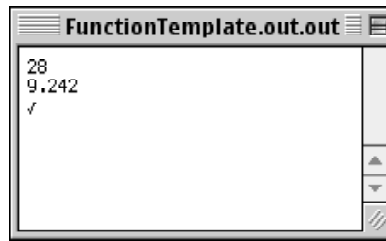


Figure 15-2: Results of Running Example 15.2

Using Multiple PLACEHOLDERS

The Sum() function template declared and defined in example 15.1 used one type placeholder named T to reserve type spots in the function. Because both of the Sum() function's parameters are reserved with the same placeholder they must be of the same type when the function is called. To illustrate, let us see what happens when the Sum() function is called with an integer and a float argument as shown in the following line of code:

```
cout<<Sum(3, 3.5)<<endl;
```

```
Error : function call 'Sum(int, double)' does not match 'Sum(T0, T0)'
main.cpp line 10  cout<<Sum(3,3.5)<<endl;
```

Figure 15-3: Error Resulting from Calling Sum() with Two Different Type Arguments

When the Sum() function template is called with two different argument types an error results. This error was produced using the Macintosh version of Metrowerks CodeWarrior Release 5. One way to eliminate this error is to declare the Sum() function template to use two different placeholders. Example 15.3 shows the redeclared Sum() function.

```

1  #ifndef SUM_TEMPLATE_H
2  #define SUM_TEMPLATE_H
3
4  template<class T, class U>
5  T Sum(T val1, U val2){
6      return val1 + val2;
7  }
8  #endif

```

15.3 sumtemplate.h

Notice now there are two type placeholders declared on line 4 of example 15.3, T and U. The U placeholder is used for the second parameter in the Sum() function while the T placeholder is used for the first parameter and the return value. This will solve one problem but introduce another. Example 15.4 shows the revised Sum() function template in use. Figure 15-4 shows the results of running Example 15.4.

```

1  #include <iostream>
2  #include "sumtemplate.h"
3  using namespace std;
4
5  int main(){
6      cout<<Sum(3, 25)<<endl;
7      cout<<Sum(3.456, 5.786)<<endl;
8      cout<<Sum('a', 'b')<<endl;
9      cout<<Sum(3, 3.5)<<endl;
10     return 0;
11 }

```

15.4 main.cpp

When the first argument is an integer the return type is an integer too...



Figure 15-4: Results of Running Example 15.4

Refer to line 9 of example 15.4. Notice now that the Sum() function is called with the first argument an integer and the second argument a float. By using two placeholders in the function template the error produced by using two different argument types is eliminated. However, the result type of the Sum() function is dictated by the first argument type. Since the T placeholder is used to reserve the type spot for both the first parameter and the return type of the function, whatever type the first argument to the function happens to be will also be the return type of the function. In this example it is an integer. So, notice in figure 15-4 that the result of calling the Sum() function with the arguments 3 and 3.5 is 6, not 6.5! Notice what happens when the order of the arguments are swapped. Example 15.5 shows the revised main() function. Figure 15-5 shows the results of running example 15.5.

```

1  #include <iostream>
2  #include "sumtemplate.h"
3  using namespace std;
4
5  int main(){
6      cout<<Sum(3, 25)<<endl;
7      cout<<Sum(3.456, 5.786)<<endl;
8      cout<<Sum('a', 'b')<<endl;
9      cout<<Sum(3.5, 3)<<endl;
10     return 0;
11 }

```

15.5 main.cpp

Argument types reversed...

When the first argument is a float the return type is a float as well.

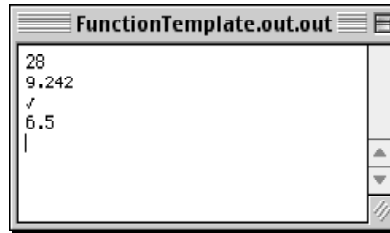


Figure 15-5: Results of Running Example 15.5

To resolve the ambiguous return type issue simply declare yet another type placeholder used specifically to dictate the function's return type. Example 15.6 gives the revised `Sum()` function with the extra type placeholder `V` declared and used to reserve the return type.

```

1  #ifndef SUM_TEMPLATE_H
2  #define SUM_TEMPLATE_H
3
4  template<class T = int, class U = int, class V = int>
5  V Sum(T val1, U val2){
6      return val1 + val2;
7  }
8  #endif

```

15.6 sumtemplate.h

Now, you may be asking yourself, how the heck will the return type be set? Good question. Since the function only has two parameters it can only be called with two arguments. The only way out of this pickle is to use the special template definition syntax when you call the function. Oh, before I forget, notice on line 4 that the template parameters now have default types assigned. Example 15.7 gives the revised `main()` function showing how to call the function templates using the template syntax.

```

1  #include <iostream>
2  #include "sumtemplate.h"
3  using namespace std;
4
5  int main(){
6      cout<<Sum<int, int, int>(3, 25)<<endl;
7      cout<<Sum<double, double, double>(3.456, 5.786)<<endl;
8      cout<<Sum<char, char, char>('a', 'b')<<endl;
9      cout<<Sum<double, int, double>(3.5, 3)<<endl;
10     return 0;
11 }

```

15.7 main.cpp

Notice the `Sum()` function call on line 6. The function name `Sum` is followed by a series of type names enclosed in angle brackets. This is referred to as a template specialization. The order of type names appearing in the specialization map to the same order as the function template type parameters. On line 6 the `Sum()` function is being specialized to take two integers as arguments to the function and return an integer value. Following the specialization is the argument list appearing in parentheses as usual.

Line 7 introduces another `Sum()` function specialization, as does line 8 and line 9. The specialization on line 9 says that the `Sum()` function will take a double as the first argument, an integer as the second argument, and return a double value.

Quick Review

Static polymorphism happens at compile time. Function templates are generic function definitions that are used by the compiler to construct specific function versions based on the types of arguments used to call the function. Function template declaration and definition should appear in the same header file unless your compiler supports separating the declaration of a function template from its definition. Check your compiler documentation.

Function templates are declared using the `template` keyword. Type placeholders are declared using the `class` keyword and can be any valid identifier. There can be more than one type placeholder declared and used in a function

template. Because the compiler uses the function template to build a specific version of the function, using more than one type in a function call to a function having only one type placeholder can be problematic.

Use the template specialization syntax to specify placeholder types when the function template is called.

DECLARING AND USING CLASS TEMPLATES

Class templates are used to declare and define a generic class structure. The compiler uses the class template and any types supplied via specialization to build a new class type. Let us begin the discussion of class templates with a simple Foo example. Example 15.8 shows the declaration and definition of a class template named Foo.

```

1  #ifndef FOOTEMPLATEDEF_H
2  #define FOOTEMPLATEDEF_H
3
4  template<class T>
5  class Foo{
6      public:
7          Foo(T _val);
8          virtual ~Foo();
9          void setVal(T _val);
10         T getVal();
11     private:
12         T val;
13 };
14
15 template<class T>
16 Foo<T>::Foo(T _val):val(_val){}
17
18 template<class T>
19 Foo<T>::~~Foo(){}
20
21 template<class T>
22 void Foo<T>::setVal(T _val){
23     val = _val;
24 }
25
26 template<class T>
27 T Foo<T>::getVal(){
28     return val;
29 }
30 #endif

```

15.8 foodef.h

The declaration of the Foo class templates begins on line 4 with the keyword `template`. There is only one template parameter declared named `T`. The `T` is used throughout the class declaration and definition to reserve a spot for the type declared when the Foo class is specialized. Notice that both the class declaration and definition appear in the same header file.

Refer to line 26. Each Foo class member function definition must be preceded with the template declaration. To keep your source code clean I recommend putting the template declaration on its own line and begin the function definition on the next line down.

Refer to line 27. Notice how the name of the class is `Foo<T>`, not plain ol' Foo! Otherwise, the definition of class functions proceed as usual except the `T` placeholder is used where needed. Example 15.9 shows a `main()` function using the Foo class template with various specializations. Figure 15-6 shows the results of running this program.

```

1  #include <iostream>
2  #include "foodef.h"
3  using namespace std;
4
5  int main(){
6      Foo<int> f1(1);
7      Foo<char> f2('d');
8      cout<<f1.getVal()<<endl;
9      cout<<f2.getVal()<<endl;
10     return 0;
11 }

```

15.9 main.cpp



Figure 15-6: Results of Running Example 15.9

Referring to line 6 of example 15.9, notice how the Foo class template is specialized to use an int type. The important point to note in this example is that Foo<int> and Foo<char> are two distinct types.

A MORE COMPLEX CLASS TEMPLATE EXAMPLE

Example 15.10 gives the source code for a template version of the DynamicArray class originally introduced in chapter 14.

```

1  #ifndef _DYNAMIC_ARRAY_H
2  #define _DYNAMIC_ARRAY_H
3
4  template<class T> class DynamicArray{
5      public:
6          DynamicArray(int _size = 5);
7          virtual ~DynamicArray();
8          T& operator[] (unsigned i);
9          int getSize();
10     private:
11         T* its_array;
12         int size;
13 };
14
15 template<class T>
16 DynamicArray<T>::DynamicArray(int _size):size(_size){
17     its_array = new T[_size];
18     for(int i=0; i<size; i++)
19         its_array[i] = static_cast<T>(0);
20 }
21
22 template<class T>
23 DynamicArray<T>::~DynamicArray(){
24     delete[] its_array;
25 }
26
27 template<class T>
28 T& DynamicArray<T>::operator[] (unsigned i){
29     if(i >= (size)){
30         int newsize = size+10;
31         T* temp = new T[newsize];
32         for(int j = 0; j<size; j++){
33             temp[j] = its_array[j];
34         }
35         delete[] its_array;
36         its_array = new T[newsize];
37         for(int j = 0; j<size; j++){
38             its_array[j] = temp[j];
39         }
40
41         for(int j=size; j<newsize; j++){
42             its_array[j] = static_cast<T>(0);
43         }
44         delete[] temp;
45         size = newsize;
46
47         return its_array[i];
48     } else return its_array[i];
49 }
50
51 template<class T>
52 int DynamicArray<T>::getSize(){ return size;}
53
54 #endif

```

15.10 dynamicarraydef.h

Converting the `DynamicArray` class into a class template increased its usefulness as it can now be used to hold different types of objects, even user-defined types. Example 15.11 gives a `main()` function showing the `DynamicArray` class template in use.

```

1  #include <iostream>
2  using namespace std;
3  #include "dynamicarraydef.h"
4
5  int main() {
6      DynamicArray<char> d1;
7      DynamicArray<float> d2;
8
9      for(int i=0; i<6; i++){
10         d1[i] = 'a';
11     }
12     for(int i=0; i<6; i++){
13         d2[i] = (i + .5);
14     }
15
16
17     for(int i=0; i<d1.getSize(); i++){
18         cout<<d1[i]<<"      "<<d2[i]<<endl;
19     }
20
21     return 0;
22 }

```

15.11 main.cpp

Figure 15-7 shows the results of running example 15.11.

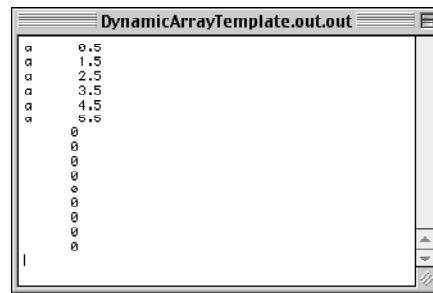


Figure 15-7: Results of Running Example 15.11

Quick Review

Class templates provide a way to write generic classes to gain an additional measure of code reuse. Class template declarations are started with the template keyword followed by the template parameter list. Although not shown in the two class template examples above, class templates can have more than one template parameter.

Group class template declaration and definition in a common header file unless your compiler supports their separation. Consult your compiler documentation.

OVERVIEW OF THE STANDARD TEMPLATE LIBRARY (STL)

Writing generic code using templates can be fun, but, it is not always easy to write truly generic code. Planning for every contingency requires time, effort, and lots of thought. Here is where the C++ Standard Template Library (STL) can be a huge help. Before sitting down to write a class or function template, take time to explore the STL. It provides lots of functionality and many smart people have contributed to both its functionality and robustness over the years. This section will introduce you to a small portion of the STL. What you have learned in this chapter about function and class templates will make the following material easily understood. To learn more about the STL I recommend reading one of the excellent references listed at the end of this chapter.

The Standard Template Library provides functionality to you in three primary forms: containers, iterators, and algorithms. STL components can be used in your programs by including the necessary header files. The number and names of these header files may vary depending on the age of your compiler. Modern compilers conforming to the C++ standard will have the following STL header files:

```

<algorithm>    <deque>    <functional>    <iterator>    <vector>
<list>        <map>        <memory>       <numeric>
<queue>       <set>        <stack>        <utility>

```

CONTAINERS AND CONTAINER ADAPTERS

Containers are STL components that manage a sequence of elements. The STL provides the container components listed in table 15-1:

Container Class	Header File	Container Type	Comment
deque	<deque>	non-associative	Double ended queue. Elements can be added and removed from either end.
list	<list>	non-associative	Stores elements in a doubly linked list. Provides quick insertion but each element must be sequentially accessed.
map	<map>	associative	Stores elements in an ordered binary tree. Does not allow adjacent elements to have keys with equivalent ordering. Stores elements of type pair<const key, T>. Only the const key participates in the ordering.
multimap	<map>	associative	Stores elements in an ordered binary tree. Allows adjacent elements to have keys with equivalent ordering. Stores elements of type pair<const key, T>. Only the const key participates in the ordering.
set	<set>	associative	Stores elements in an ordered binary tree. Does not allow adjacent elements to have keys with equivalent ordering. Stores elements of type pair<const key, T>. Both the const key and T values participate in the ordering.
multiset	<set>	associative	Stores elements in an ordered binary tree. Allows adjacent elements to have keys with equivalent ordering. Stores elements of type pair<const key, T>. Both the const key and T values participate in the ordering.
vector	<vector>	non-associative	Stores elements as an array of contiguous elements.

Table 15-1: STL Containers

The STL also provides the container adapters listed in table 15-2. A container adapter uses one of the container components listed in table 15-1 to do their dirty work.

Container Adapter	Header File	Comments
stack	<stack>	Inserts and removes elements in last-in/first out (LIFO) sequence. You specify what STL component will actually be used to store stack elements.
queue	<queue>	Inserts and removes elements in first-in/first-out (FIFO) sequence. You specify what STL component will actually be used to store queue elements.
priority_queue	<queue>	Removes elements based on their priority. You specify what STL component will actually be used to store priority_queue elements.

Table 15-2: STL Container Adapters**ITERATORS**

Iterators provide a mechanism to uniformly manipulate elements held in a container component regardless of the container type. Iterators are closely related to pointers, so, if you understand how to use pointers then you will catch on to the purpose of iterators quickly.

It is truly beyond the scope of this book to provide a detailed discussion of iterators. If you are writing template code you can define iterators that operate properly in the context of your design. If you are just using existing STL components to enhance your programs then an understanding of how to gain access to an iterator and use it to manipulate container-stored elements will take you far.

To give you some idea as to what iterators are consider the issue of manipulating an array of objects as shown in example 15.12.

```

1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      char message[] = "Hello World!";
6      char* cptr;
7      int message_length = sizeof(message);
8
9      for(int i = 0; i < message_length; i++){
10         cout<<message[i];
11     }
12
13     cout<<endl;
14
15     for(cptr = message; cptr != (message + message_length); cptr++){
16         cout<<*cptr;
17     }
18
19     return 0;
20 }
```

15.12 main.cpp

On line 5 a character array named `message` is declared and initialized to contain the string “Hello World!”. A character pointer is declared on line 6 named `cptr`, and on line 7 an integer variable named `message_length` is declared and initialized to the value of the size of the message array.

The for statement beginning on line 9 iterates over the contents of the message array in the idiomatic way you have seen throughout this text. It results in the message “Hello World” being printed to the console.

The for statement on line 15 does the same thing only it utilizes a pointer instead of an index value to iterate over the array. In the initialization section of the for statement the character pointer `cptr` is initialized to point to first character of the message array. During the iteration, the value of `cptr` is compared to the value of the address one element past the length of the message array. Using address arithmetic the value of `cptr` is incremented during each iteration. In this example, the character pointer `cptr` is used as an iterator.

Container components provide the iterators necessary to manipulate their elements. Container components also provide several member functions that simplify the use of iterators compared to the code shown in example 15.12.

Algorithms

In addition to container components and iterators, the STL provides algorithm function templates you can use to further manipulate container elements. Algorithm function templates rely heavily on iterators to provide their functionality. Table 15-3 provides a list of the different algorithms provided by the STL. For a detailed discussion of each I recommend consulting one of the STL references listed at the end of this chapter.

adjacent_find	find	lexicographical_compare	nth_element	remove_copy_if	search	swap_ranges
binary_search	find_end	lower_bound	partial_sort	remove_if	search_n	transform
copy	find_first_of	make_heap	partial_sort_copy	replace	set_difference	unique
copy_backward	find_if	max	partition	replace_copy	set_symmetric_difference	unique_copy
count	for_each	max_element	pop_heap	replace_copy_if	set_union	upper_bound
count_if	generate	merge	prev_permutation	replace_if	sort	
equal	generate_n	min	push_heap	reverse	sort_heap	
equal_range	includes	min_element	random_shuffle	reverse_copy	stable_partition	
fill	inplace_merge	mismatch	remove	rotate	stable_sort	
fill_n	inter_swap	next_permutation	remove_copy	rotate_copy	swap	

Table 15-3: STL Algorithm Function Templates

Note that it is not necessary to understand containers to utilize the algorithm function templates, however, an understanding of iterators is required.

Quick Review

The C++ standard template library provides containers, iterators, and algorithms for use in your programs. Containers are components that manage a set of elements. Container adapters rely on containers to implement their functionality. The type of container used by a container adapter is left to the choice of the programmer.

Iterators provide a mechanism to manipulate the elements managed by a container. Algorithms make extensive use of iterators to provide you with additional alternatives to manipulate container elements.

Using Standard Template Library Components

This section will demonstrate the use of several STL components and how iterators are used to manipulate container elements. Along the way you will be introduced to several important functions shared by all STL container components that make using iterators to manipulate container elements easy. Additionally, the use of several STL algorithms will be demonstrated. Essentially, once you know how to use one STL container you know how to use them all. Your job then is to determine what STL component best suits your particular need.

Using VECTOR

A vector provides the functionality of an array with the additional functionality of being dynamically sizeable. Since it is a class template, it can be used to store just about any element type. I will caution that how a vector behaves with different types, especially pointers, is a function of the quality of the version of the STL library that ships with your compiler. If you have a fairly recent implementation of the C++ STL everything should work fine. Again, consult your compiler's documentation.

Example 15.13 gives a main() function showing a vector of floats being declared and utilized like an ordinary array. The output from this program is shown in figure 15-8. Notice how the vector v is accessed using the subscript operator. Example 15.14 shows the same vector being utilized in a slightly different fashion.

```

1  #include <iostream>
2  using namespace std;
3  #include <vector>
4
5  int main(){
6      vector<float> v(10);
7
8      for(int i=0; i<10; i++){
9          v[i] = ( i + .25);
10     }
11
12     for(int i=0; i<10; i++){
13         cout<<v[i]<<endl;
14     }
15
16     return 0;
17 }

```

15.13 main.cpp

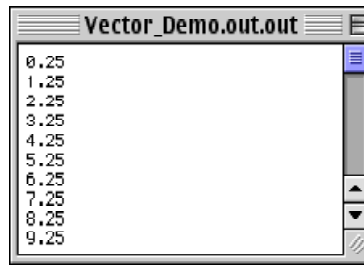


Figure 15-8: Results of Running Example 15.13

```

1  #include <iostream>
2  using namespace std;
3  #include <vector>
4
5  int main(){
6      vector<float> v;
7
8      for(int i = 0; i<10; i++){
9          v.push_back(i + .34);
10     }
11
12     for(int i = 0; i<v.size(); i++){
13         cout<<v[i]<<endl;
14     }
15     cout<<endl;
16
17     for(vector<float>::iterator i = v.begin(); i != v.end(); ++i){
18         cout<<*i<<endl;
19     }
20
21     return 0;
22 }

```

15.14 main.cpp

Notice how an iterator object is declared and used...

Compare the difference between the declaration of the vector object in example 15.13 and that of Example 15.14. In example 15.13, the vector object `v` was given a size of 10 whereas in example 15.14 no size was given. On line 9 in example 15.14 the `push_back()` function is used to insert elements at the end of the array. In the for statement shown on line 12 the `size()` function is used to determine the number of elements managed by the container.

The for statement on line 17 shows how an iterator is declared and utilized to iterate over the vector container elements. The `begin()` function returns an iterator pointing to the first element in the container. The `end()` function returns the iterator of one past the last element. Each container element is then accessed by dereferencing the iterator as shown in the body of the for statement. Figure 15-9 shows the results of running example 15.14. Notice that both versions of the for statement produce the same results.

Example 15.15 shows the sort algorithm being used on a vector object. In this example, the `push_back()` function is used to load a vector object `v` with ten wildly random integers plucked from the depths of my brain. (*Randomness not guaranteed!*) The first for statement prints the contents of the vector to the console in their insertion order. The sort algorithm function is called on line 25. It takes an iterator that points to the first element to be included in the sort

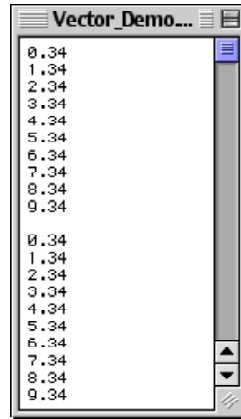


Figure 15-9: Results of Running Example 15.14

```

1  #include <iostream>
2  using namespace std;
3  #include <vector>
4  #include <algorithm>
5
6  int main(){
7      vector<int> v;
8
9      v.push_back(25);
10     v.push_back(278);
11     v.push_back(57);
12     v.push_back(82);
13     v.push_back(2);
14     v.push_back(29);
15     v.push_back(485);
16     v.push_back(123);
17     v.push_back(1);
18     v.push_back(10);
19
20     for(int i = 0; i<v.size(); i++){
21         cout<<v[i]<<" ";
22     }
23     cout<<endl;
24
25     sort(v.begin(), v.end());
26
27     for(vector<int>::iterator i = v.begin(); i != v.end(); ++i){
28         cout<<*i<<" ";
29     }
30
31     return 0;
32 }
33

```

15.15 main.cpp

and an argument denoting one element past the last element to be sorted. The proper iterators are obtained by using the vector object's `begin()` and `end()` functions. The `for` statement on line 27 then prints the now-sorted vector elements. Figure 15-10 shows the results of running example 15.15.

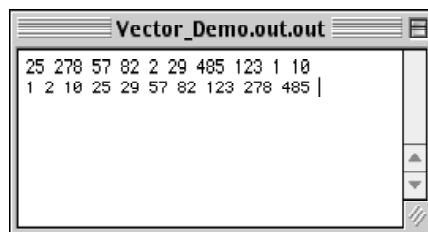


Figure 15-10: Results of Running Example 15.15

Using list

Example 15.16 gives a main() function showing an STL list component in action.

15.16 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include <list>
4
5  int main(){
6      list<int> l;
7
8      l.push_back(82);
9      l.push_back(25);
10     l.push_back(63);
11     l.push_back(8);
12     l.push_back(75);
13     l.push_back(12);
14     l.push_back(23);
15     l.push_back(129);
16     l.push_back(273);
17     l.push_back(1);
18
19     for(list<int>::iterator i = l.begin(); i != l.end(); i++){
20         cout<<*i<<" ";
21     }
22     cout<<endl;
23
24     l.sort();
25
26     for(list<int>::iterator i = l.begin(); i != l.end(); i++){
27         cout<<*i<<" ";
28     }
29     cout<<endl;
30
31     return 0;
32 }
```

Two important comparisons can be made between the use of a list as compared to a vector. First, the interface to each container is similar by design. The push_back() function works on both. Second, their contained elements are manipulated in a uniform manner by using iterators.

Third, the sort algorithm does not work on lists. Notice that the sort() function called on line 24 is actually a list member function, not a stand-alone algorithm. Figure 15-11 shows the results of running this program.

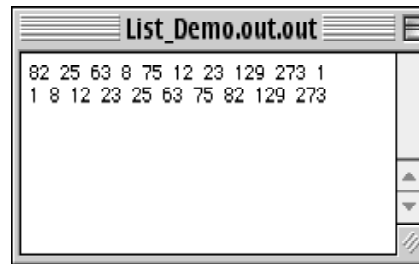


Figure 15-11: Results of Running Example 15.16

Quick Review

Container, iterators, and algorithms all work together to add ready-made functionality to your programs. Getting to know the STL can save you a lot of otherwise wasted effort expended in trying to replicate what is already available.

SUMMARY

Static polymorphism happens at compile time. Function templates are generic function definitions used by the compiler to construct specific function versions based on the types of arguments used to call the function. Function template declaration and definition should appear in the same header file unless your compiler supports separating the declaration of a function template from its definition.

Function templates are declared using the `template` keyword. Type placeholders are declared using the `class` keyword and can be any valid identifier. There can be more than one type placeholder declared and used in a function template. Because the compiler uses the function template to build a specific version of the function, using more than one type in a function call to a function having only one type placeholder can be problematic.

Use the template specialization syntax to specify placeholder types when the function template is called.

Class templates provide a way to write generic classes to gain an additional measure of code reuse. Class template declarations are started with the `template` keyword followed by the template parameter list. Class templates can have more than one template parameter just like their function template cousins. Group class template declaration and definition in a common header file unless your compiler supports their separation.

The C++ standard template library (STL) provides containers, iterators, and algorithms for use in your programs. Containers are components that manage a set of elements. Container adapters rely on containers to implement their functionality. The type of container used by a container adapter is left to the choice of the programmer.

Iterators provide a mechanism to manipulate the elements managed by a container. Algorithms make extensive use of iterators to provide you with additional alternatives to manipulate container elements.

Container, iterators, and algorithms all work together to add ready-made functionality to your programs. Getting to know the STL can save you a lot of otherwise wasted effort expended in trying to replicate what is already available.

Skill Building Exercises

1. **Research:** Explore your compiler! Study your compiler's documentation and determine how it implements template functionality. Enter and compile the function and class template examples given in this chapter. Do they behave the same or are there behavioral differences? Are there any surprises?
2. **Function Template:** Write a function template named `max` that takes two object references as arguments and returns a reference to the larger of the two objects.
3. **Function Template:** Write a function template named `compare` that takes three arguments. The first two arguments will be object references. The third argument will be a comparator function. (For a hint on how to approach this exercise see the section on function pointers in chapter 9)
4. **Class Template:** Write a class template that implements a doubly-linked list that can store any type value.
5. **Research:** Procure one of the standard template library references listed at the end of this chapter and read it.
6. **STL:** Explore the standard template library. Write a short program that tests all the container components.
7. **Research:** List the STL container components. Write a brief description of each one and explain in what programming scenarios each would come in handy.
8. **Research:** Do some more research on iterators. How many different types of iterators are there and how is each used?
9. **Research:** List each of the STL algorithms. Write a brief description of each one and on what types of components

they can be used.

10. **Pop Quiz:** What company was initially responsible for the development of the C++ standard template library?

SUGGESTED PROJECTS

1. **STL:** Use the stack STL component to write a program that emulates a hand-held calculator. Use reverse polish notation (RPN) to perform the operations.
2. **Queuing Theory:** Study queuing theory. Use the STL component queue and component adapter priority_queue to write a program to model the teller operations of a bank. Answer the following question: If customers arrive at the rate of three per minute during the noon rush, how many tellers are needed to keep the wait in line to five minutes or less? Assume each transaction takes an average of three minutes normally distributed.
3. **Modify Robot Rat:** Knowing what you know now about templates and the STL, what modifications can be made to RobotRat to make it even better? Perform those modifications. If you can't think of any, shame on you!
4. **Modify Employee:** Revisit the Employee example given in chapter 13. Use the STL map component to store employee records using Employee number as the key.

SELF TEST QUESTIONS

1. In your own words explain what static polymorphism is and how it is achieved in C++ using function and class templates.
2. In your own words give a definition of a template.
3. How are function templates related to overloaded function?
4. How are places reserved in source code for type substitution?
5. How many type placeholders can a function or class template declare?
6. If the number of type placeholders declared in a function template is greater than the number of function parameters, how can the type of extra placeholders be specified when the function is called?
7. Why is it a good idea to group the declaration and definition of a class or function template in one header file?
8. Is it necessary to group the declaration and definition of a class or function template in one header file?
9. What is the purpose of an STL container adapter?
10. What are iterators and how are they related to pointers? What's the purpose of an iterator? What benefits do you gain from using iterators to manipulate container elements vs. other iterative methods.

REFERENCES

International Standard, ISO/IEC 14882, *Programming Languages – C++*, First Edition 1998-09-01

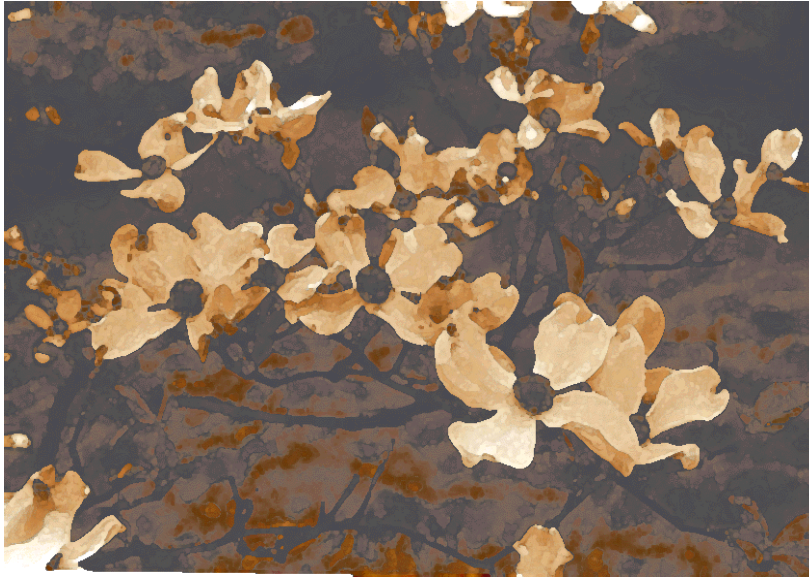
Bjarne Stroustrup. *The C++ Programming Language*, Third Edition. Addison-Wesley, Reading, Massachusetts. ISBN: 0-201-88954-4

P. J. Plauger, et. al. *The C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0-13-437633-1

Leen Ammeraal. *STL For C++ Programmers*. John Wiley & Sons. Chichester, West Sussex, England. ISBN: 0-471-97181-2

NOTES

CHAPTER 16



Dogwoods

DYNAMIC POLYMORPHISM: OBJECT-ORIENTED PROGRAMMING

LEARNING OBJECTIVES

- STATE THE DEFINITION OF DYNAMIC POLYMORPHISM
- EXPLAIN HOW TO ACHIEVE DYNAMIC POLYMORPHIC BEHAVIOR THROUGH THE USE OF BASE CLASS POINTERS AND DERIVED CLASS OBJECTS
- STATE THE IMPORTANCE OF ABSTRACT BASE CLASSES IN OBJECT-ORIENTED DESIGN
- DESCRIBE THE ROLE VIRTUAL FUNCTIONS PLAY IN IMPLEMENTING DYNAMIC POLYMORPHIC BEHAVIOR
- STATE THE PURPOSE AND USE OF VIRTUAL DESTRUCTORS
- DESCRIBE THE CONCEPT OF PURE VIRTUAL FUNCTIONS
- STATE THE PURPOSE AND USE OF ABSTRACT BASE CLASSES
- STATE THE IMPORTANCE OF A CONSISTENT DERIVED CLASS INTERFACE AND THE ROLE IT PLAYS IN ACHIEVING ROBUST POLYMORPHIC BEHAVIOR
- EXPLAIN WHY POLYMORPHIC BEHAVIOR IS A CRITICAL COMPONENT OF GOOD OBJECT-ORIENTED DESIGN
- DEMONSTRATE YOUR ABILITY TO UTILIZE DYNAMIC POLYMORPHISM IN YOUR C++ PROGRAMMING PROJECTS
- LIST AND DEFINE THE FOLLOWING TERMS: BASE CLASS, ABSTRACT BASE CLASS, VIRTUAL FUNCTION, PURE VIRTUAL FUNCTION, VIRTUAL DESTRUCTOR, INHERITANCE HIERARCHY, BASE CLASS POINTER, DERIVED CLASS OBJECT, AND DYNAMIC POLYMORPHIC BEHAVIOR

INTRODUCTION

Dynamic polymorphism enables object-oriented programming. Understanding how to properly implement dynamic polymorphic behavior leads to clean, flexible, and robust software design. Most importantly, understanding dynamic polymorphism results in a fundamental programming-thought-process paradigm shift. When the concepts associated with achieving dynamic polymorphic behavior come together in your mind in that sudden spark of understanding it will be as if the proverbial light bulb suddenly turns on. You will begin to think differently about the structure of programs. You will think in terms of objects.

The material presented in this chapter builds upon and reinforces material introduced in chapter 13. I will begin by presenting good working definitions of abstraction, object-oriented programming, and dynamic polymorphism. I will then discuss the language features used to implement dynamic polymorphic behavior. The concepts of abstract base classes, pure virtual member functions, function overriding, base class pointers, and derived class objects will be discussed in the context of dynamic polymorphism.

ABSTRACTION: Amplify THE ESSENTIAL—ELIMINATE THE IRRELEVANT

Abstraction allows you to tame software conceptual complexity. When designing class types your goal must be to identify the essential features of that class of objects. If you're designing a base class the essential features identified must also apply to all possible subclasses. The notion of essentiality can be reduced to pure abstraction by the creation of an abstract base class containing only a public interface comprised of pure virtual functions.

The irrelevant manifests itself in the implementation and thus encapsulation is used to hide implementation details. Implementation can be further removed from interface by placing implementation details in a separate class from that which declares the interface. You will see an example of this later.

Abstraction and inheritance are force multipliers. By using them you gain an incredible amount of power to reduce conceptual complexity, treating groups of different objects the same via a shared interface. This grouping of objects via a shared interface is an important concept that allows us to ignore the differences between derived class objects and treat them all the same as figure 16-1 illustrates.

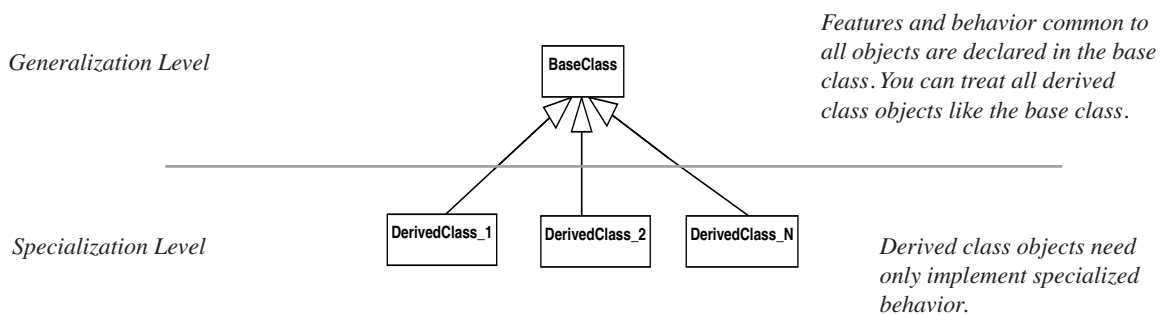


Figure 16-1: Base Class Declares Behavior Shared By All Derived Class Objects

As it turns out, most of the difficulty of any software design lies in getting the abstractions and inheritance relationships just right. One false move and you'll soon discover you have programmed yourself into a corner from which the only escape leads to design rot. Proficiency in this matter is gained only through experience.

OBJECT-ORIENTED PROGRAMMING DEFINED

Object-oriented programming is several things: First, it is a way of thinking about how abstractions called objects interact. Object-oriented thinking is different from procedural thinking. Object-oriented thinking empowers you to tame conceptual complexity. The analysis of objects and their interaction within a problem domain is referred to as Object-Oriented Analysis and Design (OOA&D).

Second, object oriented programming is a way of executing in code the design derived from applying OOA&D. Programming languages like C++, Java, SmallTalk, etc., provide features that support object-oriented programming. However, simply programming in an object-oriented language does not constitute object-oriented programming. For instance, merely converting an existing C program to C++ does not imply the converted code is now magically object-oriented.

Object-oriented programming is the implementation and application of polymorphic classes to create a software application. In C++ polymorphic classes are created with a combination of language features. Abstract base classes, virtual or pure virtual functions, public inheritance, derived classes, and base class pointers or references pointing to derived class objects all combine to enable the execution of an object-oriented design. Essentially, to write object-oriented programs in C++ requires an understanding of almost every language feature.

An object-oriented program is designed to utilize dynamic polymorphism. The application architecture is described and implemented in terms of object interfaces. A well-designed application architecture is stable and extensible.

DYNAMIC POLYMORPHISM DEFINED

Dynamic polymorphism combines dynamic binding with polymorphic classes to enable the creation of object-oriented programs in C++. Binding is the association of a symbol name to a storage address. A statically bound name is assigned a storage location at compile time. With dynamic binding the association of a name to a storage location is deferred until the first call to that name is made. With polymorphic classes, a virtual function call can result in the execution of a derived class's overriding function. The derived class object itself can be created at runtime using the new operator and its address assigned to a base class pointer.

LANGUAGE FEATURES THAT SUPPORT OBJECT-ORIENTED PROGRAMMING

Table 16-1 lists and describes the C++ language features employed to write object-oriented programs. All of the language features listed are used together to achieve dynamic polymorphic behavior in support of object-oriented programming.

Language Feature	Description
<i>base class</i>	A class that provides some form of generalized behavior to its subclasses. A base class can declare one or more virtual functions that can be overridden in subclasses. Base classes can exist to declare a common interface only. Such classes will contain pure virtual functions which make them an abstract base class. A base class must have a virtual destructor to enable the proper destruction on objects in the inheritance hierarchy.

Table 16-1: Language Features That Support Object-Oriented Programming

Language Feature	Description
<i>abstract base class</i>	<p>A class containing one or more pure virtual functions that serves as a base class. There can be no instances of abstract base class objects, however, abstract base class type pointers can be assigned the address of derived (concrete) class objects. Example:</p> <pre>class AbsClass { public: virtual ~AbsClass(); virtual void function_a() = 0; virtual void function_b() = 0; };</pre>
<i>base class pointer</i>	A pointer declared to be a particular base class type. The pointer can hold addresses of non-abstract base class objects or derived class objects. Virtual function calls are made via base class pointers to derived class objects.
<i>virtual function</i>	A function declared virtual in a base class can be overridden in a derived class and accessed via a base class pointer.
<i>pure virtual function</i>	A function declared virtual which is assigned the value of zero. The zero assignment indicates that function implementations must be provided by a derived class. For an example of a pure virtual function see abstract base class description above.
<i>inheritance</i>	<p>The act of extending the functionality of a base class through the declaration of a subclass. Virtual base class functions can be overridden in the subclass if required. A pure virtual function must be overridden in a subclass eventually. Where this happens exactly depends on your design. A base class and its set of related subclasses constitute an inheritance hierarchy. There is no limit to the extent of an inheritance hierarchy other than that dictated by good design.</p> <p>C++ supports three types of inheritance: public, protected, and private. Inheritance is controlled via the access specifiers <code>public</code>, <code>protected</code>, and <code>private</code>.</p>
<i>derived class (subclass)</i>	A class that inherits the functionality of one or more classes. The derived class provides specialized behavior, extending the generalized behavior of the base class or classes. If a base class contains a virtual function, that function can be overridden in a derived class if the design so dictates. If a base class contains a pure virtual function a derived class can override it and provide an implementation, or it can choose to not override it, in which case the derived class will itself become an abstract base class. (An overriding implementation must eventually appear in a derived class somewhere down the inheritance hierarchy)
<i>virtual destructor</i>	A base class constructor must be declared virtual for the proper destruction of objects in an inheritance hierarchy.
<i>encapsulation</i>	Implementation details should be kept private to a class. An object's desired behavior is made accessible via a set of public interface functions. Accessibility to class functions and attributes is controlled through the use of the access specifiers <code>public</code> , <code>protected</code> , and <code>private</code> .
<i>polymorphic class</i>	A class that declares or inherits a virtual function.
<i>compile-time type checking</i>	The C++ compiler will ensure a particular class supports a particular interface function call. Any attempt to call a non-supported function on an object will result in a compile-time error.

Table 16-1: Language Features That Support Object-Oriented Programming

AN EXAMPLE: CLASS INTERFACE

This section presents an example that illustrates the use of the language features discussed in the previous section. Figure 16-2 shows a UML class diagram with three classes, Interface, DerivedClassOne, and DerivedClassTwo arranged in a three-level inheritance hierarchy.

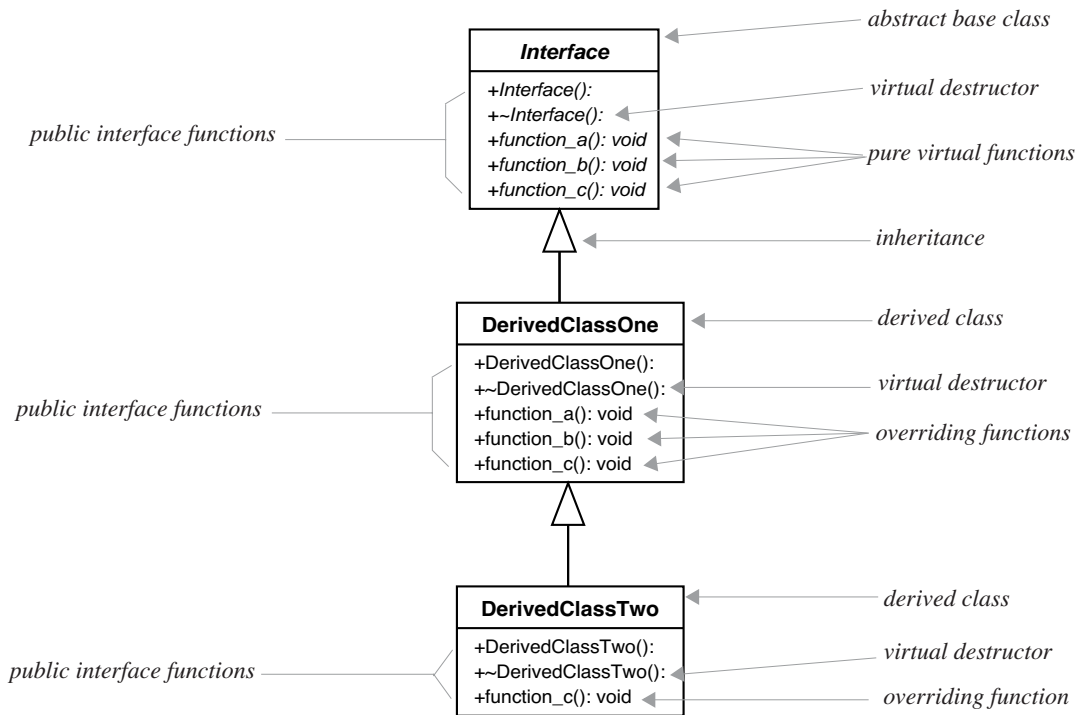


Figure 16-2: Class Diagram Showing Three-Level Inheritance Hierarchy

Class Interface is an abstract base class because it contains three pure virtual functions: function_a(), function_b(), and function_c(). It also contains a virtual destructor as well as a constructor. The code for the class Interface declaration is given in example 16.1.

```

1  #ifndef INTERFACE_H
2  #define INTERFACE_H
3  #include <iostream>
4  using namespace std;
5
6
7  class Interface {
8  public:
9      Interface(){cout<<"Interface constructor called!"<<endl;}
10     virtual ~Interface(){cout<<"Interface destructor called!"<<endl;}
11     virtual void function_a() = 0;
12     virtual void function_b() = 0;
13     virtual void function_c() = 0;
14 };
15 #endif

```

16.1 interface.h

In this example both the constructor and destructor are inline functions that contain execution trace messages. I did this to minimize the number of files required for this example. Notice that the destructor is declared virtual and that function_a(), function_b(), and function_c() are declared to be pure virtual functions by being assigned the value

of zero.

Example 16.2 gives the code for class `DerivedClassOne`.

```

1 #ifndef DERIVED_CLASS_ONE_H
2 #define DERIVED_CLASS_ONE_H
3 #include "interface.h"
4 #include <iostream>
5 using namespace std;
6
7 class DerivedClassOne : public Interface {
8     public:
9         DerivedClassOne(){cout<<"DerivedClassOne constructor called!"<<endl;}
10        virtual ~DerivedClassOne(){cout<<"DerivedClassOne destructor called!"<<endl;}
11        void function_a(){cout<<"DerivedClassOne function_a() called!"<<endl;}
12        void function_b(){cout<<"DerivedClassOne function_b() called!"<<endl;}
13        void function_c(){cout<<"DerivedClassOne function_c() called!"<<endl;}
14    };
15 #endif

```

16.2 derived_class_one.h

`DerivedClassOne` publicly inherits the behavior of class `Interface`. It then provides its own destructor and overrides `function_a()`, `function_b()`, and `function_c()`. Notice there is no need to explicitly redeclare these three functions as virtual. This is because functions declared as virtual in a base class remain virtual functions throughout the inheritance hierarchy.

Example 16.3 gives the code for `DerivedClassTwo`.

```

1 #ifndef DERIVED_CLASS_TWO_H
2 #define DERIVED_CLASS_TWO_H
3 #include "derived_class_one.h"
4 #include <iostream>
5 using namespace std;
6
7 class DerivedClassTwo : public DerivedClassOne {
8     public:
9         DerivedClassTwo(){cout<<"DerivedClassTwo constructor called!"<<endl;}
10        virtual ~DerivedClassTwo(){cout<<"DerivedClassTwo destructor called!"<<endl;}
11        void function_c(){cout<<"DerivedClassTwo function_c() called!"<<endl;}
12    };
13 #endif

```

16.3 derived_class_two.h

`DerivedClassTwo` publicly inherits the behavior of `DerivedClassOne`. It provides a virtual destructor and overrides `function_c()`.

Example 16.4 gives the code for a `main()` function used to test these three classes. On line 9 an `Interface` base class pointer is declared and initialized to point to a `DerivedClassOne` object. Lines 10 through 12 call each of the functions `function_a()`, `function_b()`, and `function_c()` via the base class pointer. The pointer is deleted on line 13 to free up memory.

Lines 16 through 19 repeat essentially the same steps but this time around a `DerivedClassTwo` object is instantiated and its address is assigned to the base class pointer.

Line 21 begins a while loop that demonstrates how derived class objects can be created “on the fly” during program execution and accessed via a base class pointer.

Figure 16-3 shows the results of running example 16.4. Follow the trace messages for the constructors and destructors of each class along with `function_a()`, `function_b()`, and `function_c()` and note their corresponding location in the source code.

Quick Review

An abstract base class provides one or more pure virtual functions that must be implemented somewhere in the inheritance hierarchy. A derived class can provide an implementation for a pure virtual function declared in its base class or it might not, in which case the derived class becomes an abstract base class. This will be demonstrated in the next section. Virtual destructors are necessary to ensure the proper destruction of objects in the inheritance hierarchy. A compiler-supplied destructor is not a virtual destructor.

16.4 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "interface.h"
4  #include "derived_class_one.h"
5  #include "derived_class_two.h"
6
7  int main(){
8
9      Interface* i_ptr = new DerivedClassOne();
10     i_ptr->function_a();
11     i_ptr->function_b();
12     i_ptr->function_c();
13     delete i_ptr;
14
15     i_ptr = new DerivedClassTwo();
16     i_ptr->function_a();
17     i_ptr->function_b();
18     i_ptr->function_c();
19     delete i_ptr;
20
21     while(true){
22         cout<<"Enter 1 for DerivedClassOne object; 2 for DerivedClassTwo object; anything else to exit: ";
23         char c;
24         cin>>c;
25         switch(c){
26             case '1': i_ptr = new DerivedClassOne();
27                       i_ptr->function_a();
28                       i_ptr->function_b();
29                       i_ptr->function_c();
30                       delete i_ptr;
31                       break;
32             case '2': i_ptr = new DerivedClassTwo();
33                       i_ptr->function_a();
34                       i_ptr->function_b();
35                       i_ptr->function_c();
36                       delete i_ptr;
37                       break;
38             default: exit(0);
39         }
40     }
41 }

```

```

Virtual_Functions.out.out
Interface constructor called!
DerivedClassOne constructor called!
DerivedClassOne function_a() called!
DerivedClassOne function_b() called!
DerivedClassOne function_c() called!
DerivedClassOne destructor called!
Interface destructor called!
Interface constructor called!
DerivedClassOne constructor called!
DerivedClassTwo constructor called!
DerivedClassOne function_a() called!
DerivedClassOne function_b() called!
DerivedClassTwo function_c() called!
DerivedClassTwo destructor called!
DerivedClassOne destructor called!
Interface destructor called!
Enter 1 for DerivedClassOne object; 2 for DerivedClassTwo object; anything else to exit: 1
Interface constructor called!
DerivedClassOne constructor called!
DerivedClassOne function_a() called!
DerivedClassOne function_b() called!
DerivedClassOne function_c() called!
DerivedClassOne destructor called!
Interface destructor called!
Enter 1 for DerivedClassOne object; 2 for DerivedClassTwo object; anything else to exit: 2
Interface constructor called!
DerivedClassOne constructor called!
DerivedClassTwo constructor called!
DerivedClassOne function_a() called!
DerivedClassOne function_b() called!
DerivedClassTwo function_c() called!
DerivedClassTwo destructor called!
DerivedClassOne destructor called!
Interface destructor called!
Enter 1 for DerivedClassOne object; 2 for DerivedClassTwo object; anything else to exit: 3

```

Figure 16-3: Results of Running Example 16.4

EXTENDED EXAMPLE: ENGINE COMPONENTS REVISITED

This section presents an extended code example that illustrates the use of dynamic polymorphic behavior in system design. My goal here is to show you how designing with dynamic polymorphism in mind enables you to treat classes of objects and sub objects orthogonally.

A BASIS FOR COMPARISON

Figure 16-4 gives the UML class diagram for the aircraft engine components model originally presented in chapter 12.

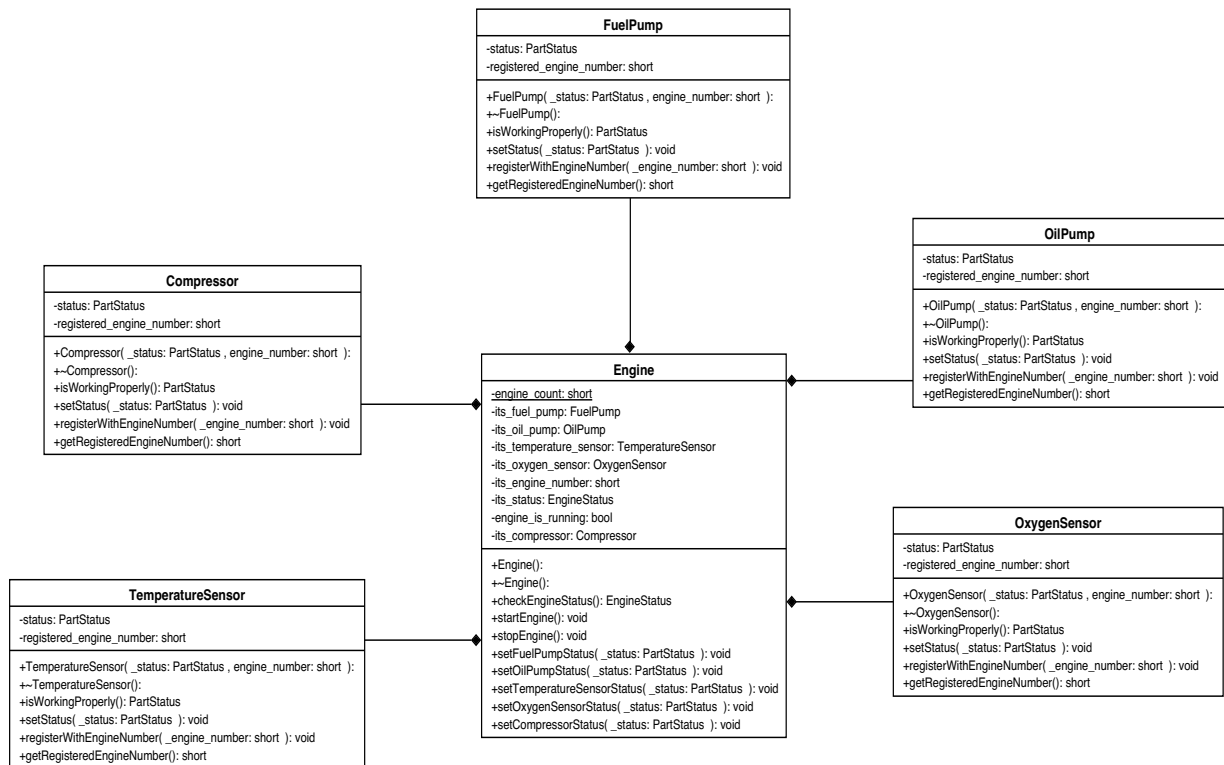


Figure 16-4: Original Aircraft Engine Components Model

In this model, the Engine class contained various types of engine components, each declared in a separate class with no relation to each other although they all declared a similar public interface. We can capitalize on the concept of interface similarity by rethinking the design in terms of dynamic polymorphic behavior. Contrast and compare this model with the UML class diagram presented in figure 16-5.

Referring to figure 16-5, the IComponent class sits at the top of the inheritance hierarchy and provides a set of pure virtual interface functions. Class Component inherits directly from IComponent and implements some of the generalized behavior common to all engine components. Note that Component is an abstract base class because although it inherits the getComponentType() virtual function from IComponent it leaves its implementation to its subclasses.

The Pump and Sensor classes inherit from Component and extend its behavior accordingly. Pump and Sensor are abstract base classes because they inherit getComponentType() from Component but leave its implementation to their subclasses.

The classes WaterPump, OilPump, FuelPump, AirFlowSensor, OxygenSensor, and TemperatureSensor are the concrete components that will actually be instantiated when it comes time to construct a SmallEngine object.

The Engine class is an abstract base class that stands alone to provide an interface for its derived classes. In this



Figure 16-5: UML Class Diagram Showing Polymorphic Engine Components

example, there is only one Engine subclass named SmallEngine. The SmallEngine class contains three Pumps and three Sensors. By using base class pointers of type Pump and Sensor the design of the SmallEngine class is simplified in that all Pump and Sensor behavior can be address via a common interface. Any behavior common to all engine components can be accessed via the IComponent interface.

POLYMORPHIC ENGINE COMPONENT CODE

The code for the revised polymorphic engine component example is given below:

icomponent.h

icomponent.h

```
#ifndef I_COMPONENT_H
#define I_COMPONENT_H
#include "engineutils.h"
#ifdef MESSAGE_TRACE
    #include <iostream>
    using namespace std;
#endif

class IComponent{
public:
    IComponent(){
#ifdef MESSAGE_TRACE
        cout<<"IComponent Object Created!"<<endl;
#endif
    }
    virtual ~IComponent(){
#ifdef MESSAGE_TRACE
        cout<<"IComponent Object Destroyed!"<<endl;
#endif
    }
    virtual bool isWorkingProperly() = 0;
    virtual void setStatus(PartStatus the_status) = 0;
    virtual PartStatus getStatus() = 0;
    virtual void registerWithEngineNumber(short the_engine_number) = 0;
    virtual short getRegisteredEngineNumber() = 0;
    virtual char* getComponentType() = 0;
    virtual int getComponentCount() = 0;
    virtual void enable() = 0;
    virtual void disable() = 0;
};
#endif
```

component.h

component.h

```
#include "icomponent.h"

class Component : public IComponent{
public:
    Component(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~Component();
    virtual void setStatus(PartStatus the_status);
    virtual PartStatus getStatus();
    virtual void registerWithEngineNumber(short the_engine_number);
    virtual short getRegisteredEngineNumber();
    virtual int getComponentCount();
    virtual bool isWorkingProperly();
    virtual void enable();
    virtual void disable();

private:
    PartStatus its_status;
    short registered_engine_number;
    static int component_count;
};
#endif
```

COMPONENT.CPP*component.cpp*

```

#include "component.h"
#include "engineutils.h"

int Component::component_count = 0;

Component::Component(PartStatus the_status, short the_engine_number):its_status(the_status),
    registered_engine_number(the_engine_number) {
    component_count++;
    #if MESSAGE_TRACE
    cout<<"Component Object Created!"<<endl;
    #endif
}

Component::~Component(){
    component_count--;
    #if MESSAGE_TRACE
    cout<<"Component Object Destroyed!"<<endl;
    #endif
}

void Component::setStatus(PartStatus the_status){
    its_status = the_status;
}

PartStatus Component::getStatus(){ return its_status;}

void Component::registerWithEngineNumber(short the_engine_number){
    registered_engine_number = the_engine_number;
}

short Component::getRegisteredEngineNumber(){ return registered_engine_number;}

int Component::getComponentCount(){ return component_count;}

bool Component::isWorkingProperly(){
    bool return_val = false;
    switch(its_status){
        case WORKING :    return_val = true;
            break;
        case NOTWORKING: break;
        default: break;
    }
    return return_val;
}

void Component::enable(){ its_status = WORKING;}

void Component::disable(){ its_status = NOTWORKING;}

```

pump.h*pump.h*

```

#ifndef PUMP_H
#define PUMP_H
#include "component.h"
#include "engineutils.h"

class Pump : public Component{
public:
    Pump(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~Pump();
    virtual bool start();
    virtual bool stop();
    virtual int getPumpCount();

private:
    static int pump_count;
    enum PumpStatus {STOPPED, RUNNING};
    PumpStatus its_status;
};
#endif

```

pump.cpp*pump.cpp*

```

#include "pump.h"
#include "engineutils.h"

int Pump::pump_count = 0;

Pump::Pump(PartStatus the_status, short the_engine_number): Component(the_status, the_engine_number),
    its_status(STOPPED){

    pump_count++;

    #if MESSAGE_TRACE
    cout<<"Pump Object Created!"<<endl;
    #endif
}

Pump::~Pump(){
    pump_count--;
    #if MESSAGE_TRACE
    cout<<"Pump Object Destroyed!"<<endl;
    #endif
}

bool Pump::start(){
    bool return_val = false;
    PartStatus component_status = getStatus();
    switch(component_status){
        case WORKING : its_status = RUNNING;
            return_val = true;
            break;
        case NOTWORKING : its_status = STOPPED;
            break;
        default : break;
    }
    return return_val;
}

bool Pump::stop(){
    bool return_val = false;
    switch(its_status){
        case RUNNING : its_status = STOPPED;
            return_val = true;
            break;
        default : break;
    }
    return return_val;
}

int Pump::getPumpCount(){ return pump_count;}

```

sensor.h*sensor.h*

```

#ifndef SENSOR_H
#define SENSOR_H
#include "component.h"
#include "engineutils.h"

class Sensor : public Component {
public:
    Sensor(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~Sensor();
    virtual void calibrate();
    virtual void unCalibrate();
    virtual void enableSensing();
    virtual void disableSensing();
    virtual float getReading() = 0;
    virtual void setReading(float reading) = 0;
    virtual bool isWorkingProperly();
    virtual int getSensorCount();

private:
    static int sensor_count;
    enum SensorStatus {SENSING_ENABLED, SENSING_DISABLED};
    enum SensorCalibration {UNCALIBRATED, CALIBRATED};
};

```

```

    SensorStatus its_status;
    SensorCalibration its_calibration;

};
#endif

```

SENSOR.CPP*sensor.cpp*

```

#include "sensor.h"
#include "engineutils.h"

int Sensor::sensor_count = 0;

Sensor::Sensor(PartStatus the_status, short the_engine_number) :
    Component(the_status, the_engine_number), its_status(SENSING_ENABLED),
    its_calibration(CALIBRATED){
    sensor_count++;
    #if MESSAGE_TRACE
        cout<<"Sensor Object Created!"<<endl;
    #endif
}

Sensor::~Sensor(){
    sensor_count--;
    #if MESSAGE_TRACE
        cout<<"Sensor Object Destroyed!"<<endl;
    #endif
}

void Sensor::enableSensing(){its_status = SENSING_ENABLED;}

void Sensor::disableSensing(){its_status = SENSING_DISABLED;}

void Sensor::calibrate(){ its_calibration = CALIBRATED;}

void Sensor::unCalibrate(){its_calibration = UNCALIBRATED;}

bool Sensor::isWorkingProperly(){
    bool return_val = false;
    switch(its_calibration){
        case CALIBRATED :    switch(its_status){
                                case SENSING_ENABLED : setStatus(WORKING);
                                                                return_val = true;
                                                                break;
                                case SENSING_DISABLED : setStatus(NOTWORKING);
                                                                return_val = false;
                                                                break;
                                default: break;
                            }
        case UNCALIBRATED : setStatus(NOTWORKING);
                            disableSensing();
                            break;
        default: break;
    }
    return return_val;
}

int Sensor::getSensorCount(){ return sensor_count;}

```

WATERPUMP.H*waterpump.h*

```

#ifndef WATER_PUMP_H
#define WATER_PUMP_H
#include "pump.h"
#include "engineutils.h"

class WaterPump : public Pump {
public:
    WaterPump(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~WaterPump();
    virtual char* getComponentType();
};
#endif

```


WATERPUMP.CPP*waterpump.cpp*

```

#include "waterpump.h"
#include "engineutils.h"

WaterPump::WaterPump(PartStatus the_status, short the_engine_number) :
Pump(the_status, the_engine_number){
    #if MESSAGE_TRACE
        cout<<"WaterPump Object Created!"<<endl;
    #endif
}

WaterPump::~WaterPump(){
    #if MESSAGE_TRACE
        cout<<"WaterPump Object Destroyed!"<<endl;
    #endif
}

char* WaterPump::getComponentType(){ return "Water Pump";}

```

oilpump.h*oilpump.h*

```

#ifndef OIL_PUMP_H
#define OIL_PUMP_H
#include "pump.h"
#include "engineutils.h"

class OilPump : public Pump {
public:
    OilPump(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~OilPump();
    virtual char* getComponentType();
};
#endif

```

oilpump.cpp*oilpump.cpp*

```

#include "oilpump.h"
#include "engineutils.h"

OilPump::OilPump(PartStatus the_status, short the_engine_number) :
Pump(the_status, the_engine_number){
    #if MESSAGE_TRACE
        cout<<"OilPump Object Created!"<<endl;
    #endif
}

OilPump::~OilPump(){
    #if MESSAGE_TRACE
        cout<<"OilPump Object Destroyed!"<<endl;
    #endif
}

char* OilPump::getComponentType(){ return "Oil Pump";}

```

fuelpump.h*fuelpump.h*

```

#ifndef FUEL_PUMP_H
#define FUEL_PUMP_H
#include "pump.h"
#include "engineutils.h"

class FuelPump : public Pump{
public:
    FuelPump(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~FuelPump();
    virtual char* getComponentType();
};
#endif

```

fuelpump.cpp*fuelpump.cpp*

```

#include "fuelpump.h"
#include "engineutils.h"

FuelPump::FuelPump(PartStatus the_status, short the_engine_number) :
    Pump(the_status, the_engine_number) {
    #if MESSAGE_TRACE
    cout<<"FuelPump Object Created!"<<endl;
    #endif
}

FuelPump::~FuelPump() {
    #if MESSAGE_TRACE
    cout<<"FuelPump Object Destroyed!"<<endl;
    #endif
}

char* FuelPump::getComponentType() { return "Fuel Pump";}

```

AIRFLOWSENSOR.H*airflowsensor.h*

```

#ifndef AIRFLOW_SENSOR_H
#define AIRFLOW_SENSOR_H
#include "engineutils.h"
#include "sensor.h"

class AirFlowSensor : public Sensor {
public:
    AirFlowSensor(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~AirFlowSensor();
    virtual char* getComponentType();
    virtual void setReading(float reading);
    virtual float getReading();
private:
    float its_reading;
};
#endif

```

AIRFLOWSENSOR.CPP*airflowsensor.cpp*

```

#include "airflowsensor.h"
#include "engineutils.h"

AirFlowSensor::AirFlowSensor(PartStatus the_status, short the_engine_number) :
    Sensor(the_status, the_engine_number), its_reading(0) {
    #if MESSAGE_TRACE
    cout<<"AirFlowSensor Object Created!"<<endl;
    #endif
}

AirFlowSensor::~AirFlowSensor() {
    #if MESSAGE_TRACE
    cout<<"AirFlowSensor Object Destroyed!"<<endl;
    #endif
}

char* AirFlowSensor::getComponentType() { return "AirFlow Sensor";}

float AirFlowSensor::getReading() {return its_reading;}

void AirFlowSensor::setReading(float reading) { its_reading = reading;}

```

OXYGENSENSOR.H*oxygensensor.h*

```

#ifndef OXYGEN_SENSOR_H
#define OXYGEN_SENSOR_H
#include "engineutils.h"
#include "sensor.h"

```

```

class OxygenSensor : public Sensor {
public:
    OxygenSensor(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~OxygenSensor();
    virtual char* getComponentType();
    virtual float getReading();
    virtual void setReading(float reading);
private:
    float its_reading;
};
#endif

```

oxygensensor.cpp*oxygensensor.cpp*

```

#include "oxygensensor.h"
#include "engineutils.h"

OxygenSensor::OxygenSensor(PartStatus the_status, short the_engine_number) :
    Sensor(the_status, the_engine_number), its_reading(0) {
    #if MESSAGE_TRACE
        cout<<"OxygenSensor Object Created!"<<endl;
    #endif
}

OxygenSensor::~OxygenSensor() {
    #if MESSAGE_TRACE
        cout<<"OxygenSensor Object Destroyed!"<<endl;
    #endif
}

char* OxygenSensor::getComponentType() { return "Oxygen Sensor";}

float OxygenSensor::getReading() {return its_reading;}

void OxygenSensor::setReading(float reading) { its_reading = reading;}

```

TEMPERATURESENSOR.H*temperaturesensor.h*

```

#ifndef TEMPERATURE_SENSOR_H
#define TEMPERATURE_SENSOR_H
#include "engineutils.h"
#include "sensor.h"

class TemperatureSensor : public Sensor {
public:
    TemperatureSensor(PartStatus the_status = WORKING, short the_engine_number = 0);
    virtual ~TemperatureSensor();
    virtual char* getComponentType();
    virtual float getReading();
    virtual void setReading(float reading);
private:
    float reading;
};
#endif

```

TEMPERATURESENSOR.CPP*temperaturesensor.cpp*

```

#include "temperaturesensor.h"
#include "engineutils.h"

TemperatureSensor::TemperatureSensor(PartStatus the_status, short the_engine_number) :
    Sensor(the_status, the_engine_number), reading(0) {
    #if MESSAGE_TRACE
        cout<<"TemperatureSensor Object Created!"<<endl;
    #endif
}

TemperatureSensor::~TemperatureSensor() {

```

```

    #if MESSAGE_TRACE
    cout<<"TemperatureSensor Object Destroyed!"<<endl;
    #endif
}

char* TemperatureSensor::getComponentType(){ return "Temperature Sensor";}

float TemperatureSensor::getReading(){ return reading;}

void TemperatureSensor::setReading(float reading){
this->reading = reading;
}

```

ENGINE.H*engine.h*

```

#ifndef ENGINE_H
#define ENGINE_H
#include "engineutils.h"

class Engine {
public:
    Engine();
    virtual ~Engine();
    short getEngineCount();
    short getEngineNumber();
    EngineStatus getEngineStatus();
    void setEngineStatus(EngineStatus the_status);
    virtual void enableComponents() = 0;
    virtual bool checkReady() = 0;
    virtual bool startEngine() = 0;
    virtual bool stopEngine() = 0;

private:
    static short engine_count;
    short engine_number;
    EngineStatus its_status;
};
#endif

```

ENGINE.CPP*engine.cpp*

```

#include "engine.h"
#if MESSAGE_TRACE
#include <iostream>
using namespace std;
#endif

short Engine::engine_count = 1;

Engine::Engine():engine_number(engine_count++), its_status(NOTREADY){
    #if MESSAGE_TRACE
    cout<<"Engine Object Created!"<<endl;
    #endif
}

Engine::~~Engine(){
    #if MESSAGE_TRACE
    cout<<"Engine Object Destroyed!"<<endl;
    #endif
}

short Engine::getEngineCount(){return engine_count;}

short Engine::getEngineNumber(){return engine_number;}

EngineStatus Engine::getEngineStatus(){return its_status;}

void Engine::setEngineStatus(EngineStatus the_status){its_status = the_status;}

```

SMALLENGINE.H*smallengine.h*

```

#ifndef SMALL_ENGINE
#define SMALL_ENGINE
#include "engine.h"
#include "icomponent.h"
#include "sensor.h"
#include "pump.h"
#include "fuelpump.h"
#include "waterpump.h"
#include "oilpump.h"
#include "oxygensensor.h"
#include "airflowsensor.h"
#include "temperaturesensor.h"

class SmallEngine : public Engine {
public:
    SmallEngine();
    virtual ~SmallEngine();
    virtual void enableComponents();
    virtual bool checkReady();
    virtual bool startEngine();
    virtual bool stopEngine();

private:
    Pump* its_fuelpump;
    Pump* its_waterpump;
    Pump* its_oilpump;
    Sensor* its_oxygensensor;
    Sensor* its_airflowsensor;
    Sensor* its_temperaturesensor;
};
#endif

```

SMALLENGINE.CPP*smallengine.cpp*

```

#include "smallengine.h"

SmallEngine::SmallEngine(){
    its_fuelpump = new FuelPump(WORKING, getEngineNumber());
    its_waterpump = new WaterPump(WORKING, getEngineNumber());
    its_oilpump = new OilPump(WORKING, getEngineNumber());
    its_oxygensensor = new OxygenSensor(WORKING, getEngineNumber());
    its_airflowsensor = new AirFlowSensor(WORKING, getEngineNumber());
    its_temperaturesensor = new TemperatureSensor(WORKING, getEngineNumber());
    #if MESSAGE_TRACE
        cout<<"SmallEngine Object Created!"<<endl;
    #endif
}

SmallEngine::~SmallEngine(){
    delete its_fuelpump;
    delete its_waterpump;
    delete its_oilpump;
    delete its_oxygensensor;
    delete its_airflowsensor;
    delete its_temperaturesensor;
    #if MESSAGE_TRACE
        cout<<"SmallEngine Object Destroyed!"<<endl;
    #endif
}

void SmallEngine::enableComponents(){
    its_fuelpump->enable();
    its_waterpump->enable();
    its_oilpump->enable();
    its_oxygensensor->enable();
    its_airflowsensor->enable();
    its_temperaturesensor->enable();
}

bool SmallEngine::checkReady(){
    bool return_val = false;

```


DISCUSSION OF THE POLYMORPHIC ENGINE COMPONENT CODE

A combination of pure-virtual, virtual, and non-virtual functions is used to achieve various types of inheritance behavior. These types of behavior are discussed below.

ICOMPONENT AND DERIVED CLASSES

The IComponent abstract base class declares a set of engine component interface functions. Since it is an abstract base class its intended use in this design is solely as a base class from which derived classes inherit IComponent's public interface.

The Component class inherits IComponent's interface and implements much of the generalized component behavior. However, Component fails to implement the `GetComponentType()` function and therefore becomes an abstract base class.

The classes Pump and Sensor inherit from Component. Pump extends the functionality of Component and adopts the generalized behavior as implemented in Component. The Sensor class also extends the functionality of Component but overrides the `isWorkingProperly()` function because of the specialized behavior required of Sensor components. However, since Pump and Sensor fail to implement the `GetComponentType()` function they too become abstract base classes.

The remaining concrete Component classes inherit from either Pump or Sensor and implement the `GetComponentType()` function.

WHAT IS MEANT BY A PURE VIRTUAL vs. A VIRTUAL MEMBER FUNCTION DECLARATION.

Since IComponent declares all of its interface functions to be pure virtual, with the exception of the destructor and constructor, it dictates that subclasses must override them eventually. Compare this with the expectation of an ordinary virtual function declaration. When the Component class overrides IComponent's pure virtual functions the new functions remain virtual, but now, since an implementation for them exists in Component, subclasses of Component may choose to live with the implementation they inherit or provide their own specialized behavior through overriding.

ENGINE AND SMALLENGINE

The Engine class declares several non-virtual functions in addition to some pure virtual functions. The non-virtual functions will be inherited by Engine subclasses but they cannot be overridden. Engine subclasses may, of course, redeclare any of the non-virtual functions the Engine class declares if the design so dictates, however, polymorphic behavior will not be achieved when those derived class objects are referenced by a Engine base class pointer.

The pure virtual functions declared by Engine must be implemented somewhere. In this example they are implemented in the SmallEngine class.

The SmallEngine class is an aggregate class comprised of three pumps and three sensors. The containment is by value, as indicated by the solid diamonds on the association lines in figure 16-5, because the lifetime of each component object is controlled by a SmallEngine object. The components come into existence when a SmallEngine object is created and they are destroyed when a SmallEngine object is destroyed.

RUNNING THE POLYMORPHIC ENGINE COMPONENT PROGRAM

The `main()` function code listed on the previous page declares an Engine base class pointer, creates an instance of SmallEngine, and then checks the readiness status of the engine by sending a `checkReady()` message. If the `checkReady()` message returns true the engine is started and a message saying so is printed to the terminal.

If the `checkReady()` message fails an alternate message is sent to the terminal. The `checkReady()` message can fail if any of engine's components are not properly initialized or are not working properly. Fault conditions can be inserted into individual components although this functionality is not demonstrated or tested in the `main()` function.

Figure 16-6 shows the results of running the polymorphic engine component program.

```

Poly_Engine_Components.out.out
Engine Object Created!
IComponent Object Created!
Component Object Created!
Pump Object Created!
FuelPump Object Created!
IComponent Object Created!
Component Object Created!
Pump Object Created!
WaterPump Object Created!
IComponent Object Created!
Component Object Created!
Pump Object Created!
OilPump Object Created!
IComponent Object Created!
Component Object Created!
Sensor Object Created!
OxygenSensor Object Created!
IComponent Object Created!
Component Object Created!
Sensor Object Created!
AirFlowSensor Object Created!
IComponent Object Created!
Component Object Created!
TemperatureSensor Object Created!
SmallEngine Object Created!
Engine Started!
FuelPump Object Destroyed!
Pump Object Destroyed!
Component Object Destroyed!
IComponent Object Destroyed!
WaterPump Object Destroyed!
Pump Object Destroyed!
Component Object Destroyed!
IComponent Object Destroyed!
OilPump Object Destroyed!
Pump Object Destroyed!
Component Object Destroyed!
IComponent Object Destroyed!
OxygenSensor Object Destroyed!
Sensor Object Destroyed!
Component Object Destroyed!
IComponent Object Destroyed!
AirFlowSensor Object Destroyed!
Sensor Object Destroyed!
Component Object Destroyed!
IComponent Object Destroyed!
TemperatureSensor Object Destroyed!
Sensor Object Destroyed!
Component Object Destroyed!
IComponent Object Destroyed!
SmallEngine Object Destroyed!
Engine Object Destroyed!

```

Figure 16-6: Results of Running Polymorphic Engine Program

A SHORT STORY

Early one morning some auto mechanics were gathered around a late model car that had manifested a few performance issues. They were talking among themselves offering different opinions as to what the problem could be when suddenly the shop supervisor walked up and yelled, “OK everybody, back to work!”

On the back to work command one mechanic went back to tuning an engine, another mechanic started cleaning the garage bays, another started to rotate some tires, while yet another mechanic started to replace an exhaust system.

These shop mechanics exhibited polymorphic behavior; one back to work command caused each to start performing a different type of work.

TAMING THE COMPLEXITY OF THE C++ LANGUAGE

To implement inheritance and dynamic polymorphic behavior requires the concurrent use of many complex C++ language features. You will master the language soon enough, but until you do it will be more helpful to think in terms of the inheritance behavior you desire from your design. Table 16-2 lists and describes the different types of inheritance behavior achieved with different C++ language features.

Feature	Behavior
<i>base class</i>	A base class provides generalized behavior. Subclasses that share a common base class will exhibit the common behavior supplied by the base class.
<i>pure virtual function</i>	A pure virtual function is an interface function that must be inherited and implemented. A subclass that inherits a pure virtual function but fails to provide an overriding implementation will become an abstract base class.
<i>ordinary virtual function</i>	A class inheriting an ordinary virtual function may accept the virtual function's implementation or provide an overriding behavior.
<i>non-virtual function</i>	A base class that declares and implements a non-virtual function prevents derived classes from overriding that function.
<i>public inheritance</i>	Public inheritance means the derived class is implementing "is a..." behavior.
<i>private inheritance</i>	Private inheritance will block client access via the derived class to any public member functions in the base class. When private inheritance is used there is no conceptual relationship between the base and derived class since the base class's public functions are now private in the derived class.

Table 16-2: Language Features vs. Inheritance Behavior

SUMMARY

Dynamic polymorphism is a powerful concept that lets you treat related objects orthogonally via a common inherited interface. The language features that enable dynamic polymorphic behavior include abstract base classes, pure virtual functions, public inheritance, derived classes, derived class objects, function overriding, and base class pointers to derived class objects.

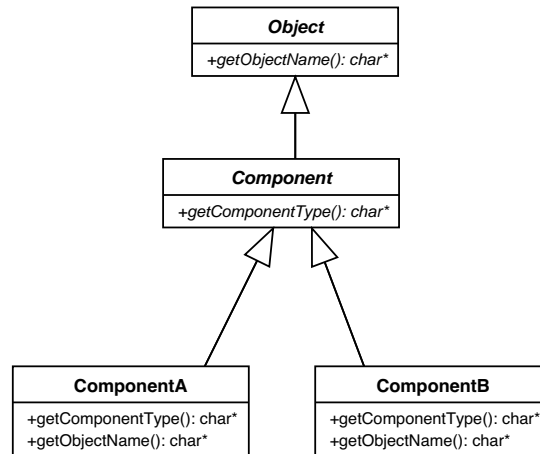
An abstract base class provides one or more pure virtual functions that must be implemented somewhere in the inheritance hierarchy. A derived class can provide an implementation for a pure virtual function declared in its base class or it might not, in which case the derived class becomes an abstract base class.

Virtual destructors are necessary to ensure the proper destruction of objects in the inheritance hierarchy. A compiler-supplied destructor is not a virtual destructor.

The C++ language is very complex. Until you master its rules it will be helpful to think in terms of language features and the behaviors resulting from their use.

Skill Building Exercises

- Research:** Procure a copy of Stanley B. Lippman's excellent book *Inside The C++ Object Model* and read it. Draw a diagram showing how C++ uses virtual tables to implement dynamic polymorphic behavior.
- Reasoning About Polymorphic Behavior:** Write a program that implements the inheritance hierarchy shown in the following diagram:



Make the `Object::getObjectName()` and `Component::getComponentType()` pure virtual functions. Implement both functions in the `ComponentA` and `ComponentB` classes. Ensure the `ComponentA` versions of the functions are implemented differently from the `ComponentB` versions.

Write a `main()` function that creates an `Object` pointer and a `Component` pointer. Dynamically create a new `ComponentA` object and assign it to the `Object` pointer, and create a `ComponentB` object and assign it to the `Component` pointer.

Answer the following questions before finishing the program:

- What function(s) can you call on a `Component` object via an `Object` pointer?
- What function(s) can you call on a `Component` object via a `Component` pointer?

Complete the program and test your answers.

3. Reasoning About Polymorphic Behavior: Study the following header and answer the questions that follow:

```

#ifndef ALL_CLASSES_H
#define ALL_CLASSES_H
#include <iostream>
using namespace std;

class A {
public:
    A(){cout<<"A created!"<<endl;}
    virtual ~A(){cout<<"A destroyed!"<<endl;}
    virtual void f() = 0;
};

class B : public A{
public:
    B(){cout<<"B created!"<<endl;}
    virtual ~B(){cout<<"B destroyed!"<<endl;}
};

class C : public B{
public:
    C(){cout<<"C created!"<<endl;}
    virtual ~C(){cout<<"C destroyed!"<<endl;}
    virtual void f(){cout<<"C::f()"<<endl;}
};
#endif
  
```

- a. Can an object of class B be created? Explain your answer.
- b. Can an object of class A be created? Explain your answer.
- c. Can an object of class C be created? Explain your answer.
- d. If an object of class C can be created what would be output of the following program:

```
#include <iostream>
using namespace std;
#include "allclasses.h"

int main() {

    A* aptr = new C;
    aptr->f();
    delete aptr;
    return 0;
}
```

4. **Testing Private Inheritance:** Write a program that tests the effects of private inheritance. Create a class named Base. In the Base class create a virtual destructor and a virtual function named f() and have it print a simple message to the console. Create another class named Derived that privately inherits the functionality of Base. Give the Derived class two functions: one named f() that has the same function signature as Base::f(), and g(). Have Derived::f() and Derived::g() print simple messages to the console. Create a main() function and write code to perform the following operations:

- a. Declare a Base class pointer and create a Base class object with the new operator.
- b. Call the Base::f() function via the Base class pointer. Note the results. Delete the pointer.
- c. Create a new Derived class object and assign its address to the Base class pointer. Note the results.
- d. What happens when you tried to execute step c? Why does this happen?
- e. Declare a Derived class pointer and create a Derived class object with the new operator.
- f. Call the Derived::f() function via the pointer. What are the results?
- g. Call the Derived::g() function via the pointer. What are the results?

5. **Deferring Implementation of a Pure Virtual Function:** Study the following code and answer the questions:

```
#ifndef ALL_CLASSES_H
#define ALL_CLASSES_H
#include <iostream>
using namespace std;

class A {
public:
    virtual ~A() {}
    virtual void f() = 0;
    virtual void g() = 0;
};

class B : public A{
public:
    virtual ~B() {}
    void f() {cout<<"B::f()"<<endl;}
};

class C : public B{
public:
    virtual ~C() {}
    void g() {cout<<"C::g()"<<endl;}
};
#endif
```

- a. What type of class is class A?
- b. What is the purpose of a virtual destructor?
- c. What type of class is class B? Explain your answer.
- d. Of the three classes A, B, and C, which class is considered the concrete class?

6. **Testing Protected Access:** Consider the following source code and answer the question:

```
#ifndef ALL_CLASSES_H
#define ALL_CLASSES_H
#include <iostream>
using namespace std;

class A {
public:
    virtual ~A() {}
    virtual void f() {cout<<"A::f()"<<endl;}
};

class B : public A{
public:
    virtual ~B() {}

protected:
    virtual void f() {cout<<"B::f()"<<endl;}
};
#endif
```

a. Given a pointer of type A that points to an object of type B, what message will be printed to the screen when function f() is called via the A pointer?

7. **Testing Non-Virtual Functions:** Consider the following source code and answer the questions:

```
#ifndef ALL_CLASSES_H
#define ALL_CLASSES_H
#include <iostream>
using namespace std;

class A {
public:
    virtual ~A() {}
    void f() {cout<<"A::f()"<<endl;}
};

class B : public A{
public:
    virtual ~B() {}
    void f() {cout<<"B::f()"<<endl;}
};
#endif
```

a: Given a pointer of type A that points to an object of type B, what message will be printed to the screen when function f() is called via the A pointer?

b: What change(s) to class A could be made to achieve dynamic polymorphic behavior.

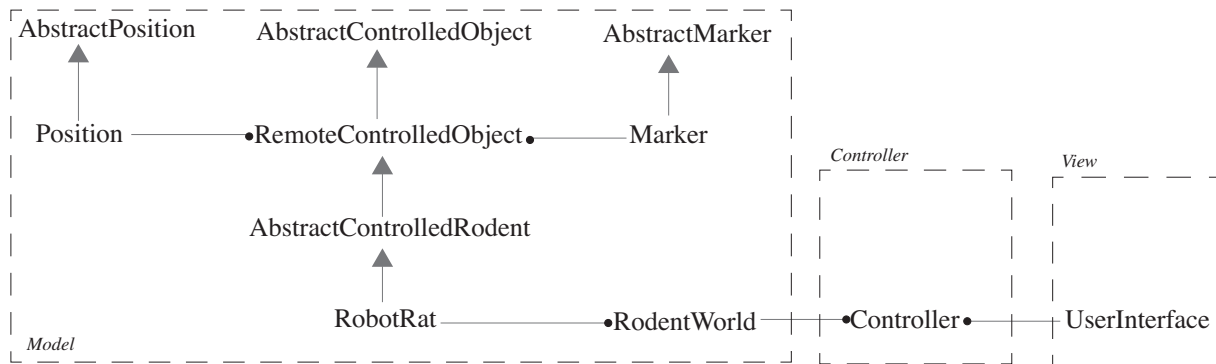
SUGGESTED PROJECTS

1. **Redesign RobotRat:** Redesign and implement the RobotRat project to take advantage of dynamic polymorphic behavior. Create an inheritance hierarchy that will allow the addition of different types of remote controlled objects.

Remote controlled objects will have a position and a marker. The position object will keep track of a remote controlled object's position on a two-dimensional floor (row and column). The position will also keep track of a remote controlled object's heading (NORTH, SOUTH, EAST, and WEST). The marker object will keep track of two possible positions (UP and DOWN).

When working up your design keep in mind the following: Functionality for robot rats could be an adaptation of a remote controlled object's behavior. For instance, a remote controlled object's marker is a robot rat's tail.

One possible inheritance hierarchy appears in the following diagram.



Additional functionality could include:

- Segregate the design of the application into three primary components: Model, View, and Controller (MVC). The MVC component organization is an important and often recurring pattern in object-oriented design. The purpose of the model is to implement the application functionality. In the above diagram, the RodentWorld class would serve as the focal point for all operations related to the management of RobotRat or other RemoteControlledObjects. The UserInterface component handles all user interface actions. The Controller component coordinates message passing between the Model and the UserInterface components.
- Users should be able to add and delete RobotRat objects.
- Users should be able to toggle between different RobotRat objects on the floor.

2. **Redesign Computer Simulation:** Redesign and implement the computer simulation project to take advantage of dynamic polymorphic behavior. Design a suitable inheritance hierarchy that will allow the creation of different processor models and memory systems.
3. **Redesign Oil Tanker:** Redesign and implement the oil tanker project presented in chapter 13, suggested project 5, to take advantage of dynamic polymorphic behavior. Design a suitable inheritance hierarchy that will allow the creation of different types of pumps and valves.
4. **Design Satellite Control System:** Design and implement a satellite control system. Your system should allow the creation and control of different types of satellites.

SELF TEST QUESTIONS

1. Describe in your own words what is meant by the term dynamic polymorphism.
2. How does dynamic polymorphism differ from ad hoc and static polymorphism?
3. What is a pure virtual function?
4. How is a pure virtual function different from an ordinary virtual function?
5. A base class that declares one or more pure virtual functions is known as what type of class?
6. What happens to a derived class if it inherits a pure virtual function but fails to override it?
7. Describe the three different inheritance behaviors achieved through the use of pure virtual, ordinary virtual, and

non-virtual functions.

8. Consider the relationship between a base class and a derived class. What type of behavior should the base class implement as compared to the derived class's behavior?
9. How does public inheritance differ from private inheritance?
10. What happens to public base class functions when a derived class privately inherits them?

REFERENCES

Scott Meyers. *Effective C++*. Second Edition. Addison-Wesley. Reading, Massachusetts. ISBN: 0-201-92488-9

International Standard, ISO/IEC 14882, *Programming Language — C++*, First Edition 1998-09-01

Bjarne Stroustrup. *The C++ Programming Language*, Third Edition. Addison-Wesley, Reading, Massachusetts. ISBN: 0-201-88954-4

Stanley B. Lippman. *Inside The C++ Object Model*. Addison-Wesley. Reading, Massachusetts. ISBN: 0-201-83454-5

NOTES

PART IV: INTERMEDIATE CONCEPTS

CHAPTER 17



BALBOA REFLECTIONS

WELL-BEHAVED OBJECTS: THE ORTHODOX CANONICAL CLASS FORM

LEARNING OBJECTIVES

- *STATE THE IMPORTANT ROLE WELL-BEHAVED OBJECTS PLAY IN GOOD OBJECT-ORIENTED DESIGN*
- *LIST AND DESCRIBE THE FUNCTIONS REQUIRED TO GET USER DEFINED OBJECTS TO BEHAVE LIKE NATIVE TYPES*
- *LIST AND DESCRIBE THE FOUR MINIMUM FUNCTIONS REQUIRED TO IMPLEMENT THE ORTHODOX CANONICAL CLASS FORM*
- *DEMONSTRATE YOUR ABILITY TO UTILIZE THE ORTHODOX CANONICAL CLASS FORM IN YOUR C++ PROGRAMMING PROJECTS*
- *DEMONSTRATE YOUR ABILITY TO EXTEND THE ORTHODOX CANONICAL CLASS FORM TO SUIT THE NEEDS OF A PARTICULAR CLASS*
- *EXPLAIN WHY COMPILER-SUPPLIED CONSTRUCTORS AND DESTRUCTORS MAY NOT PROVIDE APPROPRIATE OBJECT BEHAVIOR FOR COMPLEX CLASS TYPES*
- *LIST AND DEFINE THE FOLLOWING TERMS: ORTHODOX CANONICAL CLASS FORM, DEFAULT CONSTRUCTOR, DESTRUCTOR, COPY ASSIGNMENT OPERATOR, COPY CONSTRUCTOR*

INTRODUCTION

Correct, predictable object behavior is essential for the successful implementation of an object-oriented software application. To achieve correct, predictable object behavior you must be keenly aware of the issues regarding the use of constructors, destructors, copy constructors, and assignment operators when creating user-defined data types in C++. You must learn to identify programming situations where a naive reliance on default compiler behavior will lead to disaster. And once you spot these situations, you must be able to provide a suitable code alternative to ensure disaster is avoided.

The material presented here builds upon and reinforces material presented in chapter 11.

WHAT IS A WELL-BEHAVED OBJECT?

A well-behaved object is one that can be relied upon to perform as if it were a native C++ data type. The degree to which an object should perform like a native data type is context dependent. For instance, a Person object may not be used in all the same situations as an int, float, or double, but in the contexts in which a Person object participates it should perform flawlessly.

OBJECT USAGE CONTEXTS

When creating user-defined types you must be aware of the object usage contexts of object creation, object copying, object assignment, and object destruction.

OBJECT CREATION

The first object usage context you must be aware of is object creation. Objects come into existence in several ways. The first and perhaps most common form of object creation is by simply declaring a static variable as the following line of code illustrates:

```
Foo f;
```

Here an object named `f` of type `Foo` is declared. If `Foo` is a user-defined type with complex data type member attributes then those complex attribute objects must also be created properly when the `Foo` object comes into existence.

Another method of creating objects is to use the `new` operator and create the object dynamically in heap memory. The following line of code gives an example:

```
Foo *foo_ptr = new Foo;
```

In this example a `Foo` object is created dynamically and its address is assigned to the `foo_ptr` pointer. The same issues apply regarding the proper creation of any complex data members `Foo` might contain.

A third, and often unconsidered, way objects are created is when they are passed to functions by value. Examine the following code snippet:

```
void functionA(Foo f);           //function declaration
Foo my_foo;                     //declare Foo object
functionA(my_foo);              //call functionA() with Foo object
```

In this example, when `functionA()` is called with the `my_foo` object, the function parameter `f` is created behind the scenes by the compiler at the time of the function call.

A fourth way objects are created is when functions return object values.

Quick Review

Objects come into existence in several ways: via static object declaration, via dynamic object creation with the new operator, when objects are passed to functions by value, and when an object is returned from a function.

Object Copying

The second object usage context is object copying. Object copying and object creation are related in that objects can be created and initialized using an existing object as a guide. Objects are created by copy when an object is passed to a function, as was discussed in the previous section, and when object values are returned from functions, also discussed above.

In the object copy context usage, an object can play one of two roles. The first is the role of the object being created; the second is the role of the object being copied.

Object Assignment

The third object usage context is object assignment. In object assignment, an existing object is being assigned the state values of another existing object. Objects participating in the object assignment context can play two roles: that of the object whose state values are being changed, and that of the object whose state values are being copied.

Object assignment is different from object copying in that object assignment deals with two existing objects while object copy deals with one existing and one new object.

With object assignment, you must be aware of the difference between a shallow copy and a deep copy. Also, the object whose state values are being changed must properly manage its currently held resources before taking on its new state.

Object Destruction

The fourth object usage context you need to be aware of is object destruction. Objects exist for a specific lifetime based on their location within a program. When an object’s lifetime expires its destructor is called. The destructor’s job is to ensure the release of any resources the object may have used during its lifetime. For example, if the object allocated any dynamic memory during its lifetime, it should release the dynamic memory upon its destruction.

Other Contexts By Design

The object usage contexts presented above represent the four basic contexts objects will participate in at a minimum. You alone, as a programmer, know the other contexts in which your user-defined objects will participate. An example might be the use of Person class objects in a sorting algorithm, where one Person object is compared in some way with other Person objects. If Person objects are to behave correctly in these comparison contexts then the Person class must be developed in a manner that supports such context participation.

The usage contexts presented above are summarized in table 17-1.

Object Usage Context	Be Aware of...
object creation	Objects are created via static declaration, via dynamic allocation using the new operator, and via copy when object values are passed to and returned from functions. When objects are created they must properly initialize their member attributes.
object copying	Objects participate in the copy context when a new object is created using an existing object as a guide. The copy context occurs primarily when objects are passed to and returned from functions. The object being created will use the existing object’s attributes to initialize its own member attributes.

Table 17-1: Object Usage Contexts

Object Usage Context	Be Aware of...
object assignment	Objects participate in the object creation context when an existing object's state is set to that of another existing object. In the assignment context, the object that's being changed must properly handle its complex data members. Thus it is necessary to understand the difference between a shallow copy and a deep copy.
object destruction	Objects participate in the object destruction context when they reach the end of their lifetime in a program. The object being destroyed must properly release any allocated resources before destruction completes.
other contexts by design	User-defined data types you create are used in addition to the four basic usage contexts.

Table 17-1: Object Usage Contexts

THE ORTHODOX CANONICAL CLASS FORM (OCCF)

The orthodox canonical class form is a recipe, if you will, for proper class design. At the very least, the OCCF specifies four required functions that you implement when declaring user-defined data types to ensure they behave correctly in the four basic object usage contexts. By implementing the OCCF in user-defined types you increase your situational awareness regarding object usage. In this capacity the OCCF is much more than a recipe for correct object behavior — it is a fundamental stepping stone toward an understanding of more advanced C++ idiomatic constructs. An intuitive understanding of the OCCF will empower you to write C++ classes that are better suited to play well in complex, object-oriented applications.

FOUR REQUIRED FUNCTIONS

To support the four basic usage contexts the OCCF specifies four required functions that should be implemented for complex, user-defined data types. These are the default constructor, the copy constructor, the copy assignment operator, and the destructor. A complex user-defined type named `Foo` will be used to demonstrate each of these special functions. Example 17.1 gives the code for the `Foo` class declaration. The `Foo` class is a complex type in that it manages a pointer during its lifetime. The purpose of its OCCF functions then is to properly manage the pointer.

```

1  #ifndef FOO_H                                17.1 foo.h
2  #define FOO_H
3
4  class Foo {
5      public:
6          Foo(int i = 0);
7          virtual ~Foo();
8          Foo(Foo& rhs);
9          Foo& operator=(Foo& rhs);
10         int getVal();
11         void setVal(int i);
12     private:
13         int* iptr;
14 };
15 #endif

```

DEFAULT CONSTRUCTOR

The purpose of a constructor is to properly create new objects. A default constructor is one that can be called with no arguments. A default constructor can declare parameters but each constructor parameter must have a default value so the constructor can be called without arguments.

The default constructor for class Foo is declared on line 6. It declares one integer parameter whose default value is zero.

DESTRUCTOR

The purpose of the destructor is to properly tear down objects when they are no longer needed and release any resources reserved for the object's use during its lifetime.

The Foo destructor is declared on line 7 of example 17.1. It is declared virtual in anticipation of supporting an inheritance hierarchy.

COPY CONSTRUCTOR

The purpose of a copy constructor is to create new objects from existing objects. The copy constructor will declare at least one parameter that is a reference type to which the copy constructor belongs.

The Foo copy constructor is declared on line 8 of example 17.1. It declared one parameter named rhs (shorthand for right hand side) that is a reference to a Foo object. (*i.e.*, *Foo&*)

COPY ASSIGNMENT OPERATOR

The purpose of a copy assignment operator is to initialize an existing object to the values supplied by another existing object.

The Foo copy assignment operator is declared on line 9 of example 17.1. It declares one parameter which is a reference to a Foo object.

IMPLEMENTING FOO CLASS OCCF FUNCTIONS

Study example 17.2 to see how each of class Foo's special functions are implemented.

```

1 #include "foo.h"
2
3 Foo::Foo(int i){ iptr = new int(i); }
4
5 Foo::~~Foo(){ delete iptr; }
6
7 Foo::Foo(Foo& rhs){ iptr = new int(*(rhs.iptr)); }
8
9 Foo& Foo::operator=(Foo& rhs){
10     *iptr = *(rhs.iptr);
11     return *this;
12 }
13
14 int Foo::getVal(){ return *iptr;}
15
16 void Foo::setVal(int i){ *iptr = i; }

```

17.2 foo.cpp

CONSIDER FUTURE DESIRED BEHAVIOR

The four special functions declared in class Foo implement the basic OCCF recipe which means Foo objects will behave predictably when they are created, copied, assigned, and destroyed. These four basic behaviors will allow Foo objects to be confidently passed as arguments to functions and returned from functions as well. If, however, Foo objects will be used in other programming contexts then other functions and operators must be supplied.

The Foo class supplies two additional functions: `getVal()` and `setVal()` to get and set the integer value to which a Foo object's integer pointer points. The `main()` function given in example 17.3 shows Foo objects being created, used, and destroyed. The header file `f.h` contains the declaration for a function named `fooFunction()` that demonstrates passing Foo objects by value into a function. Examples 17.4 and 17.5 give the code for `f.h` and `f.cpp` respectively. Figure 17-1 shows the output of running example 17.3.

```

1  #include <iostream>
2  #include "foo.h"
3  #include "f.h"
4  using namespace std;
5
6  int main() {
7      Foo f0, f1(1), f2(2);
8      cout<<f0.getVal()<<endl;
9      cout<<f1.getVal()<<endl;
10     cout<<f2.getVal()<<endl;
11     cout<<"-----"<<endl;
12     Foo f3(f2);
13     cout<<f3.getVal()<<endl;
14     f3 = f1;
15     cout<<f3.getVal()<<endl;
16     cout<<"-----"<<endl;
17     f3.setVal(3);
18     cout<<f0.getVal()<<endl;
19     cout<<f1.getVal()<<endl;
20     cout<<f2.getVal()<<endl;
21     cout<<f3.getVal()<<endl;
22     cout<<"-----"<<endl;
23     f3 = fooFunction(f2);
24     cout<<f3.getVal()<<endl;
25     return 0;
26 }
```

17.3 main.cpp

```

1  #ifndef F_H
2  #define F_H
3  class Foo;
4
5  Foo& fooFunction(Foo foo);
6
7  #endif
```

17.4 f.h

```

1  #include "f.h"
2  #include "foo.h"
3  #include <iostream>
4  using namespace std;
5
6  Foo& fooFunction(Foo foo) {
7      cout<<"fooFunction() called: "<<foo.getVal()<<endl;
8      return foo;
9  }
```

17.5 f.cpp

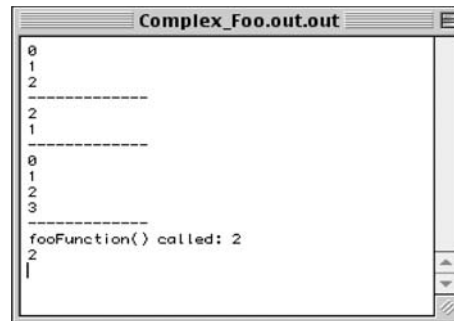


Figure 17-1: Results of Running Example 17.3

EXTENDING FOO TO PARTICIPATE IN OTHER CONTEXTS: OVERLOADING MORE OPERATORS

The key to making Foo objects behave well in their expected usage contexts is to overload the required operators. To demonstrate, I will modify class Foo so that Foo objects can be compared against each other. To help in this example I will revisit and rewrite some code originally presented in chapter 9. I will use the dumbSort() function along with the two callback functions compareAscending() and compareDescending(), rewritten as template functions. Example 17.6 gives the template versions of the dumbSort(), compareAscending(), and compareDescending() functions.

```

1  #ifndef DUMBSORT_H 17.6 dumsort.h
2  #define DUMBSORT_H (template version)
3
4  template<class T>
5  bool compareAscending(T a, T b){return a>b;}
6
7  template<class T>
8  bool compareDescending(T a, T b){return a<b;}
9
10 template<class T, class U>
11 void dumbSort(T a[], int l, int r, bool (*sortDirection)(U, U)){
12     for(int i = l; i < r; i++){
13         for(int j = (l+1); j < r; j++){
14             if(sortDirection(a[j-1], a[j])) {
15                 U temp = a[j-1];
16                 a[j-1] = a[j];
17                 a[j] = temp;
18             }
19         }
20     }
21 }
22 #endif

```

The template versions of dumbSort(), compareAscending() and compareDescending() can now be used to sort not only an array of Foo objects, but an array of any objects that can be compared using the greater-than or less-than operators. As an exercise, compare the template version of dumbSort() with the original given in example 9.53 in chapter 9. It did not require much effort to convert it into a template function. And because the Foo class was designed using the OCCF, the comparisons and assignments occurring within the body of the dumbSort() function can be relied upon to work properly without customizing them to work specifically with Foo objects.

Example 17.7 gives the modified Foo class declaration showing the addition of the required overloaded comparison operators. Example 17.8 gives their implementation. Example 17.9 gives the main() function showing the template version of dumbSort() and the template callback functions in action sorting an array of Foo objects in ascending and descending order. An important point to note about the use of the dumbSort() template function is the explicit template specialization call syntax used on lines 27 and 34 of example 17.9. Figure 17-2 shows the results of running


```

1  #ifndef FOO_H                                17.7 foo.h (modified)
2  #define FOO_H
3
4  class Foo {
5      public:
6          Foo(int i = 0);
7          virtual ~Foo();
8          Foo(Foo& rhs);
9          Foo& operator=(Foo& rhs);
10         int getVal();
11         void setVal(int i);
12         bool operator<(Foo& rhs);
13         bool operator>(Foo& rhs);
14     private:
15         int* iptr;
16 };
17 #endif

1  #include "foo.h"                                17.8 foo.cpp (modified)
2
3  Foo::Foo(int i){ iptr = new int(i); }
4
5  Foo::~~Foo(){ delete iptr; }
6
7  Foo::Foo(Foo& rhs){ iptr = new int(*(rhs.iptr)); }
8
9  Foo& Foo::operator=(Foo& rhs){
10     *iptr = *(rhs.iptr);
11     return *this;
12 }
13
14 int Foo::getVal(){ return *iptr; }
15
16 void Foo::setVal(int i){ *iptr = i; }
17
18 bool Foo::operator<(Foo& rhs){ return (*iptr) < (*rhs.iptr); }
19
20 bool Foo::operator>(Foo& rhs){ return (*iptr) > (*rhs.iptr); }

```

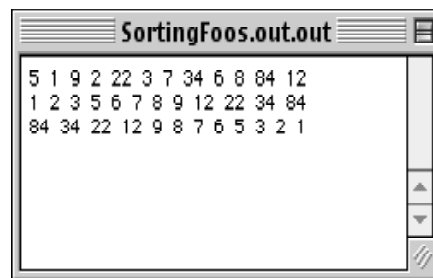


Figure 17-2: Results of Running Example 17.9

example 17.9.

Quick Review

When designing class types implement the four basic OCCF functions plus any additional functions or over-loaded operators necessary to ensure correct object behavior in anticipated object usage contexts.

```

1  #include <iostream>
2  using namespace std;
3  #include "dumbSort.h"
4  #include "foo.h"
5
6  int main(){
7      Foo foo_array[12];
8
9      foo_array[0].setVal(5);
10     foo_array[1].setVal(1);
11     foo_array[2].setVal(9);
12     foo_array[3].setVal(2);
13     foo_array[4].setVal(22);
14     foo_array[5].setVal(3);
15     foo_array[6].setVal(7);
16     foo_array[7].setVal(34);
17     foo_array[8].setVal(6);
18     foo_array[9].setVal(8);
19     foo_array[10].setVal(84);
20     foo_array[11].setVal(12);
21
22     for(int i=0; i<12; i++){
23         cout<<foo_array[i].getVal()<<" ";
24     }
25     cout<<endl;
26
27     dumbSort<Foo, Foo>(foo_array, 0, 12, compareAscending);
28
29     for(int i=0; i<12; i++){
30         cout<<foo_array[i].getVal()<<" ";
31     }
32     cout<<endl;
33
34     dumbSort<Foo, Foo>(foo_array, 0, 12, compareDescending);
35
36     for(int i=0; i<12; i++){
37         cout<<foo_array[i].getVal()<<" ";
38     }
39     cout<<endl;
40
41     return 0;
42 }

```

17.9 main.cpp

SUMMARY

Correct, predictable object behavior is essential for the successful implementation of an object-oriented software application. A well-behaved object is one that can be relied upon to perform as if it were a native C++ data type. The degree to which an object should perform like a native data type is context dependent.

When creating complex user-defined types you must be aware of the four primary object contexts: object creation, object copying, object assignment, and object destruction. The orthodox canonical class form (OCCF) can be used to ensure proper class design so that user-defined type objects behave predictably in the four basic object usage contexts.

Four special functions support the OCCF: default constructor, destructor, copy constructor, and copy assignment operator. Additional object usage contexts, such as object comparison, requires the overloading of additional operators. Keeping the OCCF in mind when you create complex user-defined types increases your object usage situational awareness.

Skill Building Exercises

1. **Research:** Procure a copy of James O. Coplien's excellent book *Advanced C++ Programming Styles and Idioms* and read it from front to back.

2. **Default Constructor and Destructor:** Given the following code for Baseline class:

```
#ifndef BASELINE_H
#define BASELINE_H

class Baseline {
public:
    virtual f();

private:
    int x;
    int y;
    char* name;

};
#endif
```

a. Add a default constructor that provides default values for Baseline attributes x, y, and name. Dynamically allocate memory for the name attribute.

b. Add a destructor that properly releases the memory allocated during the constructor call.

c. Write a driver program to test your code.

3. **Copy Constructors:** Building upon the Baseline code of the previous exercise, add a copy constructor that creates new Baseline objects from existing Baseline objects.

4. **Copy Assignment Operator:** Building upon the Baseline code resulting from the previous exercise, add a copy assignment operator to properly change the attributes of an existing Baseline object to those of another existing Baseline object. Keep in mind the difference between a shallow copy and a deep copy.

5. **Overloaded Equality Operator:** Building upon the Baseline code resulting from the previous exercise, add a equality operator so that Baseline objects can be compared with each other.

6. **Overloaded Greater-Than Operator:** Building upon the Baseline code resulting from the previous exercise, add a greater than operator so that Baseline objects can be compared with each other.

SUGGESTED PROJECTS

1. **Modify RobotRat:** Building upon the code resulting from the RobotRat project in chapter 16, implement the orthodox canonical class form for RobotRat objects. Do not forget to take into consideration the proper handling of the entire inheritance hierarchy. Write a driver program to test your code.

2. **Binary Tree:** Write a program that implements a binary tree. Each node of the binary tree should point to an object that can compare itself to other objects. The actual type of objects stored in the binary tree should be of no concern to the tree, so long as the actual types of objects inserted into the tree know how to compare themselves to other objects.

3. **Object Sorting:** Write a program that implements the sorting algorithm of your choice. The sort program will sort generic objects. Create a class of objects that can be sorted and test your program. Hint: An abstract class might publish a Sortable interface. Extend the orthodox canonical class form to implement the required sorting behavior. (*i.e., less-than, greater-than, or equality check*)

SELF TEST QUESTIONS

1. Describe in your own words why it is important to have well-behaved objects in an object-oriented software application.
2. List and describe the four basic object usage contexts.
3. What is the orthodox canonical class form?
4. What four special functions support the OCCF?
5. Why is it important to be aware of all the contexts an object will participate?
6. List several examples of object context participation that have not been discussed in this chapter.
7. List several examples of why it would be a mistake to rely on a compiler generated constructor and destructor.
8. What is the definition of a default constructor?
9. How does a copy constructor differ from a copy assignment operator?
10. What is the purpose of a destructor?

REFERENCES

James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, Massachusetts. ISBN: 0-201-54855-0

NOTES

CHAPTER 18



Park Entrance

Mixed Language Programming

Learning Objectives

- Explain how to create and integrate assembly language object modules
- Explain how to integrate legacy C code
- State the purpose and use of the `extern` keyword
- Explain why the `extern` keyword is necessary to link to legacy C code modules
- Describe the concept of name mangling
- Explain how to call C and C++ routines from Java applications
- List the steps required to create, compile, and link to an assembly language module
- List the steps required to create a Java JNI project and call a C++ native method from a Java program
- State the purpose and use of the `javah` command line tool
- Demonstrate your ability to utilize assembly language routines in your C++ programming projects
- Demonstrate your ability to call native C++ functions from Java programs
- Demonstrate your ability to use inline assembly in a Macintosh environment
- Demonstrate your ability to use inline assembly in a PC environment

INTRODUCTION

It is common for software systems to comprise software modules developed at different times in different programming languages. Sometimes this occurs by design. An example of this would be an optimized routine of some sort written in assembly language and used in a C++ program. However, it is more likely the case that a software application has “evolved” from early versions developed in one language into a later version written in a new language that still relied on legacy modules. A classic example of this is when a development team “goes object-oriented” and begins to migrate legacy application code previously developed in C to C++.

This chapter is an introduction to some of the issues surrounding mixed language programming. You will learn how to use the `extern` keyword to link functions written in C to your C++ programs, how to embed assembly code directly in a function, and how to incorporate previously developed assembly modules into your programs.

As a C++ programmer you may be tasked to write a C++ program that is called by a Java program. In this chapter you will also learn how C++ programs can be executed via the Java Native Interface (JNI). Success in this endeavor depends a lot on knowing when it is good idea to call a C++ program from Java and when it is not.

This chapter is not intended to serve as a complete treatment of the issues regarding assembly language or Java programming. Many excellent books cover these topics in great detail and a few especially good ones are listed in the references section.

C++ And C

As a C++ programmer you will encounter many situations requiring you to incorporate legacy C code into your programs. A good example of this is when you find a good C standard library function that does exactly what you need. Often, however, C++ programmers find themselves converting a legacy C application to C++. In either case, this section will explain how to use separately compiled C libraries in your programs by using the `extern` keyword, and, more importantly, why you need to use the `extern` keyword.

How C++ Allows Overloaded Functions: Name Mangling

C++ and C are different in many ways. One important difference between the two languages is that C++ allows function overloading while C does not. A C++ compiler employs a function name transformation process known as name mangling that converts function names (and other object names within a C++ program) into a form that ensures uniqueness. The C language does not support function overloading and therefore does not need to mangle names.

What does this mean to you the C++ programmer? It means that library code written in C can be used in a C++ program but you have to inform the C++ compiler that the function names of the functions contained in the C library are not mangled, or else the names of the functions appearing in the library code cannot be resolved by the linker to where they appear in the C++ source code. To help in this matter you need to use the `extern` linkage specifier.

EXTERN KEYWORD

To prevent the C++ compiler from name mangling a C function declaration use the `extern` keyword to specify the type of language linkage to perform. To illustrate how this is done I will walk you through a complete example that creates a C static library with one function named `square()`. This library will then be used in a C++ program to illustrate what happens when you try to link to the library without using `extern` and how `extern` is applied to resolve the situation.

Building A C Library: The `SQUARE()` Function

The C code library containing the `square()` function will be created using the steps shown in chapter 9 in the section Steps To Creating a Library. They are repeated here for your convenience but I must stress that the exact steps employed to create your code libraries will depend on your development environment.

- Step 1** - Put the function declarations for any functions you want in the library in a header file. (.h file)
- Step 2** - Put the definitions for the library functions in a separate implementation file. (.c file)
- Step 3** - Create an empty project in an integrated development environment and add the implementation file to it.
- Step 4** - Add any library files to the project required to support the implementation file.
- Step 5** - Set the required target settings for the project.
- Step 6** - Name the library output file and set project type.
- Step 7** - Compile the project.
- Step 8** - Use the library!

STEP 1: CREATE SQUARE.H HEADER FILE

The code for square.h is shown in example 18.1.

```

1  #ifndef SQUARE_H
2  #define SQUARE_H
3
4  double square(double d);
5
6  #endif

```

18.1 square.h

STEP 2: CREATE SQUARE.C IMPLEMENTATION FILE

The code for square.c is shown in example 18.2

```

1  #include "square.h"
2
3  double square(double d){
4      return d*d;
5  }

```

18.2 square.c

STEP 3: CREATE EMPTY PROJECT WITH IDE AND Add IMPLEMENTATION FILE

Metrowerks CodeWarrior will be used to create the empty project named Square_Lib. Figure 18-1 shows the New Project window.

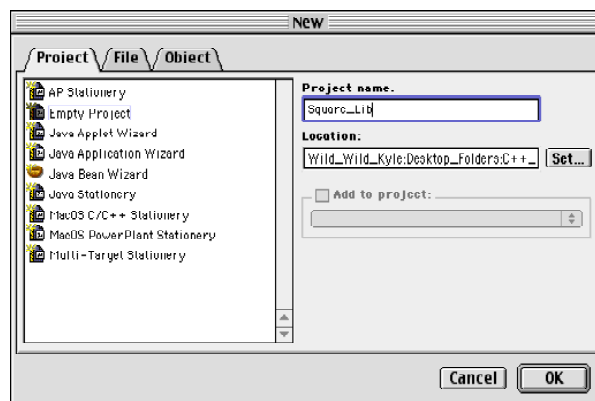


Figure 18-1: Creating a New Empty Project Named Square_Lib in CodeWarrior

This results in an empty project window as shown in figure 18-2:

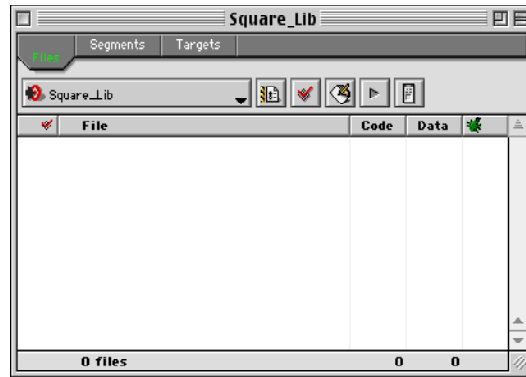


Figure 18-2: Empty Project Window

Select Add Files... from the Project menu to add the square.c file to the Square_Lib project as shown in figures 18-3 and 18-4.

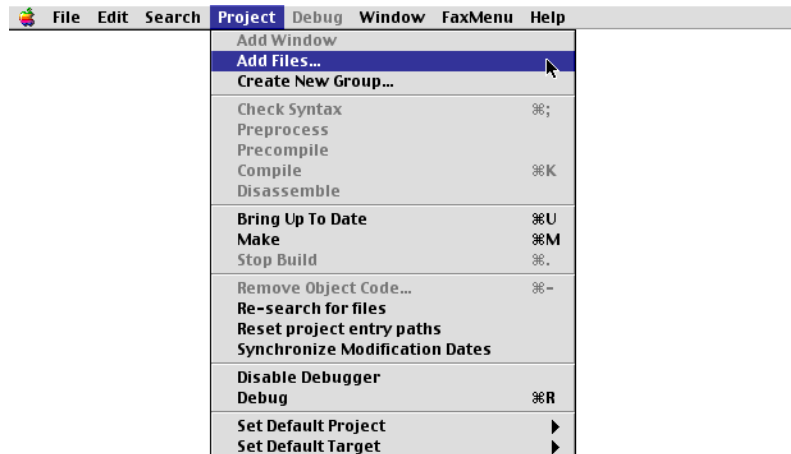


Figure 18-3: Select Add Files... from the Project Menu

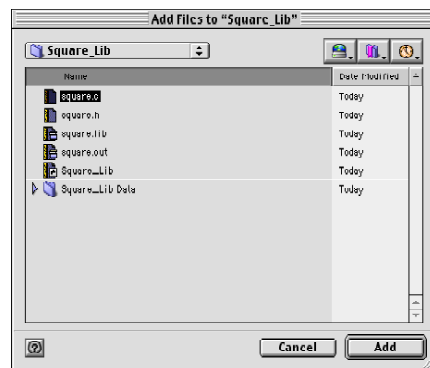


Figure 18-4: Select square.c to Add it to the Project

Figure 18-5 shows the project window after adding the square.c file.

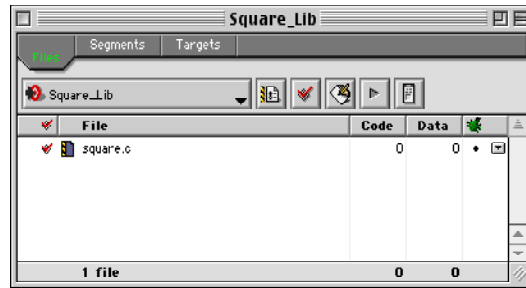


Figure 18-5: Project Window After Adding square.c

STEP 4: Add NECESSARY SUPPORTING LIBRARY FILES TO PROJECT

No supporting library files are required for the square() function.

STEP 5: SET REQUIRED TARGET SETTINGS

Because an empty project was created you must make several project settings before the square.c file can be compiled. Select Square_Lib Settings... from the Edit menu as shown in figure 18-6. This will bring up the Settings window as shown in figure 18-7.

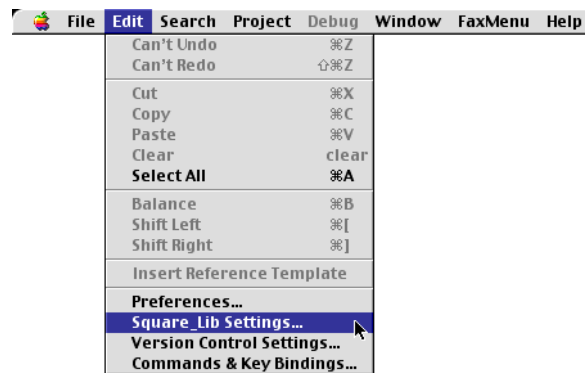


Figure 18-6: Select Square_Lib Settings... from the Edit Menu

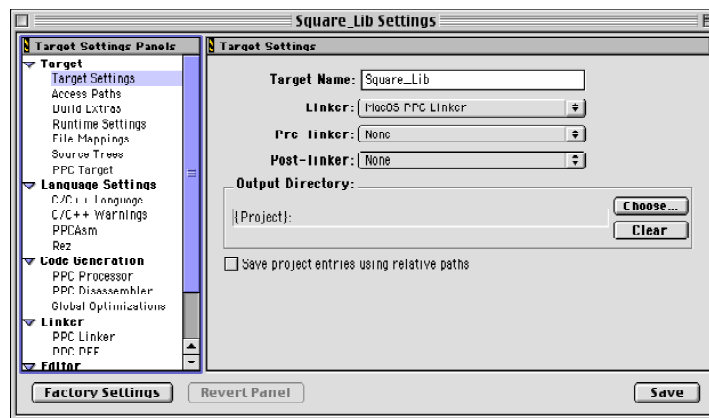


Figure 18-7: Settings Window with Target Settings Selected

The Target Settings are set as indicated in figure 18-7. The Target Name is set to Square_Lib and the Linker selected is MacOS PPC Linker.

STEP 6: NAME THE LIBRARY FILE AND SET PROJECT TYPE

After making the target settings select PPC Target and set the Project selection to Library and name the library file name to square.lib as shown in figure 18-8. When finished making the necessary target settings click the Save button.

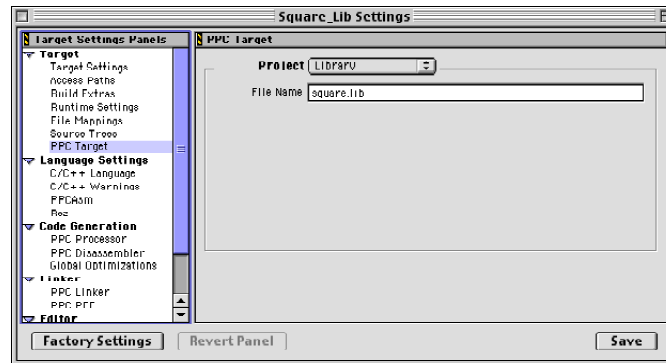


Figure 18-8: Setting Project Type and Library File Name

STEP 7: COMPIL THE PROJECT

Before closing the settings window and compiling the library select the C/C++ Language settings to ensure you are creating a C library function and not a C++ library function. This is done in CodeWarrior by ensuring the Activate C++ Compiler check box is not checked as shown in figure 18-9.

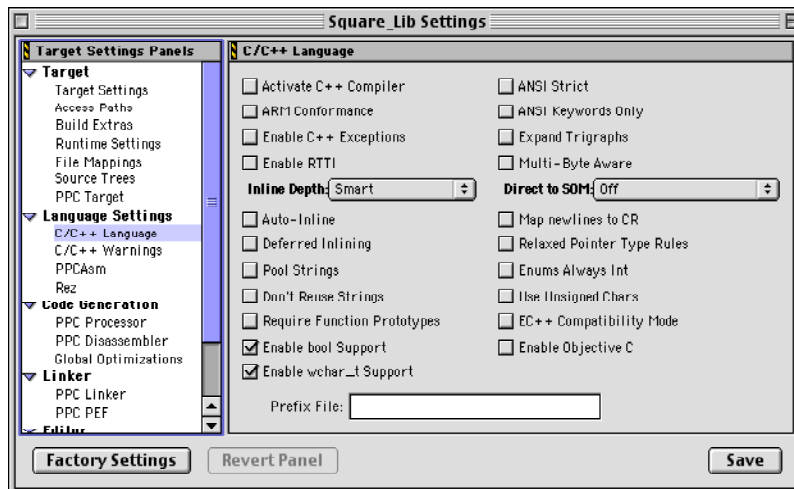


Figure 18-9: Ensure the Activate C++ Compiler Check Box is Not Checked

After checking the language settings the square.lib file can be created by making the project. Do this by selecting Make from the Project menu as shown in figure 18-10. Once the square.lib library file is created it can be used in another project. This is illustrated in step 8 below.

STEP 8: USE THE LIBRARY

OK, in steps 1 through 7 above a C library function named square() was created and placed in a static library file named square.lib. In this step the square() function will be used in a C++ program. The C++ program in this example is simply a main() function that calls the square() function which was written in C. The code for the main.cpp file is given in example 18.3. Notice on line 2 of example 18.3 that the square.h header file given in example 18.1 is included. The C++ project window with the square.lib library file added is shown in figure 18-11.

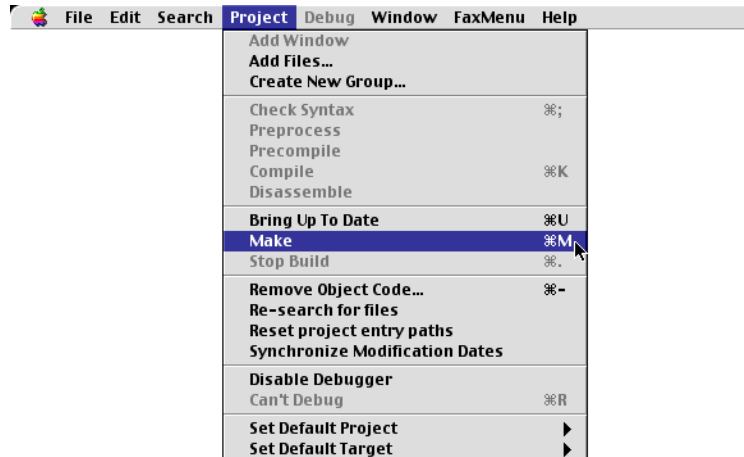


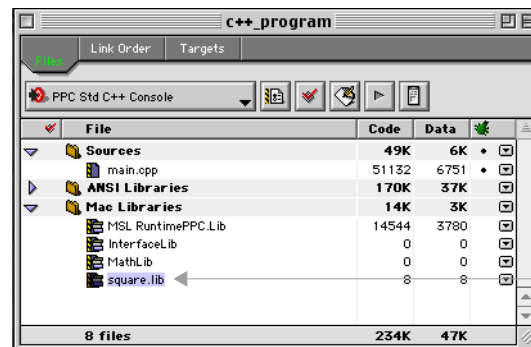
Figure 18-10: Select Make from the Project Menu to Create the square.lib File

```

1  #include <iostream>
2  #include "square.h"
3  using namespace std;
4
5  int main() {
6      cout<<square(5)<<endl;
7      return 0;
8  }

```

18.3 main.cpp



*square.lib library file
added to Mac Libraries
Group*

Figure 18-11: C++ Project Window with square.lib Library File Added.

OK, drum roll. An attempt to build the project as-is results in the link error shown in figure 18-12. The link error was produced because the square() function library was created in the C programming language. The C++ compiler is mangling the name of the square() function and because the square() function's name is not mangled in the C library the C++ linker cannot resolve the mangled function name with the non-mangled function name. What you must do to get the C++ project to work is modify the square.h header file in a way that tells the C++ compiler not to mangle the square() function name. You do this with the extern keyword as shown in example 18.4.

Referring to example 18.4, the square() function is declared to be of type C language linkage by wrapping it inside of an extern "C" { } declaration. This will indicate to the C++ compiler that this function was written in C vice C++. After this modification is made to the square.h header file the project can now be built with no link errors. The results of running the C++ project are shown in figure 18.13.

It should be noted now that the exact usage of the extern language linkage declaration is implementation dependent. Some compilers require an uppercase C while others will accept either an upper case C or a lower case c.

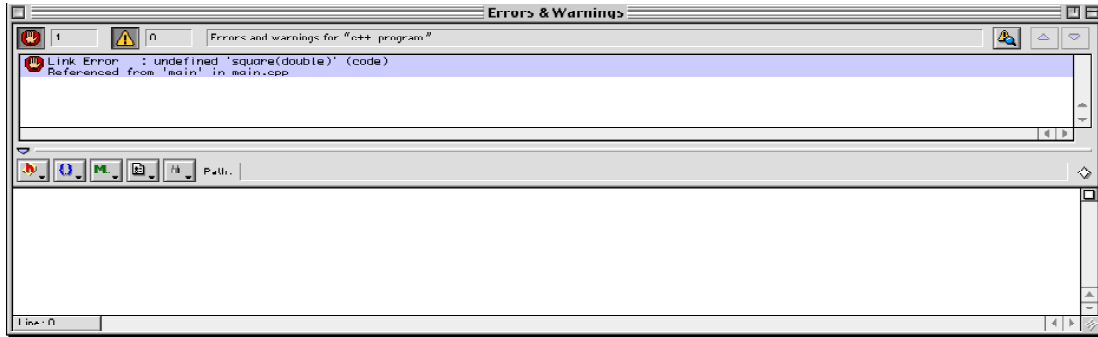


Figure 18-12: Link Error Resulting from First Attempt to Build the C++ Project that Uses a C Function

```

1  #ifndef SQUARE_H                                18.4 modified square.h
2  #define SQUARE_H
3
4  extern "C" {
5  double square(double d);
6  }
7
8  #endif

```

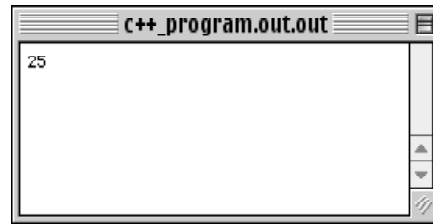


Figure 18-13: Results of Running the C++ Project Using the C square() Function

DECIPHERING C STANDARD LIBRARY FILES

Now that you understand how to incorporate C library functions into your C++ programs using the `extern` keyword you can unlock the mysteries of the C Standard Libraries. Up to this point you may have used functions from the C standard libraries in your C++ programs without a second thought. However, if you examine the C library files you will discover the C function declarations are wrapped in extern declarations. Now you know why.

Quick Review

The `extern "C"` language linkage specification is used to link C language routines with C++ programs. The distinction between C and C++ function names is necessary due to C++ name mangling. If a C function declaration omits the `extern "C"` linkage specification then the C++ compiler will not be able to resolve the mangled function name with the non-mangled name found in the C library file.

Other high-level programming languages can be used with C++ in similar fashion. The `extern` keyword can be used to specify not only "C" linkage but also "FORTRAN" and "PASCAL". Other linkage specifications may also be supported by different compiler manufacturers.

C++ And Assembly

Assembly language routines can generally be added to C++ programs in two ways: by embedding the assembly language statements directly in a C++ function, and by incorporating an object file created by an assembler in your project. In this section I will show you how to do both for the PC and Macintosh computing platforms.

SOME THINGS TO THINK ABOUT BEFORE USING ASSEMBLY

As with just about any decision you will make in your programming career the one regarding the use of assembly language in your C++ program should not be made willy-nilly. Some things to consider include the following:

- Efficient Processor Usage** - Old assembly language routines may not use modern processors efficiently. As a rule, new processor versions introduce new instructions. Old versions of assembly language will most certainly not support these new instructions.
- Code Optimization** - Face it, compilers generate better optimized code than do humans. If your reasons for using assembly include the one “I can do it better” I recommend letting the compiler have a go at it first.
- Portability** - Adding assembly to your C++ program will lock it into a specific processor and severely limit your code portability.
- Maintenance** - As time passes, assembly routines embedded directly in your C++ code will have to be upgraded more frequently than the C++ instructions to take advantage of new processor features.

KNOW THY IMPLEMENTATION DEPENDENCIES

Many implementation aspects of the C++ language are left to the discretion of the compiler manufacturer, and how each enables the integration of assembly language into your C++ code is one of the most implementation dependent aspects of all. For example, the method used to embed assembly language directly into a C++ function differs in Metrowerks CodeWarrior Release 5 between the PC and the Macintosh versions of the compiler. Your best resource for information regarding the use of assembly within your C++ code resides somewhere in your development environment’s documentation.

INLINE ASSEMBLY LANGUAGE IN A C++ FUNCTION

Assembly code can be embedded directly into the body of a C++ function by using the `asm` keyword. How the `asm` keyword is actually employed in your development environment is — you guessed it — implementation dependent. I will start first with a PC version. Example 18.5 gives the code for a header file that declares a function named `doubleVal()`.

```

1  #ifndef DOUBLE_H                                18.5 double.h
2  #define DOUBLE_H
3
4  int doubleVal(int d);
5
6  #endif

```

The `doubleVal()` function takes an integer argument and returns its doubled value. Example 18.6 implements the `doubleVal()` function with embedded assembly that targets an Intel[®] processor.

There are several important things to note in this example. First, the assembly instruction block is introduced with the `asm` keyword. Second, local variables and parameter names can be used as necessary in the assembly block. Also, Metrowerks CodeWarrior Release 5 allows the use of either the `asm` keyword or the `_asm` keyword.

Example 18.6 can now be used in a Win32 project. This is shown in figure 18-14. The `doubleVal()` function is used in a `main()` function and the results of running the project are shown in figure 18-15.

```

1 #include "double.h"
2
3 int doubleVal(int d){
4     int return_val;
5     asm{
6         mov eax, d           ;move d into the eax register
7         shl eax, 1          ;shift eax bits left by 1 bit
8         mov return_val, eax ;move result into return_val
9     }
10    return return_val;
11 }

```

18.6 double.cpp

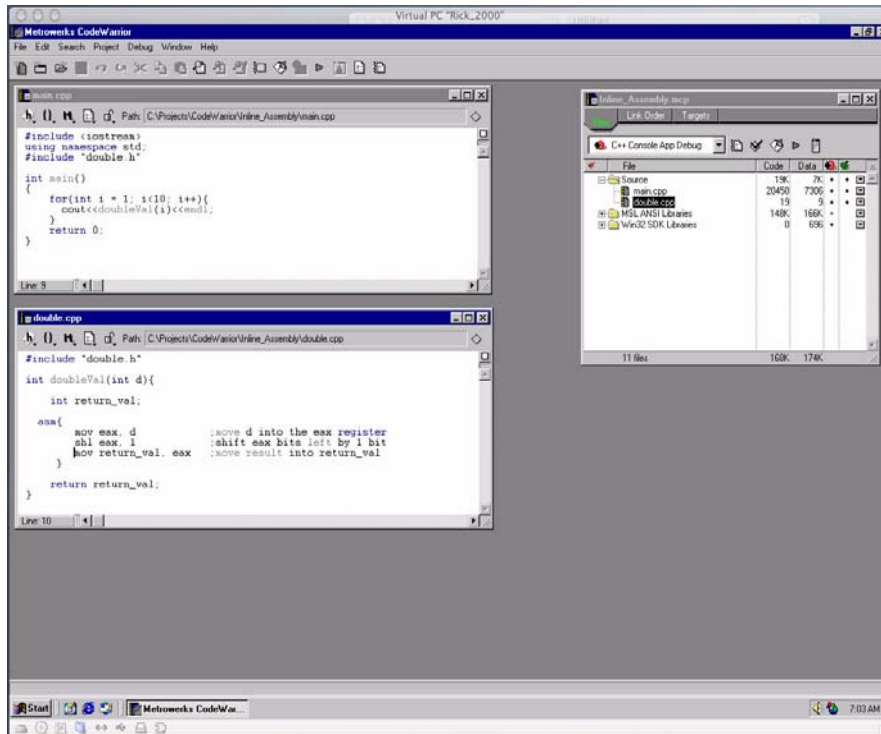


Figure 18-14: Win32 Project Using Inline Assembly Language

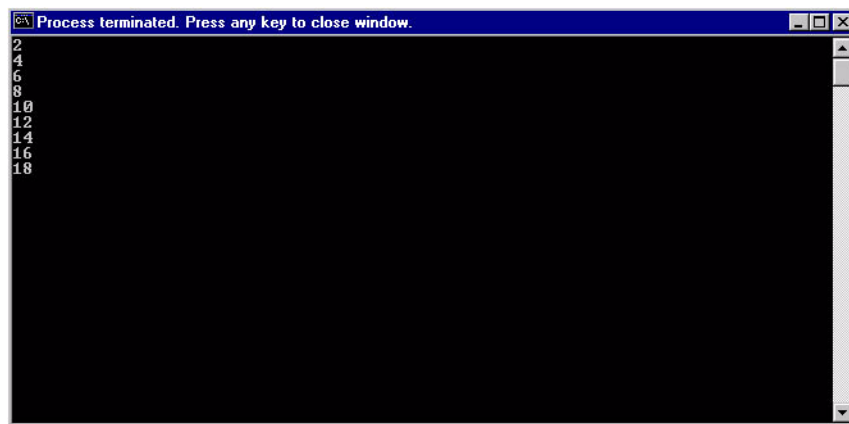


Figure 18-15: Results of Running the Inline Assembly Project

LINKING AN OBJECT FILE CREATED FROM ASSEMBLY LANGUAGE

This section will show you how to create a stand-alone assembly language routine, generate an object file, and use it in a C++ project. The target platform used for this demonstration is Win32. The assembly language file is assembled with Microsoft Macro Assembler (MASM) version 6.14 and used in a C++ Win32 project created using Metrowerks CodeWarrior Release 5.

PROCESS STEPS

The steps required to create an object file from assembly language and add it to a C++ project are listed below and illustrated in figure 18-16.

- Step 1** - Create assembly language file with a text editor and save it with the .asm extension,
- Step 2** - Assemble the file to create a Portable Executable/Common Object File Format (PE/COFF) object file,
- Step 3** - Create a C++ project using the development system of your choice,
- Step 4** - Create a header file that declares the name of the function contained in the object file. The function declaration must be declared to have extern “C” language linkage.
- Step 5** - Add the object file to the C++ project,
- Step 6** - Compile and run the project and do a victory dance!

1. Create .asm file

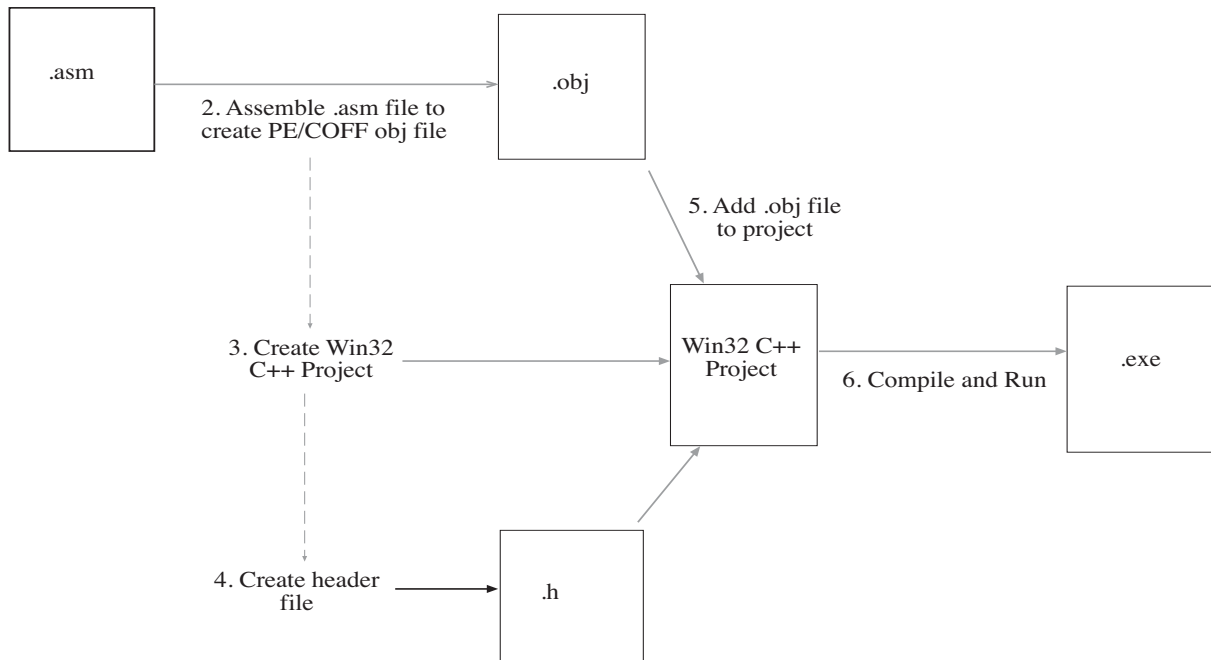


Figure 18-16: Adding Assembly Object File to C++ Project

STEP 1: CREATE ASSEMBLY LANGUAGE FILE

Using your favorite text editor create the assembly language file and save it with a .asm extension. Example 18.7 gives the assembly listing for a file named double.asm that implements an assembly procedure named doubleVal.

STEP 2: ASSEMBLE THE .ASM FILE

The assembly file now needs to be assembled into an object module. To run in a Win32 environment the object file needs to be in the PE/COFF object file format. Failure to generate the proper object file format will result in a link error.


```

1      .586
2      .MODEL flat, stdcall
3      .CODE
4      doubleVal PROC C int_val:DWORD
5          mov eax, int_val
6          shl eax, 1
7          ret
8      doubleVal endp
9      END

```

18.7 double.asm

For this example I am using Microsoft Macro Assembler version 6.14 via the command line. The command to assemble the double.asm file will look like this:

```
ml /c /coff /Fodv.obj double.asm
```

The results of running this command are shown in figure 18-17.

```

C:\Projects\MASM\project1>ml /c /coff /Fodv.obj double.asm
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: double.asm
C:\Projects\MASM\project1>_

```

Figure 18-17: Assembling double.asm with MASM ver. 6.14

Several MASM command line switches are used to obtain the right object module. First, the /c tells MASM to compile only and to not link the resulting file. The /coff switch creates a coff object module. The /Fo specifies the name of the output file. Running the assembler results in a coff object file named dv.obj that is ready to be used in a Win32 C++ project.

STEP 3: CREATE A C++ PROJECT

Create a Win32 C++ project using the IDE of your choice. The IDE of my choice is Metrowerks CodeWarrior and to it I have added a main.cpp file that calls the doubleVal() function. The code for the main.cpp file is shown in example 18-8.

```

1      #include <iostream>
2      #include "double.h"
3      using namespace std;
4
5      int main( void ){
6          for(int i=0; i<10; i++){
7              cout<<doubleVal(i)<<endl;
8          }
9          return 0;
10     }

```

18.8 main.cpp

STEP 4: CREATE A HEADER FILE

Create the double.h header file as shown in example 18.9. Since the doubleVal() function was created in assem-

bly language the function declaration must be declared to have extern “C” language linkage. Make the header file

```

1  #ifndef DOUBLE_H
2  #define DOUBLE_H
3
4  extern "C"{
5  int doubleVal(int d);
6  }
7
8  #endif

```

18.9 double.h

available to the project in the usual fashion.

STEP 5: Add THE OBJECT FILE TO THE PROJECT

When the object and header files have been created you are ready to use the object file, and the function it implements, in a C++ project. Figure 18-18 shows a Metrowerks CodeWarrior Win32 project with the dv.obj file included.

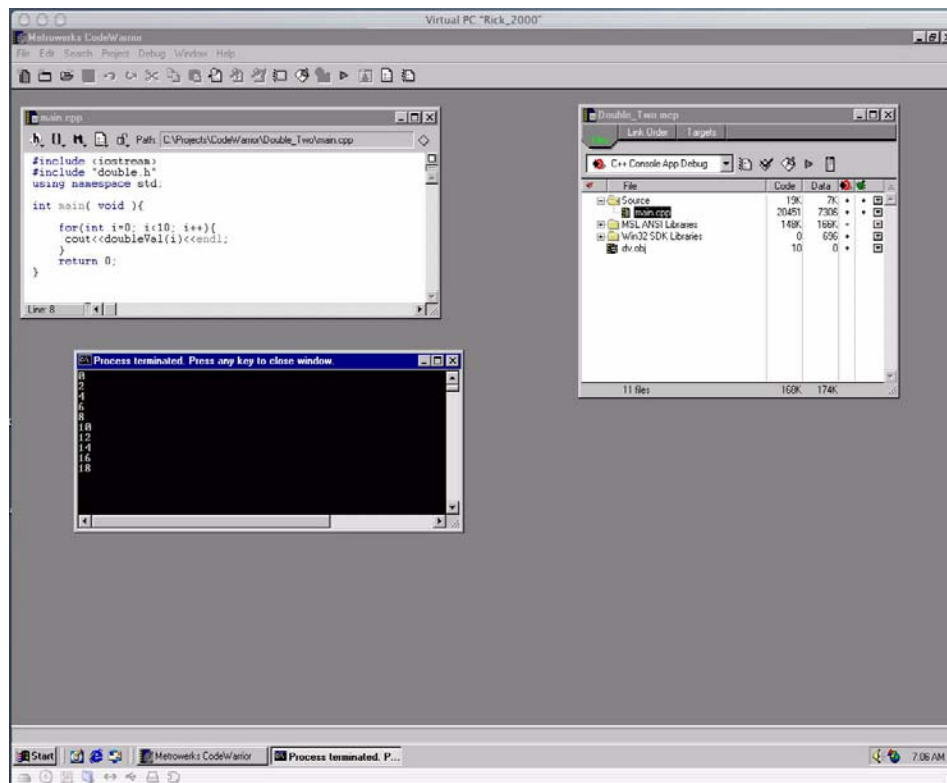


Figure 18-18: Win32 Project Using dv.obj

STEP 6. COMPILe AND RUN THE PROJECT

Figure 18-18 also shows the results of running the project.

USING INLINE ASSEMBLY IN THE MACINTOSH ENVIRONMENT

The syntax required to use inline assembly in a C++ program targeting the Macintosh™ platform is different from that required to target the PC. The obvious differences between assembly for the PC and assembly for the Macintosh™ will be the names and format of assembly instructions and the names and usages of the PowerPC™ registers. These issues are beyond the scope of this book, however, I do want to show you at least one example of inline assembly

bly targeting the Macintosh™.

I will implement the same doubleVal() function using Metrowerks CodeWarrior Release 5 for the Macintosh™. Example 18.10 gives the code for the double.cpp file. Note the differences between the Macintosh™ version of this

```

1  #include "double.h"
2
3  asm int doubleVal(int int_val){
4      lwz   r0, int_val    //load int_val into register 0
5      mulli r3, r0, 2      // multiply r0 by 2, store in r3 for return
6  }
```

*18.10 double.cpp
PowerPC Version*

file and the PC version. The asm keyword is still used but here it must precede the function definition. Local variable names can still be used in the assembly code as well as C++ style comments. Figure 18-19 shows the results of running the Macintosh™ version of the doubleVal() function.

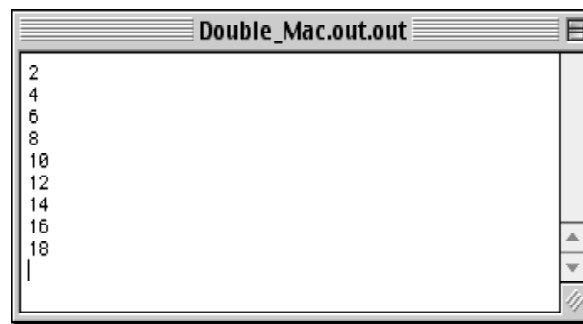


Figure 18-19: Results of Running Macintosh Version of doubleVal()

Quick Review

In this section you have learned how to incorporate assembly language into your C++ projects. The two primary methods of doing so are by including inline assembly instructions within a C++ function, or by creating stand-alone object modules with assembly language and linking those object modules to your C++ projects. Consult your compiler documentation to learn how it supports the asm keyword.

C++ AND JAVA: THE JAVA NATIVE INTERFACE (JNI)

C++ native code can be called from a Java program. This is accomplished using the Java Native Interface (JNI). With Java and JNI you write a Java program that dynamically loads a native library that contains one or more native methods declared in the Java class. The dynamic link library can be created using C++. The next section describes the process in detail and then gives both a Win32 and Macintosh OSX example.

STEPS TO CREATE A JNI C++ PROGRAM

This section lists the steps required to create a JNI C++ program. The steps are further illustrated in figure 18-20.

Step 1 - Create a Java source file that declares a class with one or more native methods. In addition to any native methods this class requires it also must also load the native dynamic library module using the System.loadLibrary() function.

Step 2 - Compile the Java source file to create a .class file.

Step 3 - Use the javah compiler with the -jni switch to automatically create a header file for use in your C++ program.

Step 4 - Create a C++ source file that implements the native method.

Step 5 - Compile the C++ source file to create a dynamic link library that exports the native method.

Step 6 - Run the Java program using the java virtual machine and do a victory dance!

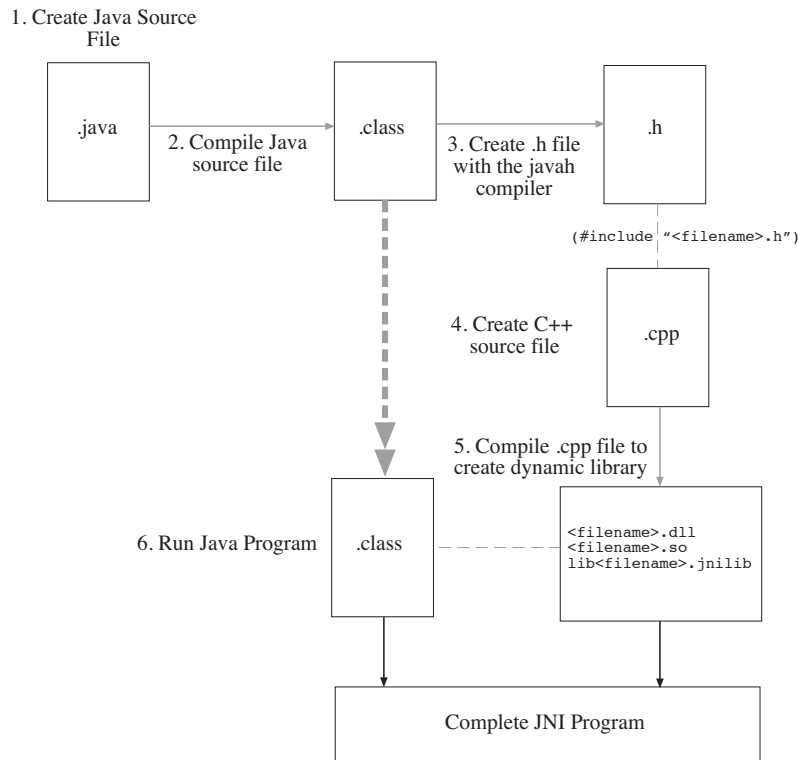


Figure 18-20: Steps to Create a Java Native Interface (JNI) Program

Everything goes smooth up until around step 6. Success here depends on what operating system you are running since the name of the library file the java virtual machine is looking for is — you guessed it — implementation dependent. Consult the Java JNI documentation to learn how to properly form dynamic library names in your computing environment. The next section provides a complete JNI example targeting the Win32 environment.

Win32 JNI Example

In this section I will show you how to create a complete JNI program that targets the Win32 platform. This example will entail creating a Java class named `SayHi`. The `SayHi` class will declare a native method named `sayHi()`. The `sayHi()` method will be implemented in C++ and saved in a Win32 dynamic link library (DLL). To call the `sayHi()` method the `SayHi` class must first load the DLL. These steps are described in detail below.

STEP 1: CREATE JAVA SOURCE FILE

Start your JNI project by creating a Java source file that declares one or more native methods. Example 18.11 gives the source code for a Java class named `SayHi`.

On line 3 the `SayHi` class declares a native method named `sayHi()` that is public, returns void, and takes no function arguments. Note that this is only a function declaration as opposed to an ordinary Java method which normally combines both the function declaration and definition within the class body.

On line 5 a function named `loadLibrary()` is defined. The `loadLibrary()` function will take a `String` argument that represents the name of the DLL the `SayHi` class must load before calling the `sayHi()` function. The `loadLibrary()` function then calls the `System.loadLibrary()` function using the `lib_name` parameter. Using a class instance function to load the dynamic library allows the overhead associated with loading the library to be deferred until it is actually needed.

```

1  public class SayHi{
2
3      public native void sayHi();
4
5      public void loadLibrary(String lib_name){
6          System.loadLibrary(lib_name);
7      }
8
9      public static void main(String args[]){
10         SayHi sh = new SayHi();
11         sh.loadLibrary(args[0]);
12         sh.sayHi();
13     }
14 }

```

18.11 SayHi.java

The main() function begins on line 9. A new SayHi object is created on line 10. Next, the loadLibrary() function is called using the library string name contained in arg[0]. Finally, the sayHi() function is called on line 12.

STEP 2: Compile JAVA SOURCE FILE

Use the Java compiler to compile the SayHi.java file. The Java compiler is invoked using the javac command as shown in figure 18-21.



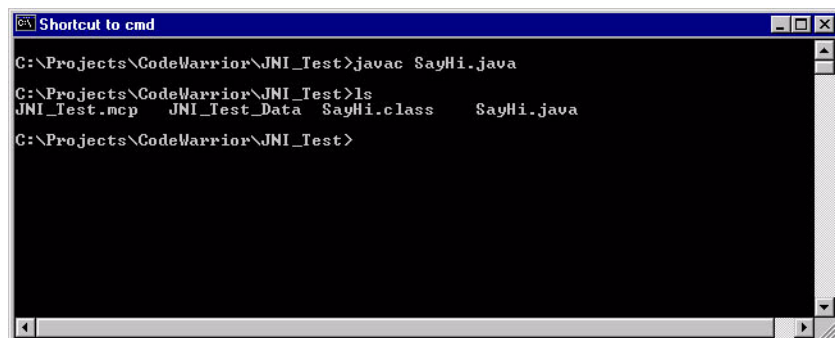
```

C:\Projects\CodeWarrior\JNI_Test>javac SayHi.java_

```

Figure 18-21: Compiling SayHi.java

Compiling the SayHi.java file results in a Java class file names SayHi.class as shown in figure 18-22.



```

C:\Projects\CodeWarrior\JNI_Test>javac SayHi.java
C:\Projects\CodeWarrior\JNI_Test>ls
JNI_Test.mcp  JNI_Test_Data  SayHi.class  SayHi.java
C:\Projects\CodeWarrior\JNI_Test>

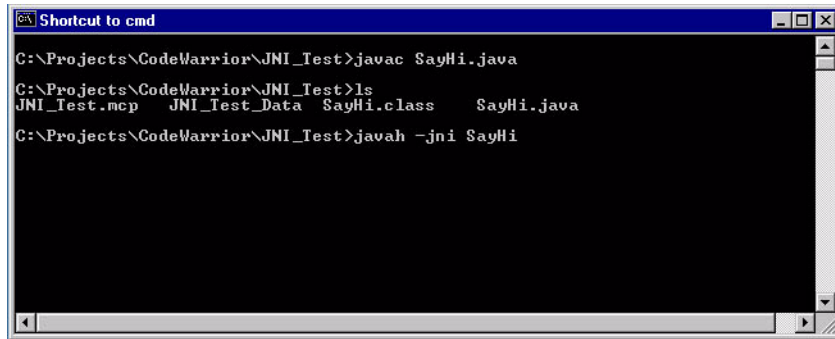
```

Figure 18-22: Compiling SayHi.java Results in SayHi.class

STEP 3: CREATE HEADER FILE

To create a C++ implementation of a Java native method you must have a header file that declares the C++ ver-

sion of the method. The Java platform supplies a command line tool named `javah` that can be used to automatically create these header files. The `javah` tool requires a Java class file. It automatically assumes the `.class` file extension. Figure 18-23 shows the `javah` tool being used to create the `SayHi.h` header file.



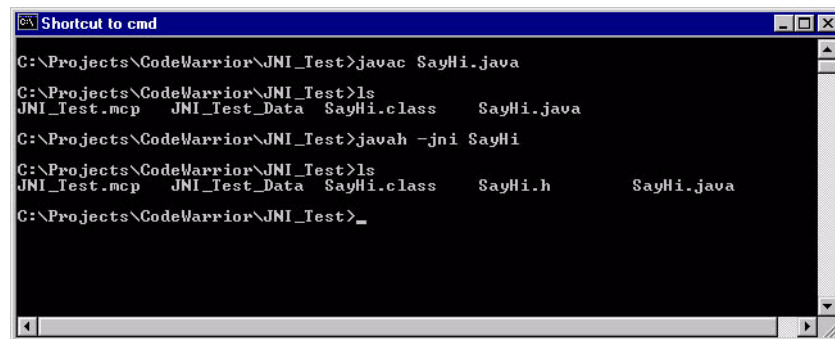
```

C:\Projects\CodeWarrior\JNI_Test>javac SayHi.java
C:\Projects\CodeWarrior\JNI_Test>ls
JNI_Test.mcp  JNI_Test_Data  SayHi.class  SayHi.java
C:\Projects\CodeWarrior\JNI_Test>javah -jni SayHi

```

Figure 18-23: Using `javah` to Create the `SayHi.h` Header File

Figure 18-24 shows the results of creating the `SayHi.h` header file with the `javah` command line tool.



```

C:\Projects\CodeWarrior\JNI_Test>javac SayHi.java
C:\Projects\CodeWarrior\JNI_Test>ls
JNI_Test.mcp  JNI_Test_Data  SayHi.class  SayHi.java
C:\Projects\CodeWarrior\JNI_Test>javah -jni SayHi
C:\Projects\CodeWarrior\JNI_Test>ls
JNI_Test.mcp  JNI_Test_Data  SayHi.class  SayHi.h      SayHi.java
C:\Projects\CodeWarrior\JNI_Test>_

```

Figure 18-24: Results of Creating `SayHi.h` Using `javah` Command Line Tool

Example 18.12 shows the contents of the `SayHi.h` header file.

```

1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class SayHi */
4
5  #ifndef _Included_SayHi
6  #define _Included_SayHi
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Class:      SayHi
12  * Method:    sayHi
13  * Signature: ()V
14  */
15 JNIEXPORT void JNICALL Java_SayHi_sayHi
16     (JNIEnv *, jobject);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif

```

18.12 *SayHi.h*

The actual native function declaration appears on lines 15 and 16. For a complete discussion on the purpose and

use of the JNIEnv* and jobject parameters refer to the JNI references listed at the end of the chapter.

Now, with the header file generated you can create the SayHi() function C++ implementation file.

STEP 4: CREATE C++ SOURCE FILE

Example 18.13 shows the source code for the sayhi.cpp file.

```

1  #include "jni.h"
2  #include "SayHi.h"
3  #include <iostream>
4  using namespace std;
5
6  JNIEXPORT void JNICALL Java_SayHi_sayHi (JNIEnv * env, jobject jo){
7
8      cout<<"C++ SayHi() function working fine!"<<endl;
9  }

```

18.13 sayhi.cpp

The sayhi.cpp file needs to include the jni.h and the freshly-generated SayHi.h header files in addition to the iostream header file. With the sayhi.cpp file in hand you can now create a DLL.

STEP 5: COMPILE C++ SOURCE FILE TO CREATE DYNAMIC LINK LIBRARY

To create the Win32 DLL I will use Metrowerks CodeWarrior Release 5 and start with an empty project. I will then add the sayhi.cpp file and any libraries needed to create the DLL.

Figure 18-25 shows a blank CodeWarrior project window named JNI_Test. Figure 18-26 shows the sayhi.cpp file added to the blank project.

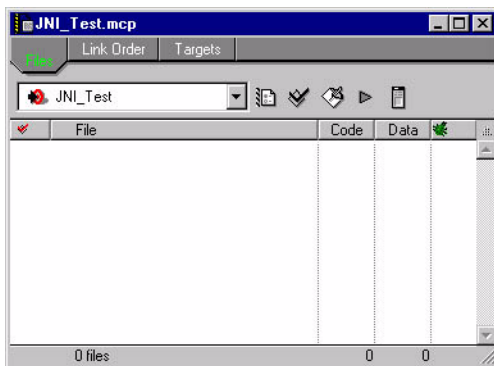


Figure 18-25: Blank CodeWarrior Project

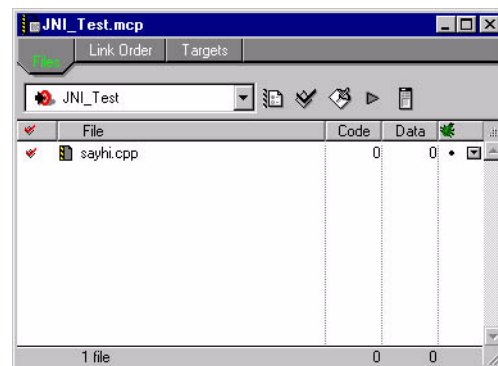


Figure 18-26: sayhi.cpp Added to Blank Project

To generate a Win32 DLL using Metrowerks CodeWarrior you must add several important libraries. Figure 18-27 shows the blank project window after adding the necessary libraries.

Almost there. Before generating the DLL you must set some target settings. Figure 18-28 shows the target settings window with the necessary settings.

Notice in the Target Settings window the Project Type is set to Dynamic Link Library (DLL) and the File Name is set to SayHi.dll.

When the necessary target settings are completed you can make the project to generate the SayHi.dll file. Figure 18-29 shows a directory listing after compiling the SayHi project.

STEP 6: RUN JAVA PROGRAM

Once you have generated the SayHi.dll file successfully you can run the SayHi Java application. The name of the SayHi dynamic link library must be supplied on the command line when the SayHi Java application is run. The command line to execute the SayHi program looks like this:

```
java SayHi SayHi
```

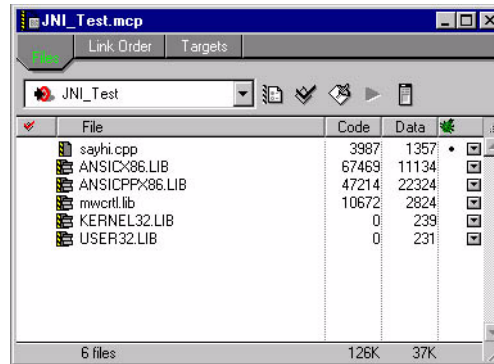


Figure 18-27: Blank Project Window Showing Added Library Files

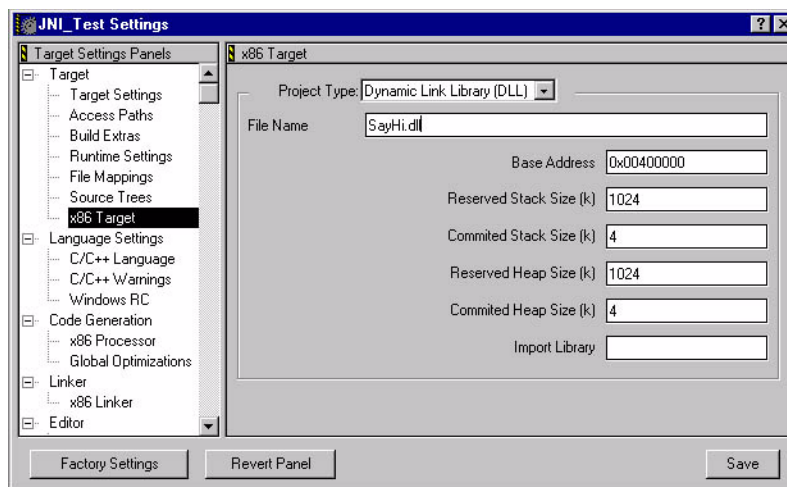


Figure 18-28: Target Settings Window

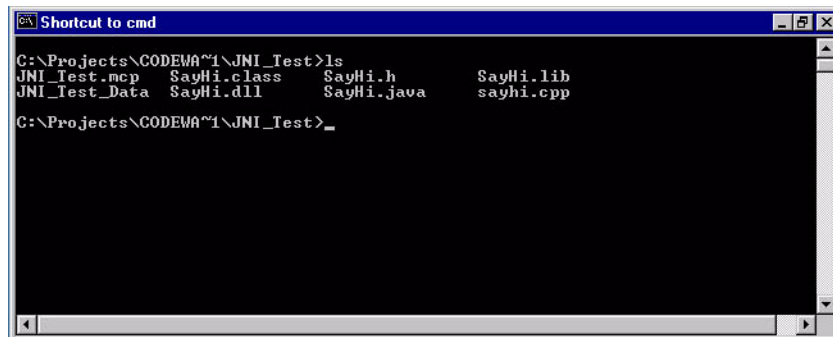
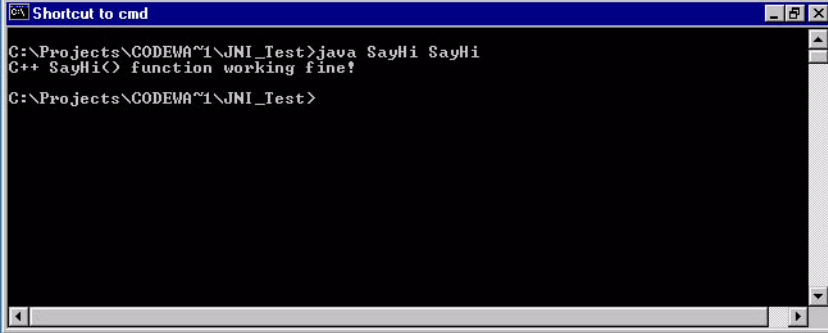


Figure 18-29: Directory Listing Showing SayHi.dll

Notice the name of the dynamic link library is just SayHi and not SayHi.dll. The `System.loadLibrary()` function will complete the library name by appending the `.dll` extension to the library name you supply. Figure 18-30 shows the results of running the SayHi Java application.

MACINTOSH OSX JNI EXAMPLE

In this section I will show you how to create the Macintosh OSX version of the same SayHi JNI program as



```

C:\Projects\CODEWA~1\JNI_Test>java SayHi SayHi
C++ SayHi() function working fine!
C:\Projects\CODEWA~1\JNI_Test>

```

Figure 18-30: Results of Running the SayHi Java Application

shown in the previous section. The steps are the same as those listed in the Win32 example with two major exceptions. 1) The name of the dynamic link library will be different and, 2) the development environment used to create the dynamic link library will be different.

STEP 1: CREATE JAVA SOURCE FILE

The same SayHi.java file used in the previous section will be used in this example. Refer to example 18.11 for the SayHi.java source code.

STEP 2: COMPILE JAVA SOURCE FILE

The Macintosh™ OSX operating system comes with a set of developer tools that includes the Java platform for OSX. The command line tools are the same for OSX as they are for Microsoft Windows™. Compile the SayHi.java file to produce the SayHi.class file.

STEP 3: CREATE HEADER FILE

Use the javah command line tool to create the SayHi.h header file. This file should look exactly like the SayHi.h file created using the Win32 version of javah.

STEP 4: CREATE C++ SOURCE FILE

The same C++ source file sayhi.cpp used in the Win32 example is used here as well. Refer to example 18.13 for the source code listing of sayhi.cpp.

STEP 5: COMPILE C++ SOURCE FILE TO CREATE DYNAMIC LINK LIBRARY

OK, here is where things go a little differently. I will use the GNU C++ compiler to create the OSX version of the dynamic link library. The GNU C++ compiler comes with Macintosh OSX but you might have to install it separately. If you have not installed the OSX developer tools you should do so now.

To create the dynamic link library with the GNU C++ compiler use the following command line:


```
g++ -I <java include file path> -dynamiclib -o <output file name> <C++ source file to compile>
```

The -I switch tells the GNU C++ compiler where to find include files. You will have to use this compiler switch to tell the compiler where to find the jni.h header file as it will most likely not be located in the environment's normal include path.

The -dynamiclib switch tells the compiler to produce a dynamic link library. The -o switch tells the compiler what to name the output file. The name of the OSX version of the dynamic link library should be lib<lib_name>.jnilib. In other word, the Java System.loadLibrary() function, when run in the OSX environment, will take the name of the library (SayHi) and prefix it with lib and suffix it with .jnilib to form the complete library name. Therefore, it will look for a library file named libSayHi.jnilib. You must instruct the GNU compiler to produce a

dynamic link library file named `libSayHi.jnilib`.

Finally, tell the GNU compiler the name of the C++ file to compile. Figure 18-31 shows the GNU C++ compiler being invoked from a Macintosh™ OS X terminal window.



```

Terminal — tcsh (tty1)
Last login: Sat Feb  8 09:35:32 on console
Welcome to Darwin!
[Rick-Millers-Computer:~] swodog% cd desktop
[Rick-Millers-Computer:~/desktop] swodog% cd pb_projects
[Rick-Millers-Computer:~/desktop/pb_projects] swodog% cd jni_test_osx
[Rick-Millers-Computer:~/desktop/pb_projects/jni_test_osx] swodog% ls
SayHi.class  SayHi.h      SayHi.java   jni.h        jni_md.h     libSayHi.jnilib  sayhi.cpp
[Rick-Millers-Computer:~/desktop/pb_projects/jni_test_osx] swodog% g++ -I /users/swodog/desktop/pb_projects/jni_test_osx -dynamiclib -o libSayHi.jnilib sayhi.cpp
[Rick-Millers-Computer:~/desktop/pb_projects/jni_test_osx] swodog% g++ -I /users/swodog/desktop/pb_projects/jni_test_osx -dynamiclib -o libSayHi.jnilib sayhi.cpp

```

Figure 18-31: Compiling `sayhi.cpp` Using `g++` to Generate an OS X Dynamic Link Library

Figure 18-32 shows the directory listing that includes the `libSayHi.jnilib` dynamic link library file.



```

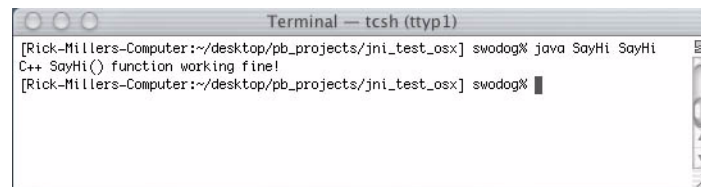
Terminal — tcsh (tty1)
[Rick-Millers-Computer:~/desktop/pb_projects/jni_test_osx] swodog% ls
SayHi.class  SayHi.java   jni_md.h     sayhi.cpp
SayHi.h      jni.h        libSayHi.jnilib
[Rick-Millers-Computer:~/desktop/pb_projects/jni_test_osx] swodog%

```

Figure 18-32: Directory Listing Showing `libSayHi.jnilib` File

STEP 6: RUN JAVA PROGRAM

All that is left to do is run the `SayHi` Java program. This is done in exactly the same fashion as the Win32 example. Use the `java` command line tool giving the name of the `SayHi` class and the name of the `SayHi` dynamic link library. Figure 18-33 shows the results of running the Java `SayHi` program.



```

Terminal — tcsh (tty1)
[Rick-Millers-Computer:~/desktop/pb_projects/jni_test_osx] swodog% java SayHi SayHi
C++ SayHi() function working fine!
[Rick-Millers-Computer:~/desktop/pb_projects/jni_test_osx] swodog%

```

Figure 18-33: Results of Running `SayHi` Java Program in an OS X Terminal Window

WHEN TO USE JNI

There is a certain amount of overhead involved with making a native function call from a Java program. If you are writing a native function with the intention of calling it frequently, say, from the body of an often-repeated loop, a native method may not be the optimum solution, especially if the function is small and takes only a fraction of the time to execute vs. the time it takes to make the function call. If, however, the function does something significant and is called infrequently, then JNI may be the way to go.

Quick Review

C++ can be used to write functions that target specific hardware platforms. These native functions can be called from a Java program using the Java Native Interface. Most of the steps involved in creating a JNI program are port-

ble across computing platforms with the exception of how the native dynamic link library is created and how it is named.

SUMMARY

This chapter showed you how to use C++ with different programming languages. The three different programming languages used included C, assembly, and Java.

The extern “C” language linkage specification is used to link C language routines with C++ programs. The distinction between C and C++ function names is necessary due to C++ name mangling. If a C function declaration omits the extern “C” linkage specification then the C++ compiler will not be able to resolve the mangled function name with the non-mangled name found in the C library file. Other high-level languages can be used with C++ by using the extern language linkage specification. Consult your development environment documentation for details.

The two primary methods of using assembly language with C++ programs include using inline assembly instructions within a C++ function, or by creating stand-alone object modules with assembly language and linking those object modules to C++ projects. Consult your compiler documentation to learn how it supports the asm keyword or other inline assembly methods.

C++ can be used to write functions that target specific hardware platforms. These native functions can be called from a Java program using the Java Native Interface. Most of the steps involved in creating a JNI program are portable across computing platforms with the exception of how the native dynamic link library is created and how it is named. As with everything else in this chapter, consult your development environment documentation for details on how to create a dynamic link library that can be loaded by a Java program to support the Java Native Interface.

Skill Building Exercises

1. **Creating and Linking C Libraries:** Write a C function that counts the number of words in a text file and returns the count. Compile the routine and create a static library. Write a C++ program that calls the wordCount() function.
2. **Inline Assembly:** Write a C++ function that takes an array of integers and the number of elements the array contains as arguments. Sum the array contents and return the result using assembly language for your particular hardware and compiler.
3. **Stand Alone Assembly Modules:** Convert the array summation function you wrote in the previous exercise to a stand-alone assembly module. Create a C++ program that calls the arraySum() function.
4. **Java Native Interface (JNI):** Write a Java program that calls the wordCount() function created in the first exercise.

SUGGESTED PROJECTS

1. **Research:** Procure a copy of Sheng Liang’s *The Java Native Interface: Programmer’s Guide and Specification* and learn how to invoke the Java virtual machine from a C++ program.
2. **Research:** Procure and read a book on assembly language programming for your particular hardware platform.
3. **RobotRat Revisited:** Write a Java program that invokes the C++ RobotRat program

SELF TEST QUESTIONS

1. (T/F) C enables programmers to overload function names.
2. What is the purpose of the extern linkage specification?
3. What is name mangling and why do C++ compilers employ this technique?
4. List and describe the steps required to create a C function library.
5. List and describe the four issues to consider before using assembly language in your programs.
6. What C++ keyword allows you to embed assembly language in the body of C++ functions?
7. (T/F) All C++ compilers implement the capability to include assembly language in the body of functions the exact same way.
8. List and describe the steps required to create, compile, and link an assembly language module to a C++ program.
9. What is the purpose of the javah command line tool?
10. List and describe the steps required to call C++ functions from a Java program using the Java Native Interface (JNI).
11. Under what circumstances would it not be a good idea to call a native C++ function from a Java program?

REFERENCES

International Standard, ISO/IEC 14882, *Programming Languages — C++*, First Edition 1998-09-01

Microsoft Macro Assembler Version 6.4 Documentation

Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts. ISBN: 0-201-32577-2

The PowerPC 601 RISC Microprocessor User's Manual. Motorola. MPC601UM/AD REV 1

Apple Technical Q&A Java28: Creating JNI Libraries with Project Builder. June 19 2001

Apple Macintosh OSX Version 10.2.3 Developer Tools GCC Compiler man pages

David Flanagan. *Java in a Nutshell: A Desktop Quick Reference*, Fourth Edition. O'Reilly. Cambridge, Massachusetts. ISBN: 0-596-00283-1

Microsoft Knowledge Base Article 121460 - Common Object File Format (COFF) Revision Date: 8 August 2001

Metrowerks CodeWarrior Release 5 Development Environment Documentation. Targeting MacOS, chapter 13: Inline Mac OS Assembler

Metrowerks CodeWarrior Release 5 Development Environment Documentation. Targeting Windows, chapter 5: Creating Static Libraries for Win32/x86

Metrowerks CodeWarrior Release 5 Development Environment Documentation. Targeting Windows, chapter 11: Inline x86 Assembler

Metrowerks CodeWarrior Release 5 Development Environment Documentation. Targeting Windows, chapter 6: Creating Win32/x86 Dynamic Link Libraries

NOTES

CHAPTER 19



Hands On!

THREE DESIGN PRINCIPLES

LEARNING OBJECTIVES

- *List the preferred characteristics of an object-oriented application architecture*
- *State the definition of the Liskov Substitution Principle (LSP)*
- *State the definition of Bertrand Meyer's Design by Contract (DbC)*
- *Recognize the close relationship between the Liskov Substitution Principle and Design by Contract*
- *Specify preconditions and postconditions for class and instance functions*
- *Specify class invariants*
- *State the definition of the Open-Closed Principle (OCP)*
- *State the definition of the Dependency Inversion Principle (DIP)*
- *Apply the Liskov Substitution Principle in the design and implementation of a class inheritance hierarchy*
- *Apply Design by Contract in the design and implementation of a class inheritance hierarchy*
- *Apply the Open-Closed Principle in the design and implementation of a class inheritance hierarchy*
- *Apply the Dependency Inversion Principle in the design and implementation of a class inheritance hierarchy*

INTRODUCTION

Building complex, well-behaved, object-oriented software is a difficult task for several reasons. First, simply programming in C++ does not automatically make your application object-oriented. A legacy C application rewritten in C++ without a proper object-oriented architecture design is just Cplusplusified C code. This comes as a shock to those who embrace and adopt C++ in hopes of finding a panacea.

Second, the process by which you become proficient at object-oriented design is characterized by experience. It takes a lot of time and money to learn the lessons of bad architecture design and then apply those lessons-learned to create good object-oriented architectures.

This chapter will help you jump-start your object-oriented architecture design efforts. It begins with a discussion of the preferred characteristics of a well-designed object-oriented architecture. It then presents and discusses three important object-oriented design principles and guidelines you can immediately apply to your architecture designs to drastically improve performance, reliability, and maintainability.

The three design principles include the Liskov substitution principle, the open-closed principle, and the dependency inversion principle. Bertrand Meyer's design by contract programming is discussed in the context of its close relationship to, and extension of, the Liskov substitution principle.

An understanding of these three design principles, along with an understanding of how to apply them using the C++ language, will significantly improve your ability to design robust, object-oriented architectures.

THE PREFERRED CHARACTERISTICS OF AN OBJECT-ORIENTED ARCHITECTURE

From a programmer's perspective, a well-designed, object-oriented architecture manifests itself as an inheritance hierarchy, including a set of abstract data type vertical and horizontal relationships, that exhibits several key characteristics. It is 1) easy to understand, 2) easy to reason about, and 3) easy to extend. Each of these three characteristics are discussed briefly below.

EASY TO UNDERSTAND – (HOW DOES THIS THING WORK?)

A programmer, when shown a component diagram of a complex software system, should be able to understand what it does, or what it is you are trying to do, in about five minutes flat. To do this a software architecture must be designed to be understood.

The organizational complexity of large software systems can be overwhelming if the architecture is poorly designed. An application comprised of even a small number of tightly coupled software components requires significantly more effort to understand than one designed to be understood quickly. An application software architecture must be thoroughly understood by a programmer before the effects of changing its components or adding functionality can be accurately assessed.

EASY TO REASON ABOUT – (WHAT ARE THE EFFECTS OF CHANGE?)

The effects of changing pieces of a software application must be fully predictable. Programmers must be confident that the changes they make to one code module will not mysteriously break another, seemingly unrelated, module in the system. If the effects of change can be accurately predicted then the architecture can be reasoned about. The best way to reason about the effects of change is to render the necessity for code changes unnecessary. (The effects of no change is definitely predictable!)

EASY TO EXTEND – (WHERE DO I ADD FUNCTIONALITY?)

Well-designed application architectures accommodate the addition of features and facilitate component reuse. A programmer, when tasked with adding new functionality to an application, must know exactly where to put it. The act of adding functionality should not require the changing of existing code, but rather its extension.

THE LISKOV SUBSTITUTION PRINCIPLE & DESIGN BY CONTRACT

Dr. Barbara Liskov and Dr. Bertrand Meyer are both important figures in the object-oriented software research community. The two design principles and guidelines that bear their name are the Liskov substitution principle (LSP) and Bertrand Meyer's design by contract (DbC). These closely related object-oriented design concepts are covered together in this section and can be summarized in the following statement:

Subtype objects must be behaviorally substitutable for supertype objects. Programmers must be able to reason correctly about and rely upon the behavior of subtypes using only the supertype behavior specification.

REASONING ABOUT THE BEHAVIOR OF SUPERTYPES AND SUBTYPES

Programmers must be able to reason correctly about the behavior of abstract data types and their derived subtypes. The LSP and DbC provide both theoretical and applied foundations upon which programmers can build well-behaved class inheritance hierarchies that facilitate the object-oriented architectural reasoning process.

Relationship BETWEEN THE LSP AND DbC

The LSP and DbC are closely related concepts primarily because they both draw from largely the same body of research in the formulation of their theories. They each address the question of how a programmer should be able to reason about the behavior of a subtype object when it is substituted for a supertype object, they each address the role of method preconditions and postconditions in the specification of desired object behavior, and they each discuss the role of class invariants and how method postconditions should ensure invariant state conditions are preserved. They both seek to provide a mechanism for programmers to create reliable object-oriented software.

Design by contract differs from the LSP in its emphasis on the notion of contracts between supertype and subtype. The base class (supertype) is a contractor that may, at runtime, have its interface functions performed by a subcontractor (subtype). Programmers should not need any apriori knowledge of the subtype's existence when they write the code that may come to rely on the subtype's behavior. The subtype, when substituted for the supertype, should fulfill the contract promised by the supertype. In other words, the subtype object should not pull any surprises.

Another difference between the LSP and DbC is that the LSP is more notional, while DbC is more practical. By this I mean no language, as of this writing, directly supports the LSP specifically, with perhaps the exception of the type checking facilities provided by a compiler. Design by contract, on the other hand, is directly supported by the Eiffel programming language.

THE COMMON GOAL OF THE LSP AND DbC

The LSP and DbC share a common goal. They both aim to help software developers build correct software from the start. Given this common goal I will occasionally refer to both concepts collectively as the LSP/DbC.

C++ SUPPORT FOR THE LSP AND DbC

With the exception of type checking, C++ does not provide direct language support for either the LSP or DbC. However, there are techniques you can use to enforce preconditions and postconditions, and to ensure the state of class invariants. Regardless of the level of language support for either the LSP or DbC, programmers can realize significant improvements in their overall class hierarchy designs by simply keeping the LSP and DbC in mind during the design process.

DESIGNING WITH THE LSP/DbC IN MIND

The LSP/DbC focuses on the correct specification of supertype and subtype behavioral relationships. By keeping the LSP/DbC in mind when designing class hierarchies programmers are much less likely to create subclasses that

implement behavior incompatible with that specified by the base class.

The POWER AND DANGER of C++

C++ gives programmers the powerful object-oriented language features of redeclaration, polymorphism, and dynamic binding. By declaring pointers to base class objects programmers can substitute derived class objects at runtime and thereby implement dynamic polymorphic behavior. It is this dynamic polymorphic behavior that programmers must be able to correctly reason about. Yet, what is powerful can also be dangerous.

The same language features of redeclaration, polymorphism, and dynamic binding that provide C++ programmers with enormous power and flexibility can cause significant problems if not wielded properly.

CLASS DECLARATIONS VIEWED AS BEHAVIOR SPECIFICATIONS

A class declaration introduces a new abstract data type into a programmer's environment. The class declaration is, by its very nature, a behavioral specification. The behavior is specified by the set of public interface functions made available to clients, by the set of possible states an object may assume, and by the side effects resulting from method execution.

A class declaration can specify behavior only, as is the case with an abstract base class containing only pure virtual functions, or, it can both specify and implement behavior, as is the case where class functions are implemented for a particular class.

An abstract data type can adopt the behavioral specification of another abstract data type. The former would be the subtype and the latter the supertype. When the supertype is an abstract base class the subtype inherits only a behavior specification. It must then either implement the specified behavior or further defer the implementation to yet another subtype. When a supertype provides behavior implementation, a subtype may adopt the supertype behavior outright or provide an overriding behavior. It is the correct implementation of this overriding behavior about which the LSP/DbC is most concerned. Programmers can create well-behaved subtypes by employing preconditions, postconditions, and class invariants.

PRECONDITIONS, POSTCONDITIONS, AND CLASS INVARIANTS

Preconditions, postconditions, and class invariants are the three cornerstones of both the LSP and DbC. Their definitions and application are discussed in this section.

CLASS INVARIANT

A class invariant is an assertion about an object property that must hold true for all valid states an object can assume. For example, suppose an airplane object has a speed property that can be set to a range of integer values between 0 and 800. This rule should be enforced for all valid states an airplane object can assume. All methods that can be invoked on an airplane object must ensure they do not set the speed property to less than 0 or greater than 800.

PRECONDITION

A precondition is an assertion about some condition that must be true before a function can be expected to perform its operation correctly. For example, suppose the airplane object's speed property can be incremented by some value and there exists in the set of airplane's public interface functions one that increments the speed property anywhere from 1 to 5 depending on the value of the argument supplied to the function. For this function to perform correctly, it must check that the argument is in fact a valid increment value of 1, 2, 3, 4, or 5. If the increment value tests valid then the precondition holds true and the increment function should perform correctly.

The precondition must be true before the function is called, therefore it is the responsibility of the caller to make the precondition true, and the responsibility of the called function to enforce the truth of the precondition.

POSTCONDITION

A postcondition is an assertion that must hold true when a function completes its operations and returns to the caller. For example, the airplane's speed increment function should ensure that the class invariant speed property being $0 \leq \text{speed} \leq 800$ holds true when the increment function completes its operations.

An Example

Example 19.1 gives the source code for a header file named `incrementer.h`. An `incrementer` object can simply be incremented by 1, 2, 3, 4, or 5, and maintain a state value between 0 and 100.

19.1 *incrementer.h*

```

1  #ifndef INCREMENTER_H
2  #define INCREMENTER_H
3
4  class Incrementer {
5      /*****
6      Class invariant: 0 <= Incrementer::val <= 100
7      *****/
8  public:
9      Incrementer(int i = 0);
10     virtual ~Incrementer();
11     /*****
12     function: void increment(int i);
13     precondition: 0 < i <= 5
14     postcondition: 0 <= Incrementer::val <= 100
15     *****/
16     virtual void increment(int i);
17
18     private:
19         int val;
20         void checkInvariant();
21     };
22 #endif

```

Besides a constructor and destructor, class `Incrementer` contains a private integer instance attribute named `val`, a public virtual function named `increment()`, and a private function named `checkInvariant()`.

The class invariant for `Incrementer` states that the `val` attribute can be any integer value between and including 0 and 100. The `increment()` function is introduced by a comment block that details the function's precondition and postcondition. Example 19.2 gives the source code for the `incrementer.cpp` file.

In this example, the class invariant, precondition, and postcondition are enforced using the `assert` function found in the standard C library. Include the `assert.h` header file to access the `assert` function. The `assert` function will cause the program to immediately stop execution should the assertion fail.

Example 19.3 gives the code for a `main()` function that uses an `Incrementer` object.

19.3 *main.cpp*

```

1  #include <iostream>
2  using namespace std;
3  #include "incrementer.h"
4
5  int main(){
6      Incrementer* inc_ptr = new Incrementer(95);
7      inc_ptr->increment(4);
8      inc_ptr->increment(5);
9      inc_ptr->increment(3);
10     delete inc_ptr;
11     return 0;
12 }

```

On line 6 of example 19.3 an `Incrementer` pointer named `inc_ptr` is declared and a new `Incrementer` object is created with its `val` property initialized to 95. On lines 7 through 9 the `increment()` method is called via `inc_ptr` with integer arguments that satisfy the function's precondition. The results of running example 19.3 are shown in figure 19-1.

A programmer using the `Incrementer` class will know how `Incrementer` objects will behave by reading the class invariant, precondition, and postcondition comments in the `incrementer.h` file. But what will happen if a programmer calls the `increment()` function with an argument that fails the precondition? Example 19.4 gives a slightly revised version of the `main()` function originally given in example 19.3.

```

1  #include "incrementer.h"
2  #include "assert.h"
3  #include <iostream>
4  using namespace std;
5
6  Incrementer::Incrementer(int i):val(i){
7      cout<<"Incrementer object created!"<<endl;
8      checkInvariant();
9  }
10
11 Incrementer::~Incrementer(){
12     cout<<"Incrementer object destroyed!"<<endl;
13 }
14
15 void Incrementer::increment(int i){
16     // enforce precondition 0 < i <= 5
17     assert((i > 0) && (i <= 5));
18
19     // change incrementer object state
20     if((val+i) <= 100){
21         val += i;
22     }else{
23         int temp = val;
24         temp += i;
25         val = (temp - 100);
26     }
27
28     // enforce class invariant
29     checkInvariant();
30
31     cout<<"Incrementer value is: "<<val<<endl;
32 }
33
34
35 void Incrementer::checkInvariant(){
36     assert((val >= 0) && (val <= 100));
37 }

```

19.2 incrementer.cpp

Enforce class invariant when object created

Enforce precondition when function called

Perform function operations

Enforce the postcondition

```

Liskov.out.out
Incrementer object created!
Incrementer value is: 99
Incrementer value is: 4
Incrementer value is: 7
Incrementer object destroyed!

```

Figure 19-1: Results of Running Example 19.3

In this example the call to the `increment()` function on line 9 uses an integer argument value of 6 that fails the precondition. Figure 19-2 shows the results of running example 19.4.

USING INCREMENTER AS A BASE CLASS

A programmer using `Incrementer` objects knows from reading the class invariant, precondition, and postcondition specification for the `Incrementer` class and its functions how those objects can be used in a program and how they should behave. And that is all they should have to know, even when an `Incrementer` pointer points to an object that belongs to a class that was derived from `Incrementer`.

There are several issues that demand the attention of the programmer who plans to extend the functionality of

```

1  #include <iostream>
2  using namespace std;
3  #include "incrementer.h"
4
5  int main(){
6      Incrementer* inc_ptr = new Incrementer(95);
7      inc_ptr->increment(4);
8      inc_ptr->increment(5);
9      inc_ptr->increment(6);
10     delete inc_ptr;
11     return 0;
12 }

```

19.4 main.cpp

```

Liskov.out.out
Incrementer object created!
Incrementer value is: 99
Incrementer value is: 4
Assertion ((i > 0) && (i <= 5)) failed in "incrementer.cpp" on line 17

```

Figure 19-2: Results of Running Example 19.4

Incrementer. First, they must be aware of the point of view of the client program that will use the derived object. That code expects certain behavior from Incrementer objects. For example, a client program calling the `increment()` function on Incrementer objects can rely on proper behavior if the arguments to the function satisfy the precondition of being greater than zero or less than or equal to five. If an object derived from Incrementer is substituted at runtime for an Incrementer object, the derived object must not break the client code by behaving in a manner not anticipated by the client program.

Second, with the expectations of the client code in mind, what rules should a programmer follow when extending the functionality of a base class to ensure the derived object continues to live up to or meet the expectations of the client code? These issues are explored further in this section.

Example 19.5 gives the code for a class named `Derived` that extends the functionality of `Incrementer`.

The `Derived` class does essentially the same thing that `Incrementer` does with two exceptions. First, the class invariant of `Derived` states that the `derived_val` property cannot exceed 50. `Derived`'s version of the `checkInvariant()` function is used to enforce this class invariant property. Next, the postcondition of `Derived`'s `increment()` function is tailored to ensure that `derived_val` satisfies `Derived`'s class invariant property.

The precondition of `Derived`'s `increment()` function is the same precondition for `Incrementer`'s `increment()` function. Example 19.6 gives the source code for the `derived.cpp` file.

On line 7 the `Derived` class constructor calls the `Incrementer` constructor and initializes its private attribute `derived_val` using the parameter `i`. On line 9 the `Derived` constructor then calls its version of the `checkInvariant()` function.

`Derived`'s version of the `increment()` function begins on line 16. The precondition is enforced on line 18 followed by a call to `Incrementer`'s version of the `increment()` function on line 19. If the precondition assertion passes then `Incrementer::increment()` should perform flawlessly.

Example 19.7 gives a `main()` function that uses both `Incrementer` and `Derived` objects. Figure 19-3 shows the results of running example 19.7.

Refer to examples 19.6 and 19.7 above when examining figure 19-3. The `Derived` object was successfully used in place of the `Incrementer` object. However, things could have gone horribly wrong. How? What if the precondition for the `Derived` version of the `increment()` function was different from that of the `Incrementer` version? The next section explores this topic.

```

1  #ifndef DERIVED_CLASS
2  #define DERIVED_CLASS
3  #include "incrementer.h"
4
5  class Derived : public Incrementer {
6      /*****
7      Class Invariant: 0 <= derived_val <= 50
8      *****/
9      public:
10     Derived(int i = 0);
11     virtual ~Derived();
12     /*****
13     function: void increment(int i)
14     precondition: 0 < i <=5
15     postcondition: 0 <= derived_val <= 50
16     *****/
17     virtual void increment(int i);
18
19     private:
20     int derived_val;
21     void checkInvariant();
22 };
23 #endif

```

19.5 *derived.h*

```

1  #include "incrementer.h"
2  #include "derived.h"
3  #include <iostream>
4  using namespace std;
5  #include <assert.h>
6
7  Derived::Derived(int i):Incrementer(i),derived_val(i){
8      cout<<"Derived object created!"<<endl;
9      checkInvariant();
10 }
11
12 Derived::~Derived(){
13     cout<<"Derived object destroyed!"<<endl;
14 }
15
16 void Derived::increment(int i){
17     // enforce precondition 0 < i <= 5
18     assert((i > 0) && (i <= 5));
19     Incrementer::increment(i);
20
21     // change incrementer object state
22     if((derived_val+i) <= 50){
23         derived_val += i;
24     }else{
25         int temp = derived_val;
26         temp += i;
27         derived_val = (temp - 50);
28     }
29
30     // enforce class invariant
31     checkInvariant();
32
33     cout<<"Derived value is: "<<derived_val<<endl;
34 }
35
36
37 void Derived::checkInvariant(){
38     assert((derived_val >= 0) && (derived_val <= 50));
39 }

```

19.6 *derived.cpp*

CHANGING THE PRECONDITIONS OF DERIVED CLASS FUNCTIONS

The version of the `increment()` function in class `Derived` discussed above implemented the same precondition as

```

1  #include <iostream>
2  using namespace std;
3  #include "incrementer.h"
4  #include "derived.h"
5
6  int main(){
7      Incrementer* inc_ptr = new Incrementer(95);
8      inc_ptr->increment(4);
9      inc_ptr->increment(5);
10     inc_ptr->increment(3);
11     delete inc_ptr;
12
13     inc_ptr = new Derived(45);
14     inc_ptr->increment(4);
15     inc_ptr->increment(5);
16     inc_ptr->increment(3);
17     delete inc_ptr;
18
19     return 0;
20 }

```

19.7 main.cpp

```

Liskov.out.out
Incrementer object created!
Incrementer value is: 99
Incrementer value is: 4
Incrementer value is: 7
Incrementer object destroyed!
Incrementer object created!
Derived object created!
Incrementer value is: 49
Derived value is: 49
Incrementer value is: 54
Derived value is: 4
Incrementer value is: 57
Derived value is: 7
Derived object destroyed!
Incrementer object destroyed!

```

Figure 19-3: Results of Running Example 19.7

the Incrementer class version, namely, that the integer argument passed to the function was in the range 1 through 5. However, it is possible to specify a different precondition for the Drived class version of increment().

In regards to derived class function preconditions you can go three ways: 1) adopt the same precondition(s), as was illustrated in the previous section, 2) weaken the precondition(s), or 3) strengthen the precondition(s).

Adopting the Same Preconditions

Derived class functions can adopt the same preconditions as the base class methods they override. The increment() function in class Derived, shown in the previous section, adopted the same precondition as the Incrementer class's version of increment(). When a derived class function adopts the same preconditions as its base class counterpart its behavior is predictable from the point of view of any client program using a base class pointer to a derived class object. In other words, you can safely reason about the behavior of a derived class object whose overriding functions adopt the same preconditions as their base class counterparts.

Weakening Preconditions

Derived class functions can weaken the preconditions specified in the base class methods they override. Weakening can also be thought of as loosening or relaxing. The increment() function in class Derived could have weakened the precondition specified in the base class version of increment() by allowing a wider range of increment values to be called as arguments.

Example 19.8 gives a modified version of the Derived class declaration that weakens the preconditions on the increment function. Example 19.9 gives the modified version of the derived.cpp file and example 19.10 shows the modified Derived class object being used in a main() function.

```

1  #ifndef DERIVED_CLASS 19.8 derived.h (weakened
2  #define DERIVED_CLASS precondition)
3  #include "incrementer.h"
4
5
6  class Derived : public Incrementer {
7      /*****
8      Class Invariant: 0 <= derived_val <= 50
9      *****/
10     public:
11         Derived(int i = 0);
12         virtual ~Derived();
13         /*****
14         function: void increment(int i)
15         precondition: 0 < i <=10
16         postcondition: 0 <= derived_val <= 50
17         *****/
18         virtual void increment(int i);
19     private:
20         int derived_val;
21         void checkInvariant();
22     };
23 #endif

```

Notice on line 16 of example 19.8 that the precondition specification for the Derived class version of increment() has been weakened to allow a greater range of increment values. The argument to the increment function now can be anywhere from 1 to 10. The change to the derived.cpp code to implement the weakened preconditions occurs on lines 17 and 18 of example 19.9 on the following page. The main() function shown in example 19.10 below is the same version as that given in example 19.7.

```

1  #include <iostream> 19.10 main.cpp
2  using namespace std;
3  #include "incrementer.h"
4  #include "derived.h"
5
6  int main(){
7      Incrementer* inc_ptr = new Incrementer(95);
8      inc_ptr->increment(4);
9      inc_ptr->increment(5);
10     inc_ptr->increment(3);
11     delete inc_ptr;
12
13     inc_ptr = new Derived(45);
14     inc_ptr->increment(4);
15     inc_ptr->increment(5);
16     inc_ptr->increment(3);
17     delete inc_ptr;
18
19     return 0;
20 }

```

```

1  #include "incrementer.h"
2  #include "derived.h"
3  #include <iostream>
4  using namespace std;
5  #include <assert.h>
6
7  Derived::Derived(int i):Incrementer(i),derived_val(i){
8      cout<<"Derived object created!"<<endl;
9      checkInvariant();
10 }
11
12 Derived::~Derived(){
13     cout<<"Derived object destroyed!"<<endl;
14 }
15
16 void Derived::increment(int i){
17     // enforce precondition 0 < i <= 10
18     assert((i > 0) && (i <= 10)); ← Weakened precondition
19     Incrementer::increment(i);
20
21     // change incrementer object state
22     if((derived_val+i) <= 50){
23         derived_val += i;
24     }else{
25         int temp = derived_val;
26         temp += i;
27         derived_val = (temp - 50);
28     }
29
30     // enforce class invariant
31     checkInvariant();
32
33     cout<<"Derived value is: "<<derived_val<<endl;
34 }
35 }
36
37 void Derived::checkInvariant(){
38     assert((derived_val >= 0) && (derived_val <= 50));
39 }

```

The results of running this program will be exactly the same as those shown in figure 19.3. Remember, that from the point of view of the programmer writing the main() function, the Incrementer class specification is the one that is expected to perform correctly when the program is executed. The precondition for the Incrementer version of the increment() function remains the same, namely, that the range of increment values can be 1 through 5. The Derived class's version of the increment() function can take a wider range of values and therefore the Derived class object performs as expected when substituted for an Incrementer object.

STRENGTHENING PRECONDITIONS

So far you have seen how a Derived class object can be substituted for an Incrementer class object when the Derived class's increment() function adopts the same precondition or weakens the precondition of the Incrementer class's increment() function. When preconditions are kept the same or weakened in the overriding functions of a derived class, objects of the derived class type can be substituted for base class objects with little problem. However, if you happen to strengthen the precondition of an overriding derived class function you will break the code that relies on the original preconditions specified for the base class function.

A strengthening precondition in a derived class function places limits on or restricts the original precondition specified in the base class function it is overriding. In the case of the Incrementer and Derived classes, the preconditions on the Derived version in increment() can be strengthened to limit the range of authorized increment values to, say, 1 through 3. This would effectively break any code that relies on the Incrementer's version of the increment() function that allows the increment values 1 through 5.

So, strengthening preconditions of derived class functions is a bad thing. But, I will give you an example to drive the point home. Example 19.11 gives another version of the Derived class declaration that strengthens the increment() function's precondition to the values 1 through 3 as discussed above.

Example 19.12 gives the modified derived.cpp file. Example 19.13 gives the main() function, which is the same version used in example 19.10.

The difference now, however, when running this program is shown in figure 19-4.


```

1  #ifndef DERIVED_CLASS
2  #define DERIVED_CLASS
3  #include "incrementer.h"
4
5
6  class Derived : public Incrementer {
7      /*****
8       Class Invariant: 0 <= derived_val <= 50
9       *****/
10     public:
11         Derived(int i = 0);
12         virtual ~Derived();
13         /*****
14          function: void increment(int i)
15          precondition: 0 < i <=3
16          postcondition: 0 <= derived_val <= 50
17          *****/
18         virtual void increment(int i);
19     private:
20         int derived_val;
21         void checkInvariant();
22     };
23 #endif

```

19.11 *derived.h (strengthened precondition)*

```

1  #include "incrementer.h"
2  #include "derived.h"
3  #include <iostream>
4  using namespace std;
5  #include <assert.h>
6
7  Derived::Derived(int i):Incrementer(i),derived_val(i){
8      cout<<"Derived object created!"<<endl;
9      checkInvariant();
10 }
11
12 Derived::~Derived(){
13     cout<<"Derived object destroyed!"<<endl;
14 }
15
16 void Derived::increment(int i){
17     // enforce precondition 0 < i <= 3
18     assert((i > 0) && (i <= 3));
19     Incrementer::increment(i);
20
21     // change incrementer object state
22     if((derived_val+i) <= 50){
23         derived_val += i;
24     }else{
25         int temp = derived_val;
26         temp += i;
27         derived_val = (temp - 50);
28     }
29
30     // enforce class invariant
31     checkInvariant();
32
33     cout<<"Derived value is: "<<derived_val<<endl;
34 }
35
36
37 void Derived::checkInvariant(){
38     assert((derived_val >= 0) && (derived_val <= 50));
39 }

```

19.12 *derived.cpp (strengthened precondition)*

← *Strengthened precondition*

```

1  #include <iostream>
2  using namespace std;
3  #include "incrementer.h"
4  #include "derived.h"
5
6  int main() {
7      Incrementer* inc_ptr = new Incrementer(95);
8      inc_ptr->increment(4);
9      inc_ptr->increment(5);
10     inc_ptr->increment(3);
11     delete inc_ptr;
12
13     inc_ptr = new Derived(45);
14     inc_ptr->increment(4);
15     inc_ptr->increment(5);
16     inc_ptr->increment(3);
17     delete inc_ptr;
18
19     return 0;
20 }

```

```

Liskov.out.out
Incrementer object created!
Incrementer value is: 99
Incrementer value is: 4
Incrementer value is: 7
Incrementer object destroyed!
Incrementer object created!
Derived object created!
Assertion ((i > 0) && (i <= 3)) failed in "derived.cpp" on line 18

```

Figure 19-4: Results of Running Example 19.13

Quick Review

The preconditions of a derived class function should either adopt the same or weaker preconditions as the base class function it is overriding. A derived class function should never strengthen the preconditions specified in a base class version of the function. Derived class functions that strengthen base class function preconditions will render it impossible for programmers to reason about the behavior of subtype objects and lead to broken code should the ill-behaved derived class object be substituted for a base class object.

CHANGING THE POSTCONDITIONS OF DERIVED CLASS FUNCTIONS

Derived class function postconditions can be adopted, weakened, or strengthened just like preconditions. However, unlike preconditions, where a weakening condition is preferred to a strengthening condition, the opposite is true for postconditions: A derived class function should specify and implement a stronger, rather than weaker, postcondition.

The `Incrementer` and `Derived` class examples shown previously each had their own private attribute that was part of each class's invariant. (`Incrementer::val` and `Derived::derived_val`) Each class's `increment()` function had a separate postcondition to preserve each class's invariant. The two postconditions did not conflict or contradict and were therefore compatible.

If, on the other hand, `Incrementer::val` had been declared `protected`, and was inherited and used by `Derived`, the `Derived` version of the `increment()` function would need a postcondition that either maintained the class invariant specified by the `Incrementer` class (adopting postcondition) or a postcondition that strengthened `Incrementer`'s class invariant (strengthening postcondition).

A weakening postcondition will cause problems. Consider for a moment what would happen if the `Derived` class version of `increment()` allowed inherited `Incrementer::val` to assume values outside the range of those allowed by

Incrementer's class invariant specification. Disaster would strike the code sooner than later.

SPECIAL CASES OF PRECONDITIONS AND POSTCONDITIONS

Function preconditions can specify and enforce more than just the values of function parameters, and postconditions can specify and enforce more than just class invariant states.

A function precondition can, for example, specify that the class invariant must hold true or that a combination of conditions hold true before it can do its job properly. A function postcondition can, in addition to enforcing the class invariant, specify the state of the object or reference the function returns (if any), or it can specify any number of conditions hold true upon completion of the function call. The conditions or combination of conditions imposed by derived class overriding function preconditions and postconditions can be weakening or strengthening.

The weakening and strengthening effects of preconditions and postconditions can apply to more than just simple conditions. Function parameter types, return types, and function access rights all play a part and are discussed below.

FUNCTION ARGUMENT TYPES

Derived class function preconditions can be weakened or strengthened by their function parameter types. An overriding function must agree with the function it overrides in the type, number, and order of its function parameters. (see chapter 13) Function parameter types can belong to a type hierarchy. This means that a function parameter might be related to another class via a subtype or supertype relationship.

A derived class function specifying a parameter that is a base class to the matching parameter declared in its base class counterpart is an overriding function. If, however, the derived class function declares a parameter that is a subclass of the parameter type declared in the base class function then the derived class function hides the base class version of the function. This is due to the transitive nature of subtypes. (i.e., Given two classes, B and D, if D is derived from B, then D is a B but a B is not a D)

In other words, an overriding function can only provide a weakening precondition with regards to parameter types because to strengthen the parameter type required would result in the declaration of a new function (requiring a new type from the point of view of the base class version of the function) not the overriding of the virtual function. To illustrate this point assume there exists the class inheritance hierarchy shown in figure 19-5.

Each function `f()` in each class A, B, and C, requires a reference to an object of type A. Each function overrides the previous function. If an A type pointer is created and initialized to point to a C type object, then the C version of the function will be called as expected. This is demonstrated using the `main()` function shown in example 19.14. The results of running the program are shown in figure 19-6.

Referring to example 19.14, the `main()` function begins on line 7. On lines 8 through 10 three objects are created, one of each type A, B, and C. An A type pointer named `a_ptr` is declared on line 12 and initialized to point to a C object. The `f()` function is then called via `a_ptr` using the three objects of type A, B, and C as arguments. As you can see in figure 19-6 everything runs as expected.

Now, let us change one thing. Let us change the signature of the C class version of the `f()` method to take an argument of type B, then rerun example 19.14. Figure 19-7 gives the results. As you can see, changing C's version of the `f()` function to take a different type (subtype of A) results in a hiding function (not called polymorphically) vs. an overriding function (one that will be called polymorphically).

In short, an overriding function can specify a weaker (supertype) parameter than originally called for in the base class version of the function. A stronger (subtype) parameter type will result in a hiding function.

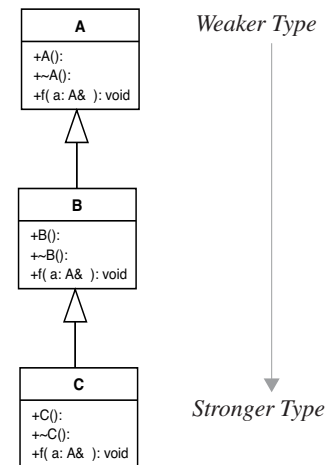


Figure 19-5: Inheritance Hierarchy Showing Weaker and Stronger Types

19.14 main.cpp

```

1  #include <iostream>
2  using namespace std;
3  #include "a.h"
4  #include "b.h"
5  #include "c.h"
6
7  int main(){
8      A a1;
9      B b1;
10     C c1;
11
12     A* a_ptr = new C();
13
14     a_ptr->f(a1);
15     a_ptr->f(b1);
16     a_ptr->f(c1);
17
18     delete a_ptr;
19     return 0;
20 }

```

```

Liskov_Special.out.out
Object A created!
Object A created!
Object B created!
Object A created!
Object B created!
Object C created!
Object A created!
Object B created!
Object C created!
C::f() called.
C::f() called.
C::f() called.
Object A destroyed!
Object C destroyed!
Object B destroyed!
Object A destroyed!
Object B destroyed!
Object A destroyed!
Object A destroyed!
|

```

Figure 19-6: Results of Running Example 19.14

```

Liskov_Special.out.out
Object A created!
Object A created!
Object B created!
Object A created!
Object B created!
Object C created!
Object A created!
Object B created!
Object C created!
B::f() called.
B::f() called.
B::f() called.
Object A destroyed!
Object C destroyed!
Object B destroyed!
Object A destroyed!
Object B destroyed!
Object A destroyed!
Object A destroyed!

```

Figure 19-7: Results of Running Example 19.14 with Modified C Class Function

FUNCTION RETURN TYPES

Function return types are considered special cases of postconditions. An object or reference to an object may be returned from a function as a result of its execution. Refer again to the inheritance hierarchy illustrated in figure 19-5. If a snippet of client code expects a return type from a function to be of a certain type, the function can strengthen that condition and return a subtype of the type expected. This strengthening of return types is in line with the strengthening usually required of postconditions.

FUNCTION ACCESS RIGHTS

Function access rights are a special case of preconditions. In C++, access to an object's functions are controlled via the access specifiers public, protected, and private. The general rule-of-thumb regarding access rights is that an overriding function should be made available to the same, or wider, audience as the base class function it overrides. Thus, a relaxing of the access rules for overriding functions is a weakening precondition, which is acceptable.

However, in C++, when designing for polymorphic behavior using virtual functions, the accessibility of an overriding derived class function is of no concern to client code accessing the function via a base class pointer. To illustrate this point let us make another change to the C class version of the f() function. I will change its parameter type back to A, and its visibility to private. The code for the modified c.h file appears in example 19.15.

```

1  #ifndef C_H
2  #define C_H
3  #include "b.h"
4  #include <iostream>
5  using namespace std;
6
7  class C : public B {
8  public:
9      C(){cout<<"Object C created!"<<endl;}
10     ~C(){cout<<"Object C destroyed!"<<endl;}
11 private:
12     virtual void f(A& a){ /* expects reference of type A */
13         cout<<"C::f() called."<<endl;}
14 };
15 #endif

```

19.15 c.h

This version of class C will be used in the same version of the main() function shown in example 19.14. Because the C class version of the f() function is accessed polymorphically via an A type base class pointer, the private C::f() is called because the A::f() version is publicly accessible. Users of A::f() need no knowledge of the accessibility of C::f(). Everything works as expected. However, if access to C::f() were attempted via a C object then an error would result since C::f() is private to clients of the C interface. Figure 19-8 shows the results of running example 19.14 with the modified C class shown above.

C::f() is called even though it is declared private. This is because it's being accessed via an A pointer and A::f() is public.

Figure 19-8: Results of Running Example 19.14 Using Private C::f() Overriding Function

Quick Review

Function parameter types are considered special cases of preconditions. Preconditions should be weakened in the overriding function, therefore, parameter types should be the same or weaker than the parameter types of the function being overridden. A base class is considered a weaker type than one of its subclasses.

Function return types are considered special cases of postconditions. The return type of an overriding function should be stronger than the type expected by the client code. A subclass is considered a stronger type than its base class.

Function access rights are considered special cases of preconditions. Access rights should be kept the same or weakened for an overriding function. However, polymorphic behavior using base class pointers to derived class objects renders this a mute point in the C++ language. The accessibility of an overriding function may change in a derived class, but, from the point of view of client code using the interface published by a base class pointer type, the accessibility of the base class function is what counts.

THREE RULES OF THE SUBSTITUTION PRINCIPLE

In their book *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, Barbara Liskov and John Guttag say that the substitution principle must support three properties: the signature rule, the methods rule, and the properties rule. Each of these rules are discussed below.

SIGNATURE RULE

The signature rule deals with the functions published or made public by a type specification. In C++ these functions would have public accessibility. For a subtype to obey the signature rule it must support all the functions published by its base class and that each overriding function is compatible with the function it overrides. C++ enforces this type compatibility, although, as was shown in the previous section, the accessibility of derived class overriding functions (public, protected, and private) can be effectively ignored when accessing derived class functions via a base class pointer.

METHODS RULE

The methods rule says that calls to overriding functions should behave like the base class functions they override. A type may be substitutable from a strictly type perspective but the behavior may be all wrong. Correct behavior of overriding functions is the aim of LSP and DbC.

PROPERTIES RULE

The properties rule is concerned with the preservation of provable base class properties by subtype behavior. A subtype should preserve the base class invariant. If a subtype's behavior violates a base class invariant then it is breaking the properties rule.

THE OPEN-CLOSED PRINCIPLE

Software systems change over time. Change takes many forms, but changing and evolving system requirements provide the primary catalyst. A software system must accommodate change. It must evolve gracefully throughout its useful lifecycle. A software system that is rigid, fragile, and change-resistant exhibits bad design. A software system that is resilient, flexible, and extensible possesses the hallmark characteristics of thoughtful object-oriented architecture. The open-closed principle (OCP) provides the necessary framework for achieving an extensible and accommodating software architecture.

Formulated by Bertrand Meyer, the open-closed principle makes the following assertion:

Software modules must be designed and implemented in a manner

that opens them for extension but closes them for modification.

Said another way, changes to software modules should be avoided and new system functionality added by writing new code.

It should be noted that writing code that is easy to extend and maintain is a requirement in and of itself. Writing such code takes longer initially but pays a big dividend later. I call it the design dividend.

ACHIEVING THE OPEN-CLOSED PRINCIPLE

The key to writing code that conforms to the open-closed principle is to depend upon abstractions, not upon implementations. The reason — abstractions tend to be more stable. (Correctly designed abstractions are very stable!) This is achieved in C++ through the use of abstract base classes and dynamic polymorphic behavior. Code should rely only upon the interface functions and behavior promised via abstract base class interfaces. A code module that relies only upon abstractions will exhibit the characteristic of being closed to the need for modification yet open to the possibility of extension.

AN OCP EXAMPLE

A good example of code written with the OCP in mind is the fleet simulation model originally discussed and presented in chapter 13. Figure 19-9 gives the UML class diagram showing the static relationship between the classes that comprise the fleet simulation model.

The model foundation consists of three core abstractions: Vessel, Weapon, and Plant. A Vessel is comprised of a Plant and a Weapon. Vessel depends only upon these abstractions and is therefore closed to modification. Weapon depends on nothing as does Plant.

Plant and Weapon can be extended as necessary to create new power plant and weapon system types. Vessel cares not about the new types, so long as the new types abide by the LSP and DbC and provide the behavior promised by the abstractions they extend.

Vessel itself can be extended to create new vessel types. A new vessel type still depends only upon the Weapon and Plant abstractions, and of course upon the Vessel class it extends.

Additional OCP Conventions

There are also a few other conventions the fleet simulation code abides by that brings it further in line with the OCP. All member attributes are kept private, there are no global variables, and the Runtime Type Identification (RTTI) mechanism is not used to determine what types are being dealt with in the code.

RELATIONSHIP BETWEEN THE OCP AND THE LSP/DbC

The OCP and LSP/DbC share a close relationship. Code written with the OCP in mind depends upon behavior promised by abstract base class specification. Depending upon promised behavior enables the subtype reasoning process. The LSP/DbC is used to achieve the proper subtype behavior within a type hierarchy thereby enabling the OCP.

Quick Review

The open-closed principle (OCP) attempts to optimize object-oriented software architecture design so it can accommodate change. Software modules should be designed so they are closed to modification yet open to extension. The OCP is achieved by depending only upon software abstractions. In C++ this means designing with abstract base classes with the goal of dynamic polymorphic behavior. The OCP relies heavily upon the Liskov substitution principle and design by contract (LSP/DbC).

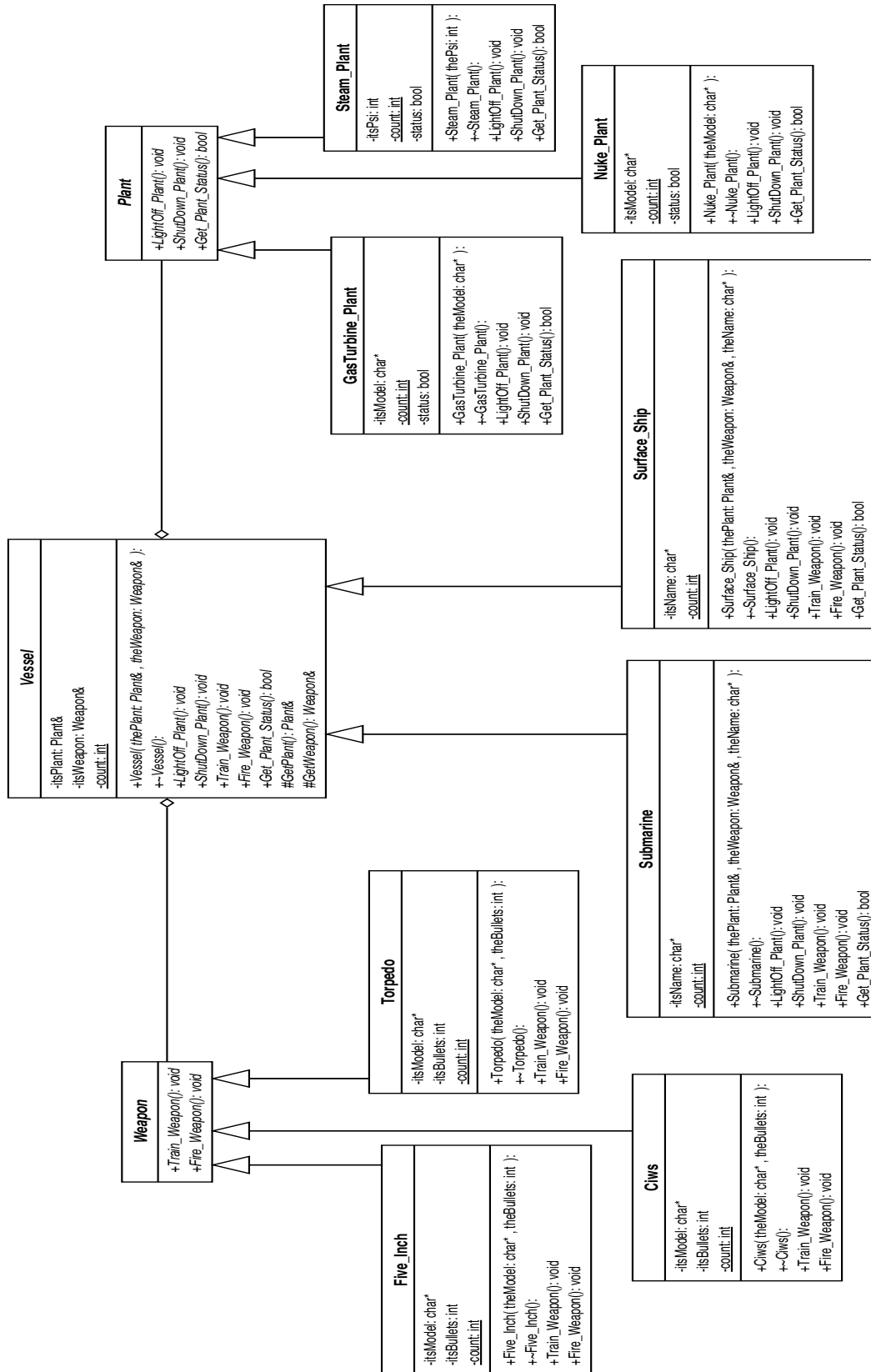


Figure 19-9: Fleet Simulation Model Class Diagram

THE DEPENDENCY INVERSION PRINCIPLE

When used together in a disciplined approach, the OCP and the LSP/DbC yield a desirable inversion of program module dependencies that is different from the usual top-down module dependencies attained with functional decomposition. This dependency inversion is generalized into a principle in its own right known as the dependency inversion principle (DIP). Robert C. Martin stated the definition of the DIP in two parts:

*“A. High-level modules should not depend upon low-level modules.
Both should depend upon abstractions.*

*B. Abstractions should not depend upon details. Details should
depend upon abstractions.”*

CHARACTERISTICS OF BAD SOFTWARE ARCHITECTURE

When a software module depends on the details of a lower-level software module it is hard to change and hard to reuse. Consider the software module hierarchy shown in figure 19-10 where high-level modules depend on low-level modules.

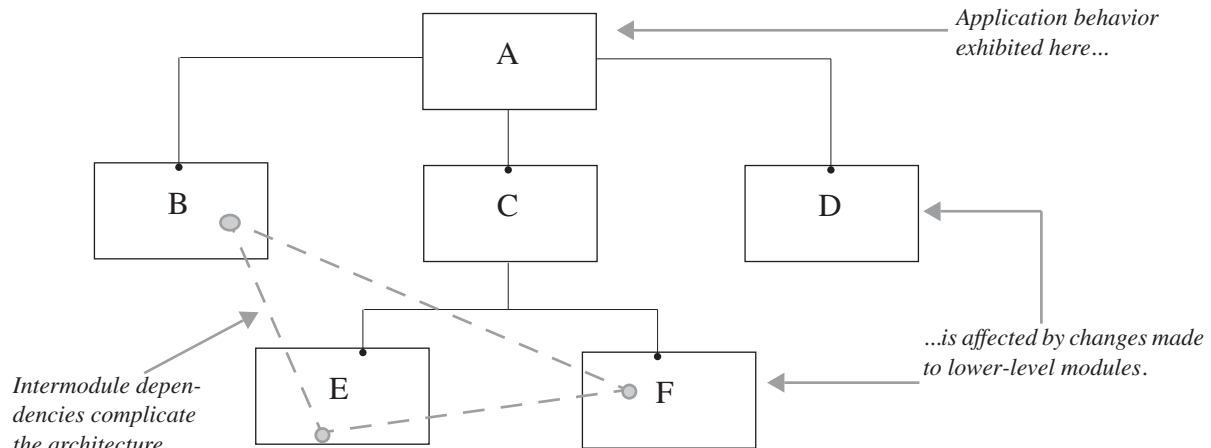


Figure 19-10: Procedure-Oriented Software Module Hierarchy

The behavior of module A depends on modules B, C, and D. The behavior of module C depends of the behavior of modules E and F. A change to module E affects module C which, in turn, affects module A. Any intermodule dependencies, such as global variables, will further complicate the issue. A complex software system sporting this sort of architecture will have the undesirable characteristics of bad design, namely, it will be fragile, rigid, and immobile.

A fragile software architecture is one that breaks in unexpected ways when a change is made to one or more software modules. Fragile software leads to rigid software.

A rigid software architecture is one that is so difficult and painful to change that programmers do not want to change it.

An immobile software architecture is characterized by the inability to successfully extract the software module for reuse in another system. The software module may exhibit desirable behavior but if it is too dependent on other modules or anchored to the application architecture by intermodule dependencies then it will be difficult if not impossible to reuse in another similar context. If it is easier to rewrite a module from scratch than it is to adopt and reuse the module then the module is immobile.

CHARACTERISTICS OF GOOD SOFTWARE ARCHITECTURE

Object-oriented software architectures that subscribe to the OCP and the LSP/DbC will depend heavily upon abstractions. These abstractions will appear at or near the top of the software module hierarchy. Refer again to the fleet simulation model class diagram shown in figure 19-9. The Vessel, Weapon, and Plant abstract base classes serve as the foundation for all behavior inherited by the lower-level implementation classes. This inheritance relationship means that the lower-level derived classes are dependent upon the behavior specified by the higher-level base class abstractions.

The key to success with the DIP lies in choosing the right software abstractions. A software architecture based upon the right kinds of abstractions will exhibit the desirable characteristic of being easy to extend. It will be flexible because of its extensibility, it will be non-rigid in that the addition of new functionality via new derived classes will not affect the behavior of existing abstractions. Lastly, software modules that depend upon abstractions can generally be reused in a wider variety of contexts, thus achieving a greater degree of mobility.

SELECTING THE RIGHT ABSTRACTIONS TAKES EXPERIENCE

The ability to identify essential software component abstractions takes practice and experience. However, applying the OCP and the LSP/DbC in your object-oriented software architecture design will yield a better design, even if you do not get all the abstractions right the first time around.

Quick Review

The OCP and the LSP/DbC, when applied together, result in the realization of a third design principle known as the dependency inversion principle (DIP). The key to the DIP is that high-level software modules should not rely on low-level details, and that software modules at all hierarchy levels should rely only upon abstractions. When a software architecture achieves the goals of the DIP it is easier to extend and maintain (flexible and non-rigid). Software modules that conform to the DIP are easier to reuse in other contexts (mobile).

SUMMARY

From a programmer's perspective, a well-designed, object-oriented architecture manifests itself as an inheritance hierarchy, including a set of abstract data type vertical and horizontal relationships, that exhibits several key characteristics. It is 1) easy to understand, 2) easy to reason about, and 3) easy to extend.

The preconditions of a derived class function should either adopt the same or weaker preconditions as the base class function it is overriding. A derived class function should never strengthen the preconditions specified in a base class version of the function. Derived class functions that strengthen base class function preconditions will render it impossible for programmers to reason about the behavior of subtype objects and lead to broken code should the ill-behaved derived class object be substituted for a base class object.

The open-closed principle (OCP) attempts to optimize object-oriented software architecture design so it can accommodate change. Software modules should be designed so they are closed to modification yet open to extension. The OCP is achieved by depending only upon software abstractions. In C++ this means designing with abstract base classes with the goal of dynamic polymorphic behavior. The OCP relies heavily upon the Liskov substitution principle and design by contract (LSP/DbC).

The OCP and the LSP/DbC, when applied together, result in the realization of a third design principle known as the dependency inversion principle (DIP). The key to the DIP is that high-level software modules should not rely on low-level details, and that software modules at all hierarchy levels should rely only upon abstractions. When a software architecture achieves the goals of the DIP it is easier to extend and maintain (flexible and non-rigid). Software modules that conform to the DIP are easier to reuse in other contexts (mobile).

TERMS AND DEFINITIONS

The terms and definitions listed in table 19-1 were used throughout this chapter:

Term	Definition
<i>Abstraction</i>	The separation of the important from the unimportant. (i.e. interface vs. implementation)
<i>Abstract Data Type</i>	A type specification that separates the interface to the type from the type's implementation. An abstract data type represents a set of objects that can be manipulated via a set of interface functions.
<i>Supertype</i>	An abstract data type that serves as a specification for related subtypes.
<i>Subtype</i>	An abstract data type that derives all or part of its specification from another abstract data type. A subtype can inherit the specification of a supertype then add something extra if required.
<i>Type Specification</i>	A declaration of the behavioral properties of an abstract data type. A specification describes the important characteristics of the data abstraction.
<i>Encapsulation</i>	The act of hiding private implementation details behind a publicly accessible interface.
<i>Precondition</i>	A condition, constraint, or set of constraints that must hold true during a call to an abstract data type interface function to ensure its proper operation.
<i>Postcondition</i>	A condition, constraint, or set of constraints that must be satisfied when an abstract data type function completes execution.
<i>Inheritance Hierarchy</i>	A set of abstract data type specifications that implement a supertype and subtype relationship between each abstract data type.
<i>Class</i>	The declaration of an abstract data type specifying a set of attributes and interface functions common to a set of objects.
<i>Abstract Class</i>	The declaration of an abstract data type specifying a set of attributes and interface functions common to a set of objects. Interface function implementations are deferred to subclasses. In C++ interface functions in an abstract class are declared to be pure virtual.
<i>Subclass</i>	A declaration of an abstract data type taking all or part of its specification from another, possibly abstract, class.
<i>Class Invariant</i>	An assertion about the state of an object which must hold true for all possible states the object may assume.

Table 19-1: Terms and Definitions Related to the LSP

Skill Building Exercises

1. **Research:** Procure a copy of Bertrand Meyer's excellent book *Object-Oriented Software Construction, Second Edition*, and read it from front to back.
2. **Research:** Procure a copy of Robert C. Martin's excellent book *Designing Object-Oriented C++ Applications Using The Booch Method*. (Although the Booch diagramming notation has been largely superseded by the Unified Modeling Language, this book is still, in my opinion, a must-read for intermediate and advanced students of the C++ language.)

3. **Research:** Conduct a web search for the keywords Liskov substitution principle, open-closed principle, dependency inversion principle, and Meyer design by contract programming.

SUGGESTED PROJECTS

1. **RobotRat:** Evaluate your latest version of RobotRat and apply each of the three design principles to your design. What improvements, if any, can be realized by applying each principle? How would your design have to be modified to take full advantage of each of the three design principles?

SELF TEST QUESTIONS

1. List and describe the preferred characteristics of an object-oriented architecture.
2. State the definition of the Liskov substitution principle.
3. Define the term class invariant.
4. What is the purpose of a function precondition?
5. What is the purpose of a function postcondition?
6. List and describe the three rules of the substitution principle.
7. Write the definition and goals of the open-closed principle.
8. Explain how the open-closed principle uses the Liskov substitution principle and Meyer design by contract programming to achieve its goals.
9. Write the definition and goals of the dependency inversion principle.
10. Explain how the dependency inversion principle builds upon the open-closed principle and the Liskov substitution principle/Meyer design by contract programming.

REFERENCES

Barbara Liskov, *Data Abstraction and Hierarchy*, SIGPLAN Notices, 23,5 (May 1988).

W. Al-Ahmad, *On The Interaction of Programming By Contract and Liskov Substitution Principle*.

Bertrand Meyer, *Applying "Design by Contract"*, IEEE Computer, Vol. 25 Number 10, October 1992, pp. 40 - 51.

Barbara H. Liskov, Jeannette M. Wing, *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems, Vol 16, No 6, November 1994, pp. 1811-1841.

James O. Coplien, *Advanced C++: Programming Styles and Idioms*. Addison-Wesley Publishing Company,

Reading, Massachusetts, 1992. ISBN: 0-201-54855-0

Barbara Liskov, John Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, Massachusetts, 2001. ISBN: 0-201-65768-6

Robert C. Martin. *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice Hall, Englewood Cliffs, New Jersey, 1995. ISBN: 0-13-203837-4

Bertrand Meyer. *Towards practical proofs of class correctness*, to appear in Proc. 3rd International B and Z Users Conference (ZB 2003), Turku (Finland), June 2003, ed. Didier Bert, Springer-Verlag, 2003.

Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Printice Hall PTR, Upper Saddle River, New Jersey 07458. ISBN: 0-13-629155-4

NOTES

CHAPTER 20



Rusty Shackle

Using A UML Modeling Tool

LEARNING OBJECTIVES

- *STATE THE PURPOSE OF A UML MODELING TOOL*
- *LIST KEY UML MODELING FEATURES SUPPORTED BY EMBARCADERO TECHNOLOGIES' DESCRIBE™*
- *UTILIZE USE-CASE, SEQUENCE, AND CLASS DIAGRAMS TO ANALYZE AND DESIGN A SOLUTION TO A GIVEN PROGRAMMING PROBLEM*
- *UTILIZE DESCRIBE™ TO DEVELOP A SOLUTION TO A GIVEN PROGRAMMING PROBLEM UP TO THE POINT OF C++ CODE GENERATION*
- *SELECT THE APPROPRIATE UML DIAGRAM BASED ON THE CORRESPONDING PROBLEM ANALYSIS OR DESIGN PHASE*
- *EMPLOY THE UML CONSTRUCTS OF AGGREGATION AND GENERALIZATION TO CREATE COMPLEX CLASS RELATIONSHIPS*
- *UTILIZE DESCRIBE™ TO REVERSE ENGINEER EXISTING C++ SOURCE CODE*
- *UTILIZE DESCRIBE™ TO GENERATE A WEB-BASED PROJECT REPORT*

INTRODUCTION

Throughout this book I have narrowly focused your expose to the Unified Modeling Language (UML) and concentrated on the class diagram construct, showing you how to use class diagrams to model the static relationship between classes in an application design. Even a limited amount of modeling goes a long way towards expressing and clarifying software design ideas. But there is more to the UML than just class diagrams.

The UML is a rich modeling language that can be employed to assist you in your complete round-trip software system engineering effort. However, the key to tapping the full modeling potential of the UML lies in finding the right UML modeling tool. This chapter shows you what a good UML modeling tool can do for you.

There are many types of UML modeling tools on the market today. Some are just fancy drawing tools that do little more than help you draw the different UML diagrams, while others, like Embarcadero Technologies' Describe™, provide an integrated modeling and development environment (IMDE). There are advantages and disadvantages to both types of tools with the central issue being cost vs. features. The simple drawing tools are least expensive and often available as freeware or shareware. The comprehensive tools like Describe™, Rational Rose™, or Sybase's PowerDesigner™ do more for you but you pay extra for the privilege of using a comprehensive design tool.

Included with the paperback and CD ROM version of this book are two UML modeling tools. The first is a demo version of Embarcadero Technologies' Describe™ UML modeling tool. Describe™ is used in this chapter to demonstrate how a good UML modeling tool can significantly augment your software system analysis, design, and implementation efforts. If you downloaded the ebook/PDF edition of this book you can download Describe™ from the Embarcadero Technologies website: [<http://embarcadero.com>]. Describe™ runs on the Microsoft Windows™ operating system.

The second UML tool is called ObjectPlant™ and runs on Macintosh™ computers. ObjectPlant™ is shareware and can be downloaded from [<http://www.arctaedi.us.com/ObjectPlant/>]. I have used ObjectPlant™ extensively throughout this book to generate encapsulated postscript versions of the class and sequence diagrams.

There are many other UML modeling tools available for both Macintosh™ and Windows™ machines. If you search the web long enough you will eventually discover them all. However, I have found that there are many more UML tools available for the Windows™ platform than for Macintosh™.

Before moving on it is worth noting that the UML, while potentially powerful in its expressive capability, is just another tool you can add to your software development tool bag. It can help you — but it will not design your application for you or write every line of your code. (Although tools like Describe™ and ObjectPlant™ will write some of the code!)

Also, this chapter is not a complete treatment of Describe™ nor is it intended as a comprehensive treatment of UML modeling. There are many great books on the UML topic alone with some of the best having been published by Addison-Wesley in their Object-Technology Series. A few of the books in that series are listed as references at the end of the chapter. A lot of great information about the UML can also be found on the Object Management Group (OMG) website: [<http://www.omg.org/>].

THE PURPOSE OF A UML MODELING TOOL

The purpose of a UML modeling tool is to assist the object-oriented analyst, designer, or programmer in the analysis, design and implementation of their object-oriented software system. A modeling tool also plays a critical role in communicating software designs to other analysts, designers, programmers, managers, and customers. A good UML modeling tool is therefore both a design and communication tool at the same time.

Keep in mind, however, that a modeling tool is simply one of many tools available to software professionals to help them get their jobs done, much like an integrated development environment (IDE) or a simple text editor. In fact, different modeling tools compare with each other in much the same way as IDEs compare to text editors. A text editor will help you create source code files. An IDE will let you create files, compile files, debug programs, and help you with many other programming tasks as well. A simple UML modeling tool might let you draw and print diagrams, but a comprehensive modeling tool will let you effectively manage the entire software system design process.

INTRODUCING EMBARCADERO TECHNOLOGIES' DESCRIBE™

Embarcadero Technologies' Describe™ is a good example of a comprehensive UML modeling tool. Describe™ will be used throughout the remainder of this chapter to develop an object-oriented solution to a modified version of the robot rat project solved earlier in a procedural fashion in chapter 3. This section presents an overview of the Describe™ feature set. For a complete description of Describe™ features visit the Embarcadero Technologies web-site.

PRIMARY FEATURES

Describe is an integrated modeling and development environment (IMDE) that integrates well with third-party requirements management programs like Telelogic's DOORS™, testing tools like Mercury Test Director™, and traditional IDEs like Microsoft Visual C++. You can use describe to create and maintain the full range of UML diagrams. Diagrams can be thoroughly documented and, if necessary, linked to external documentation sources like text files, Microsoft Word™ documents, or web sites. Diagrams can be exported into various graphic file formats like JPEG. An entire system's worth of documentation and diagrams can be exported via standard or customized web reports.

Describe can generate C++ and Java source code from class diagrams, or it can create class diagrams from existing C++ or Java source code. Creating class diagrams from source code is known as reverse engineering.

Describe can run in stand alone mode or connect to a design data repository as figure 20-1 depicts.

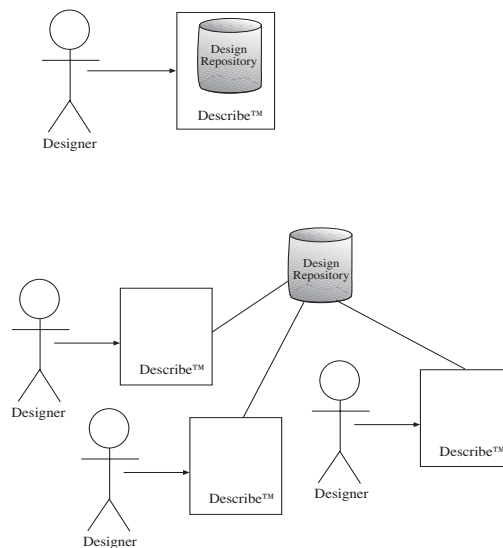


Figure 20-1: Describe User Modes

This lets designers create and edit system designs on their own machine or collaborate with other designers during the design process.

Describe™ can also be integrated with source code version control systems like CVS or PVCS. A source code version control system is an integral part of a what is usually an enterprise-wide product configuration management (CM) system that includes tools (such as Describe™), processes (such as regular design reviews, etc.), and good people like you.

Describe™ integrates with other Embarcadero Technologies design tools to extend its functionality. It can also be customized in many ways using Microsoft Visual C++, Visual Basic, or Java.

THE PROJECT SPECIFICATION: ROBOT RAT

To demonstrate how a UML design tool can help I will revisit the robot rat project first presented in chapter 3. This version of robot rat will have much more capability than the version designed and coded in chapter 3. The project specification is given in figure 20-2 below.

```

                                Programming Project
                                Robot Rat

Objectives:
    Utilize the following C++ language features and constructs in a program:
    •Variables
    •Constants
    •Statements
    •Expressions
    •Functions
    •Virtual functions
    •Pure virtual functions
    •Enumerated types
    •Arrays
    •Pointers
    •References
    •Ad hoc polymorphism (operator overloading)
    •Abstract base classes
    •Concrete classes
    •Aggregation
    •Association
    •Inheritance
    •Dynamic polymorphic behavior
    •Model-View-Controller (MVC) pattern

Task:
    Write a program that will allow users to create multiple robot rats and control
    their movements around a floor. The user should be able to interact with the program via
    a simple, text-based menu that might have the following choices:

    1. Add Rodent
    2. Toggle Rodent
    3. Turn Left
    4. Turn Right
    5. Set Tail Up
    6. Set Tail Down
    7. Move
    8. Display Floor
    9. Exit

Design Approach and Hints:
    Create an inheritance hierarchy that will allow you, the programmer, the flexibility
    to add other types of remote controlled objects besides robot rats.
    Use a separate class to handle all user interface functions such as menu display and
    floor pattern display. Use only iostream for input and output. Use another class to man-
    age the creation and movement of robot rat objects, and another class to coordinate mes-
    sage passing between the robot rat manager class and the user interface class.
```

Figure 20-2: Robot Rat Project Specification

As you can see by comparison with the earlier robot rat project there is much more to do here. The primary differences are: 1) The program must be able to create multiple robot rat objects and control their movements around the floor, 2) the program's architecture must support the creation of different types of remote controlled objects besides rats. Perhaps at some point in the future remote controlled people objects will have to be added. The application architecture must be designed to facilitate such anticipated changes, and 3) the notion of the rats and the visually displayed floor upon which they roam are now separated. All user interface functions, including the floor pattern display, are to be handled by a class that is separate from the class that actually controls or manages robot rat objects.

All these additional robot rat requirements call for the design and implementation of quite a different program from the one created to satisfy the original robot rat requirements.

CREATING USE CASE DIAGRAMS

The new robot rat project specification offers a lot to think about. The best way to begin to tackle the project is to solidify your understanding of the requirements by creating several use case diagrams. Start by creating a new project in Describe™ to which you can then add the necessary diagrams. Figure 20-3 shows the Create New System window with the name of the new system being set as RobotRat.

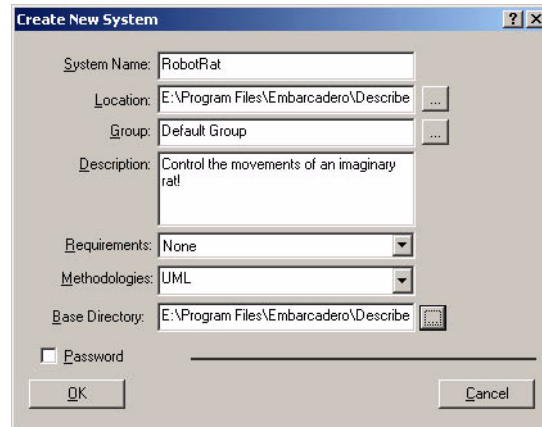


Figure 20-3: Creating the New RobotRat System

In the Create New System window you can set the system location, assign it to a particular group, add a brief description, and define the type of methodology to use in the diagrams. You can additionally link to a requirements document or integrate with the Telelogic DOORS™ requirements management system, and password protect the system. Click the OK button to create the RobotRat system.

With the RobotRat system created you can now add the use case diagrams. From the File menu select New Diagram as shown in figure 20-4.

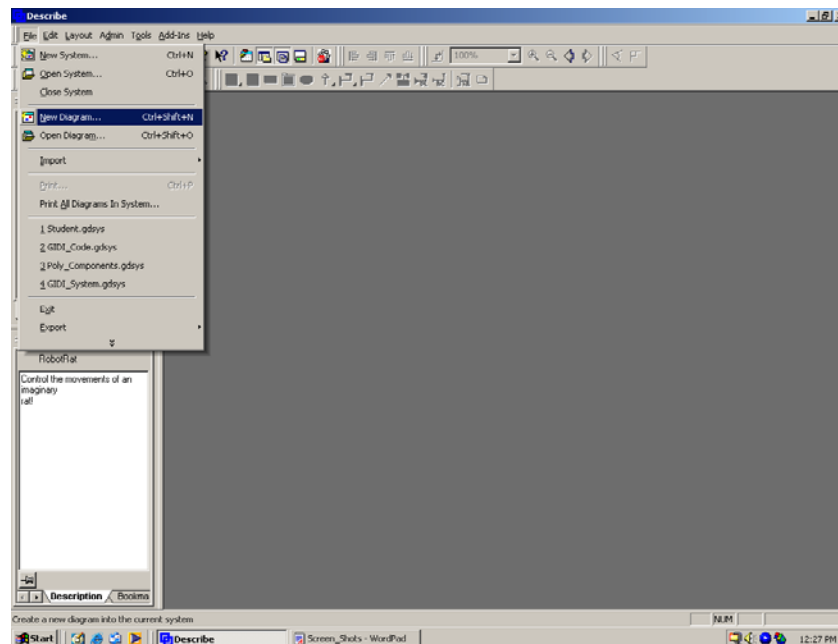


Figure 20-4: Adding a New Diagram

You will get a dialog window asking you to name the new diagram and to select the type of diagram you wish to create as shown in figure 20-5.

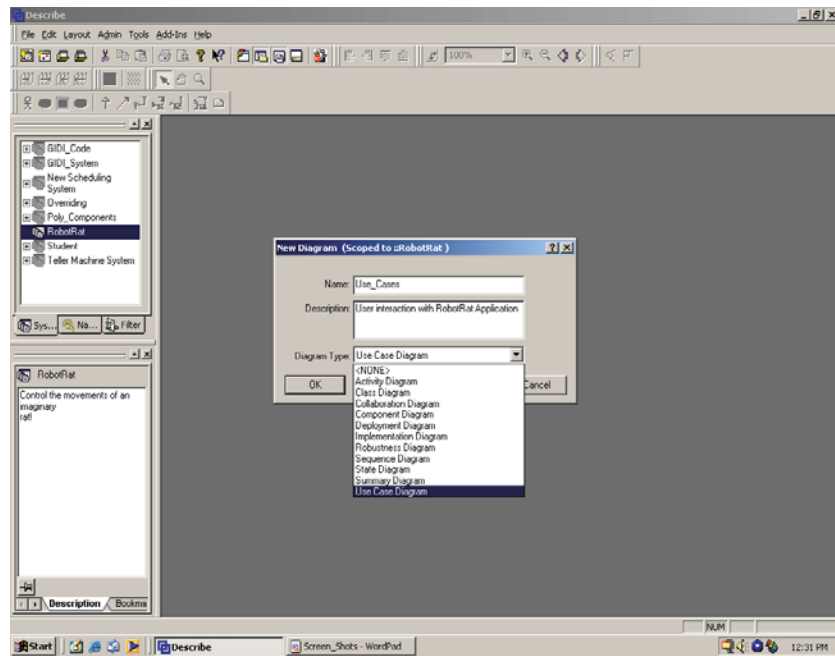


Figure 20-5: Creating Use Case Diagram

When you have named the diagram (I named this one Use_Cases) and have selected the diagram type click OK and you are ready to create the use case diagram for the robot rat project. Figure 20-6 shows a partial use case diagram for robot rat functionality from the user's perspective with a new use case named Create New RobotRat being added.

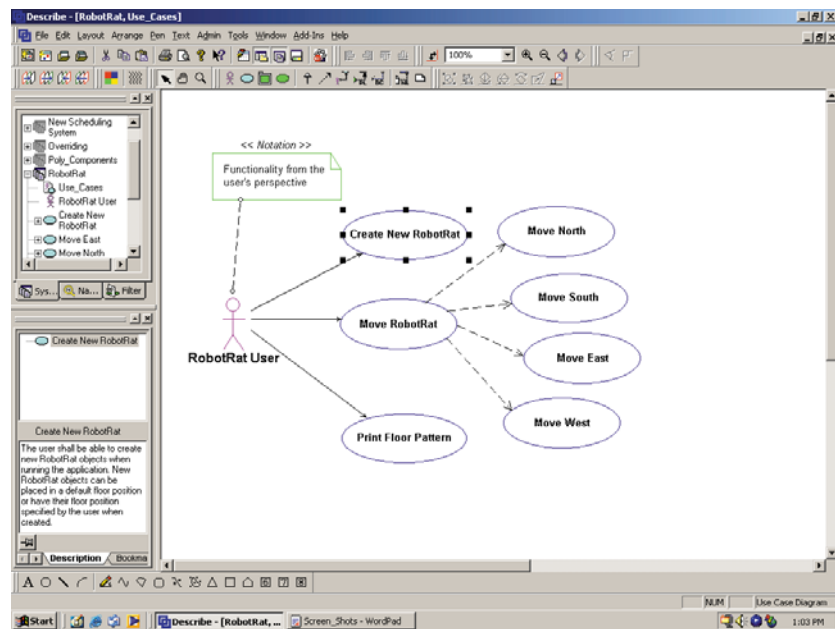


Figure 20-6: User Perspective Use Cases

The various use case diagram elements can be added by selecting them from the tool bar. Diagram elements can be thoroughly documented when they are added to the diagram.

Adding Documentation to Diagram Elements

Figure 20-7 shows the Properties Editor window for the Add New RobotRat use case. In addition to adding brief

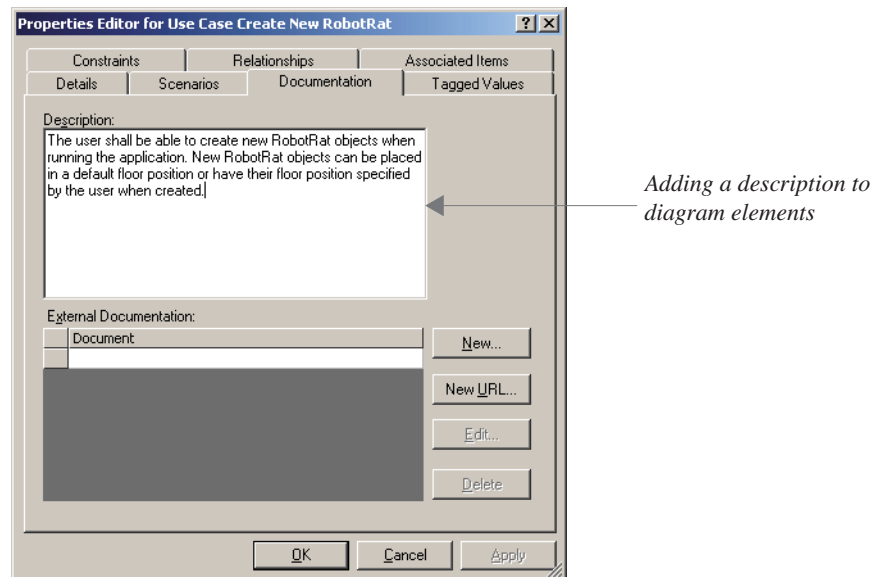


Figure 20-7: Adding Documentation via the Properties Editor Window

descriptions to diagram elements Describe™ offers the capability to link each diagram element to supporting documentation. Figure 20-8 shows the Properties Editor window for the RobotRat User actor. Documenting your diagrams as you build them is a good habit to develop and offers many benefits. (I like to call this “document-as-you-go”) Primarily, it helps you clarify your design to yourself. If you have trouble explaining the reason for each diagram element to yourself then perhaps you need to rethink the need for the element.

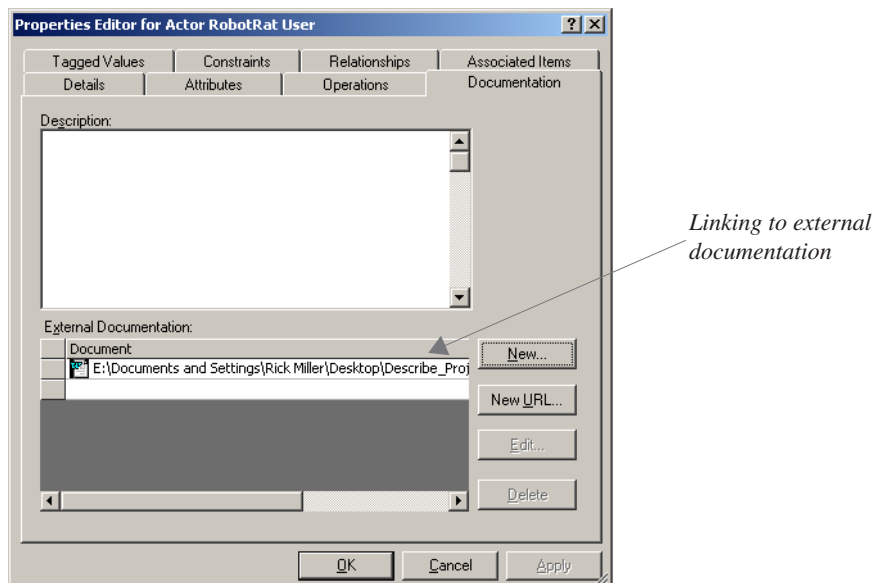


Figure 20-8: Linking to External Documentation via the Properties Editor Window

The next obvious benefit of document-as-you-go is you do not have to stop your design and development efforts to go back and create all your documentation. If you practice document-as-you-go you will end up with more than enough design documentation, in addition to any diagrams you create. You will see the results of document-as-you-go

when you generate project web reports. Figure 20-9 shows the completed robot rat user’s perspective use case diagram. Each diagram element is documented and linked to the requirements document.

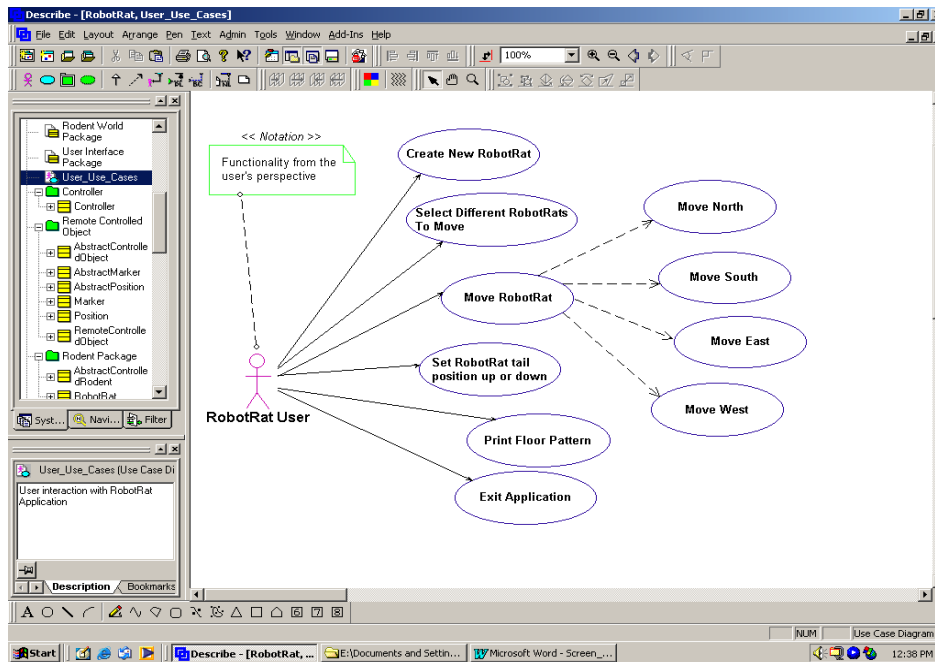


Figure 20-9: Completed Robot Rat User’s Perspective Use Case Diagram

It seems, from studying the complete user’s perspective use case diagram, that all user functionality is captured. A robot rat application user can create new robot rats, select which robot rat to move, move the selected robot rat, set the robot rat’s tail up or down, print the floor pattern, and exit the application. Notice how the Move RobotRat use case depends on four individual directional move use cases. The note in the upper-left corner of the diagram drives the point home that this diagram is expressing robot rat functionality from one particular actor’s perspective — the user.

PROGRAMMER PERSPECTIVE USE CASES

An important use case perspective to consider when designing an application and its supporting architecture is that of the programmer. Explicitly building programmer perspective use cases forces you to consider what features the application architecture must support to satisfy the project requirements. In real life, architectural requirements are often not explicitly stated or clearly understood by designers. However, when a client says to the contractor “I want to be able to add functionality to the application over time” what they are saying essentially is that you the contractor must build into the application the ability to accept new and possibly unforeseen features. The ability of the application to grow gracefully is a requirement in itself and the application architecture must be designed to do so.

The robot rat project specification makes its architectural requirements fairly clear. It says that the robot rat application must have the ability to add different types of remote controlled objects should the need arise. This requirement has a significant impact on the design of the inheritance hierarchy and class relationships.

Another requirement stated in the robot rat project specification is the use of the Model-View-Controller (MVC) pattern to separate the user interface from the rest of the program. Although patterns are not covered formally in this book, the use of the MVC pattern is necessary to achieve the stated requirements. Figure 20-10 shows the use case diagram for the application functionality from the programmer’s perspective.

As illustrated in figure 10-10 the robot rat application architecture must be able to accommodate new types of remote controlled objects, however, version 1.0 of the new and improved robot rat application will only allow the user to add more robot rats to the floor. The application architecture must also have a separate user interface class, a separate controller class, and a separate robot rat manager class.

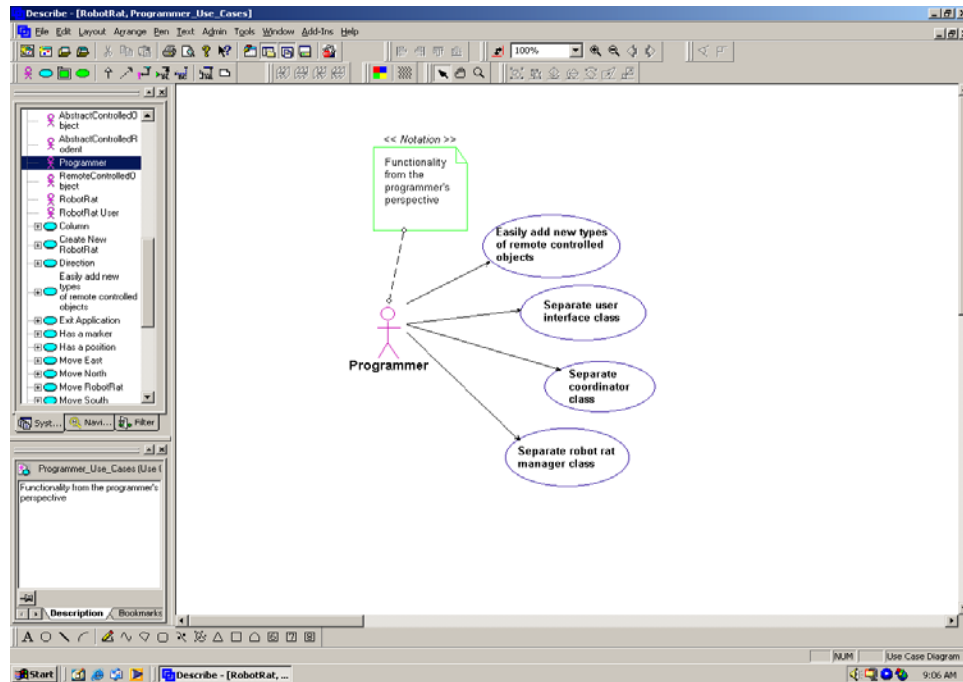


Figure 20-10: Programmer Perspective Use Cases

PAUSING TO CONSIDER DESIGN ISSUES

Now that the use cases for both the user and programmer perspectives are complete it would be nice if the modeling tool could interpret them and, on its own, offer one or more suitable architectures for you to consider. This is not likely to happen anytime soon so here we experience head-on the limitations of modeling tools. They help, they assist, they enable, but they do not do the design for you. How then to best proceed with the robot rat application design?

Here is where you do lots of thinking, thinking, and more thinking. In fact, if you observed most software engineers in the process of designing a system you would easily convince yourself they are doing nothing but just sitting in their offices daydreaming. Not true. A lot of the design process involves pure brain power, augmented by pencil and paper.

Taking our pencil and paper we could do an analysis of the robot rat project and identify key components that are in need of designing. Table 20-1 lists the results of a first attempt.

Design Artifact	Consideration
User Interface Class	The user interface class will handle all user input and output using iostreams. The user interface class will also be responsible for printing the floor pattern. **If the user interface prints the floor then it probably contains and manages the floor array, since the floor array for an iostream representation may be different than a GUI floor representation.

Table 20-1: Robot Rat Project Design Considerations

Design Artifact	Consideration
Robot Rat Manager Class	The robot rat manager class will keep track of and control all robot rat objects. Any command the user wants to execute from the user interface will be sent to the robot rat manager class. **The robot rat manager class must be able to add more rats to the floor. **Should it also be able to delete robot rat objects? **How will it manage multiple robot rat objects? With an array?
Controller Class	The controller class will handle all message passing between the user interface class and the robot rat manager class.
Remote Controlled Objects	A remote controlled object is any object that will ultimately be controlled by the user in its movement around an imaginary floor. Remote controlled objects must have some knowledge of the notion of their position upon the floor and the direction they are facing. Remote controlled objects will also be able to mark their position on the floor in some way. **Do remote controlled objects mark the floor directly or is the floor marked based on the position of a remote controlled object's marker? **Do remote controlled objects consist of a position and a marker? **Does a remote controlled object have knowledge of a floor? Or, rather, just knowledge of some notion of a boundary?
Robot Rat	A robot rat is an instance of a remote controlled object. A robot rat has a tail that can be set up or down.
Floor	The imaginary surface the remote controlled objects move upon. The "floor" the user sees, with pattern marking created by robot rats, may be represented by a two-dimensional array. **Where does the floor reside? In the user interface? In the robot rat manager class?
Position	A remote controlled object has a position upon the floor. The position consists of the current row, current column, and heading or direction (NORTH, SOUTH, EAST, or WEST).
Marker	A remote controlled object has a marker. The marker can be set to either the UP or DOWN position.

Table 20-1: Robot Rat Project Design Considerations

Table 20-1 provides enough information to begin a more detailed design, but one more use case diagram may prove helpful. I like to model application artifacts in a use case diagram. This use of a use case diagram may be considered by some as unorthodox, but I find any picture that helps you to better understand your design is worth the time it takes to draw.

Figure 20-11 shows a use case diagram with multiple actors. In this diagram each actor represents a potential design artifact and its associated attributes. The actors are arranged in an inheritance hierarchy complete with generalization arrows. As figure 20-11 illustrates, a RemoteControlledObject is a software entity that has a marker and a position attribute. The position attribute is further decomposed into direction, row, and column attributes.

The inheritance hierarchy shown in figure 20-11 is one of several possible alternatives. An AbstractControlledObject entity sits at the root of the hierarchy. It may very well be implemented as an abstract base class containing pure virtual functions existing only to publish an interface to the RemoteControlledObject class and further subclasses. The AbstractControlledRodent entity inherits its functionality from the RemoteControlledObject entity. The RobotRat entity then inherits from AbstractControlledRodent. This will enable the addition of different types of remote controlled objects. For instance, if you needed to add a different type of AbstractControlledRodent, such as a mouse, the you would simply extend the AbstractControlledRodent entity and name the new class RobotMouse. A RobotMouse may very well have different behavior than a RobotRat.

Entirely different types of remote controlled objects besides rodents could be added as well. For example, an AbstractControlledPerson entity could inherit from RemoteControlledObject. This would facilitate the addition of a

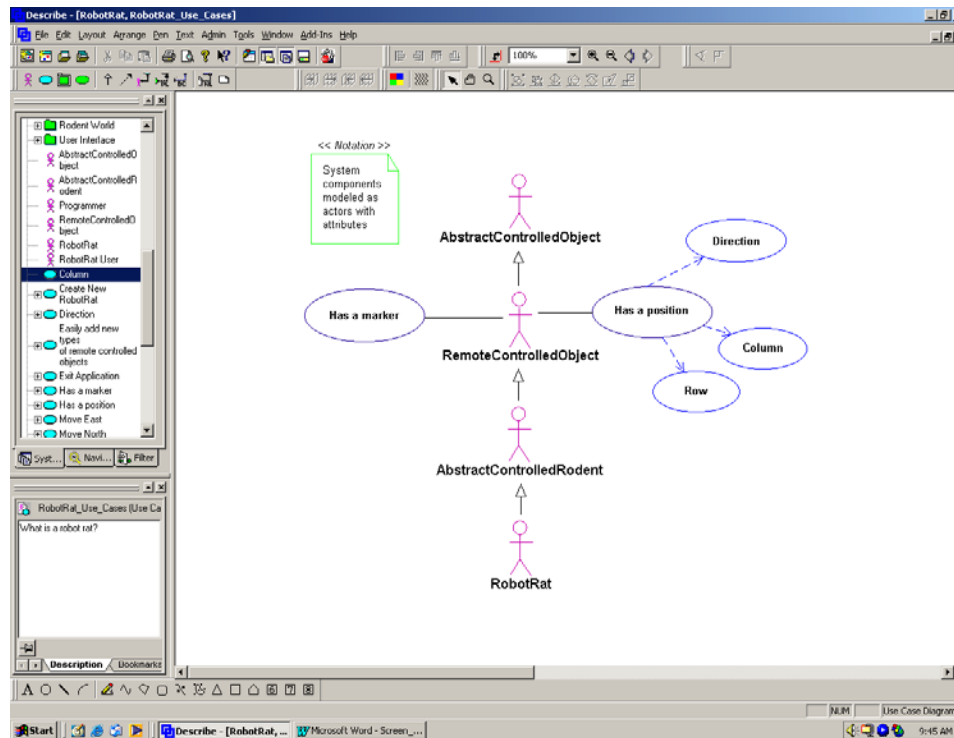


Figure 20-11: Partial Application Architecture Use Case Diagram

RobotMan or RobotWoman entity.

The inheritance hierarchy depicted in figure 20-11 seems to support the addition of new types of remote controlled objects as required by the robot rat project specification. This looks like a good place to begin a more detailed model design.

CREATING CLASS DIAGRAMS

Given the information provided in table 20-1 and the assurance gained from the analysis of the partial application architecture use case of figure 20-11 we can begin a more detailed design effort. A good place to begin this effort is by modeling the overall application package architecture. This will give you a guiding light or road map as you progress further with your design.

CREATING AN OVERALL PACKAGE ARCHITECTURE DIAGRAM

Figure 20-12 shows an overall robot rat application package architecture diagram. The package diagram is just a class diagram showing only the high-level package structure of the application. As you add classes later in the design process you can logically group them according to their package location.

You create a new class diagram in Describe™ the same way you create a use case diagram, except in the drop down box choose Class Diagram. You will find the class diagram artifacts in the tool bar. Click the Package icon to create each package. As you can see from figure 20-12 I have identified five packages for my solution to the robot rat project. They are listed in table 20-2.

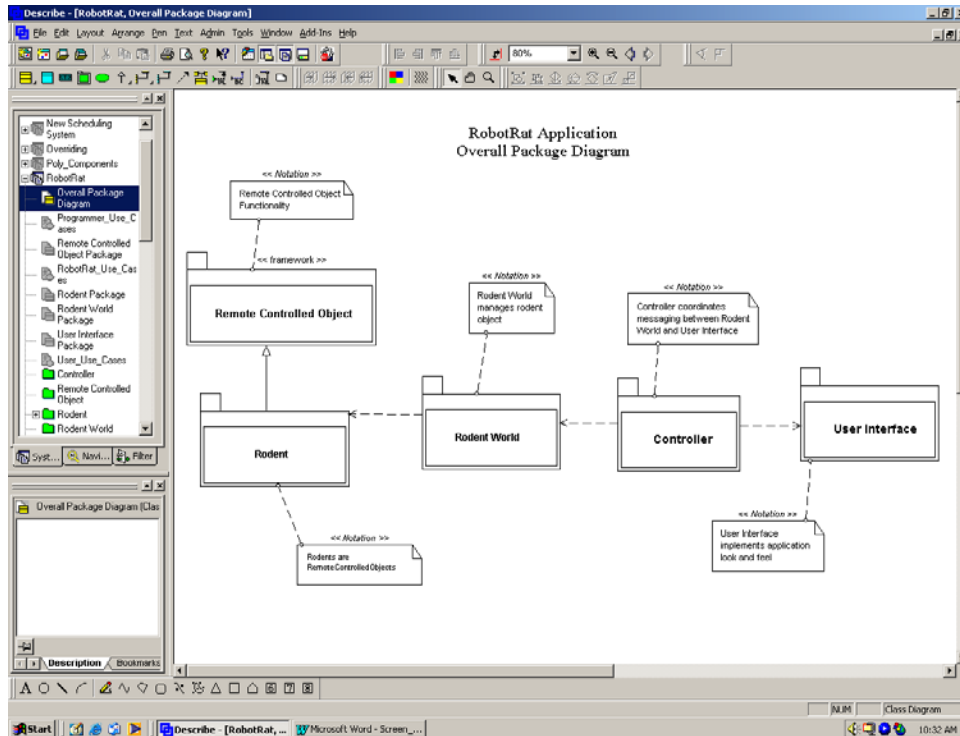


Figure 20-12: Overall Robot Rat Application Package Architecture

Package Name	Purpose
Remote Controlled Object	The remote controlled object package contains all classes necessary to implement a remote controlled object.
Rodent	The rodent package contains all classes necessary to implement remote controlled rodents like RobotRat.
Rodent World	The rodent world package contains the class or classes necessary to implement the robot rat manager class. Rodent world seemed like a good name for the robot rat manager class.
Controller	The controller package contains all the classes necessary to implement the controller object.
User Interface	The user interface package contains all the classes necessary to implement the user interface.

Table 20-2: Robot Rat Application Package Names and Their Purpose

MOVING BEYOND THE PACKAGE DIAGRAM

Using the overall package diagram as a guide you can now begin to define the classes that belong to each package. I will start with the Remote Controlled Object package and work on its associated classes. You can create a new class diagram for each package or create an overall class diagram and annotate the diagram in some way to denote what package each class or set of classes belong to. In this example I choose the overall diagram approach. Figure 20-13 shows the beginning of the overall class diagram. At this stage of the detailed design you can postpone the consideration of individual class functions and attributes until you have fleshed out the classes and their relationships.

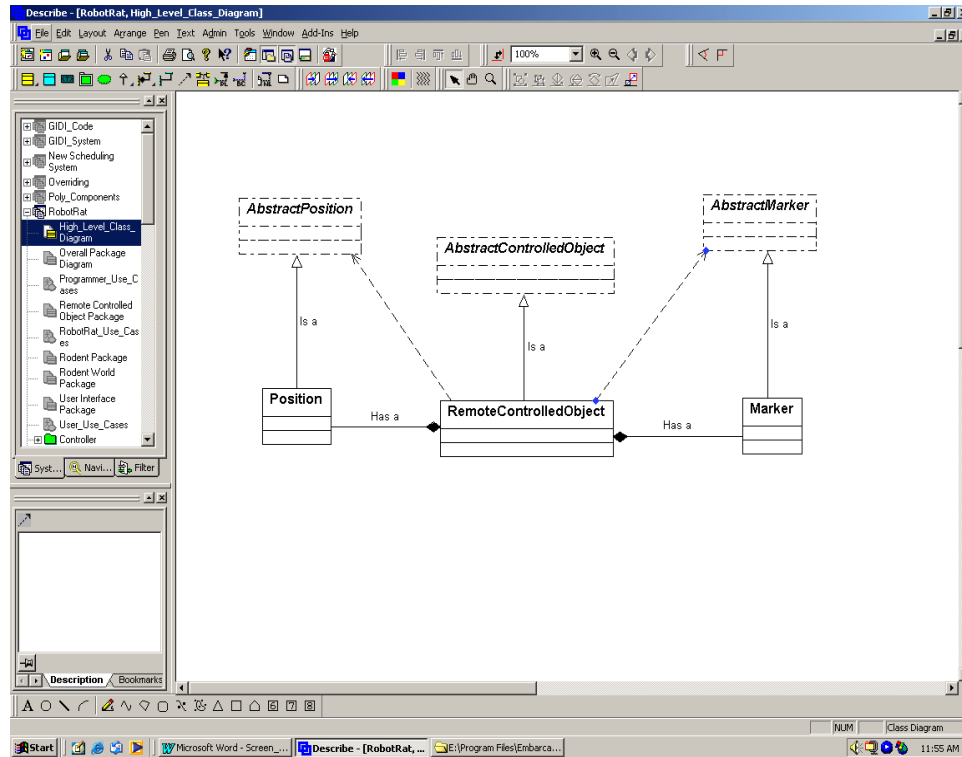


Figure 20-13: Class Diagram Showing Remote Controlled Object Package Classes

The class diagram shown in figure 20-13 implements the partial architectural use case diagram shown previously in figure 20-11 with the addition of two abstract base classes: `AbstractPosition` and `AbstractMarker`. The `RemoteControlledObject` class is an aggregate containing a `Position` object and a `Marker` object.

Adding OPERATIONS AND ATTRIBUTES TO CLASSES

When you have a set of classes and their static relationship captured in a diagram you can begin to add methods and attributes to each class. Figure 20-14 shows the Properties Editor window for the `AbstractPosition` class.

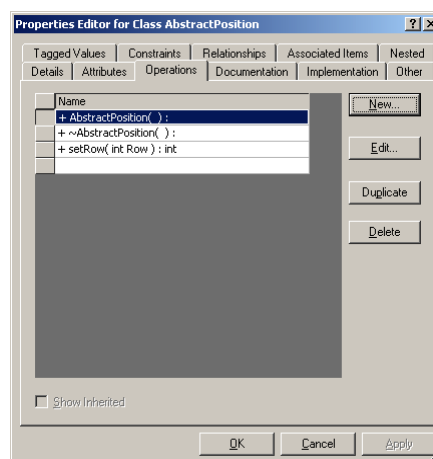


Figure 20-14: Properties Editor for AbstractPosition Class

To add operations to the class click on the `Operations` tab, then click on the `New` button. This will bring up another Properties Editor window. Figure 20-15 shows the Properties Editor window for the `AbstractPosition` constructor.

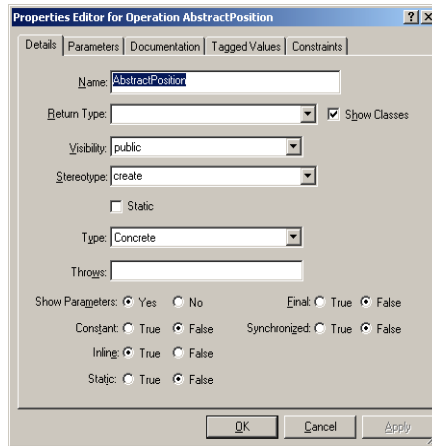


Figure 20-15: Properties Editor Window for the AbstractPosition Operation

Describe™ will automatically generate code for class constructors and destructors if you make the necessary setting prior to code generation, however, the default generated constructors and destructors may not be satisfactory for your purposes.

Figure 20-15 shows the AbstractPosition operation, which is a constructor, being created. The Properties Editor window is used to set the name of the operation, its return type if needed, its visibility, stereotype, etc. In my approach to the robot rat design I wanted to create an abstract base class named AbstractPosition which has an inline constructor and destructor and its remaining functions pure virtual. Notice the settings used here to create the proper constructor. Compare these settings with those used in figure 20-16 below to create a pure virtual function named setRow.

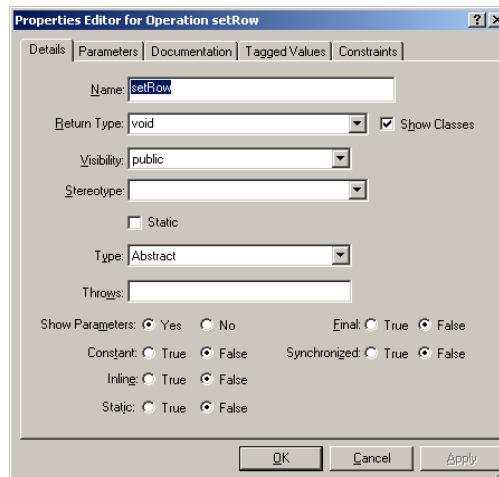


Figure 20-16: Properties Editor Window for the setRow() Function

Notice the return type is set to void, the visibility is public, and the type is set to Abstract. The setRow() function will have one integer parameter. To define function parameters click the Parameters tab in the Properties Editor window. This will reveal the Parameters section of the Properties Editor window as shown in figure 20-17. To add a parameter simply type in the parameter type followed by the parameter name. When you have finished adding operations, parameters, or editing other operation attributes click the OK button.

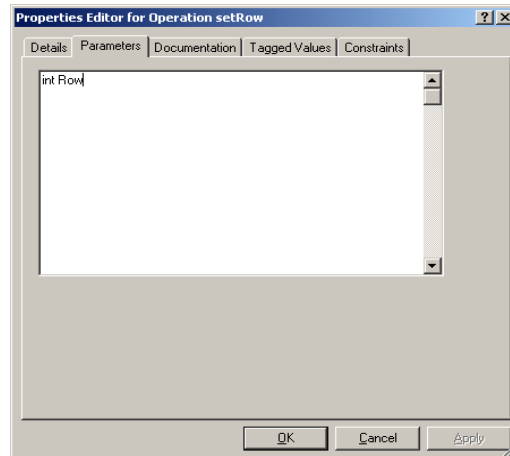


Figure 20-17: Adding Operation Parameters Using the Properties Editor Window

ITERATING THROUGH THE DESIGN PROCESS

At this point in the design process you will begin to iterate through the process of adding classes to design diagrams and defining any necessary functions and attributes. You might very well start generating code for the classes you have created thus far to test the quality of the automatically generated code and see how things are going to turn out. Following my earlier advice of “design a little, program a little, test a lot”, I recommend this approach and advise you not to wait until you have designed everything up front to test the quality of the generated code.

In this section I will provide you with a table that lists all the classes I created in my version of the robot rat project, their functions, and their attributes. I will also show you the final overall class diagram. I will postpone the formal discussion of code generation until after the section on sequence diagrams.

Table 20-3 lists the classes used in the robot rat project along with their package affiliation, attributes, and functions.

Class Name (Type)	Package	Attributes	Functions
AbstractControlledObject (Abstract) [Header File Only]	Remote Controlled Object	None	<pre> +AbstractControlledObject(){} +virtual ~AbstractControlledObject(){} +virtual void move(int spaces)=0 +virtual void turnLeft()=0 +virtual void turnRight()=0 +virtual void setBoundaries(int rows, int columns)=0 +virtual void setRow(int row)=0 +virtual void setColumn(int column)=0 +virtual void setPosition(int row, int column)=0 +virtual int getRow()=0 +virtual int getColumn()=0 +virtual int getUpperRowBoundary()=0 +virtual int getUpperColumnBoundary()=0 +virtual void setMarkerUp()=0 +virtual void setMarkerDown()=0 +virtual void toggleMarker()=0 +virtual bool isMarkerDown()=0 +virtual char* getObjectType()=0 </pre>

Table 20-3: Robot Rat Project Classes

Class Name (Type)	Package	Attributes	Functions
AbstractPosition (Abstract) [Header File Only]	Remote Controlled Object	None	+AbstractPosition(){ +virtual ~AbstractPosition(){ +virtual void move(int spaces)=0 +virtual void turnLeft()=0 +virtual void turnRight()=0 +virtual void setBoundaries(int rows, int columns)=0 +virtual void setRow(int row)=0 +virtual void setColumn(int column)=0 +virtual void setPosition(int row, int column)=0 +virtual int getRow()=0 +virtual int getColumn()=0 +virtual int getUpperRowBoundary()=0 +virtual int getUpperColumnBoundary()=0
AbstractMarker (Abstract) [Header File Only]	Remote Controlled Object	None	+AbstractMarker(){ +virtual ~AbstractMarker(){ +virtual void setMarkerUp()=0 +virtual void setMarkerDown()=0 +virtual void toggleMarker()=0 +virtual bool isMarkerDown()=0
RemoteControlledObject (Abstract) [It does not implement the char* getObjectType() function]	Remote Controlled Object	-static int count -AbstractPosition* itsPosition -AbstractMarker* itsMarker	+RemoteControlledObject() +virtual ~RemoteControlledObject() +void move(int spaces) +void turnLeft() +void turnRight() +void setBoundaries(int rows, int columns) +void setRow(int row) +void setColumn(int column) +void setPosition(int row, int column) +int getRow() +int getColumn() +int getUpperRowBoundary() +int getUpperColumnBoundary() +void setMarkerUp() +void setMarkerDown() +void toggleMarker() +bool isMarkerDown()
Position (Concrete)	Remote Controlled Object	-int its_row -int its_column -int upper_row_boundary -int upper_column_boundary -int lower_row_boundary -int lower_column_boundary -enum Direction {NORTH, SOUTH, EAST, WEST} -Direction its_direction	+Position(int row = 0, int column = 0) +~Position() + void move(int spaces)=0 + void turnLeft()=0 +void turnRight()=0 +void setBoundaries(int rows, int columns)=0 +void setRow(int row)=0 +void setColumn(int column)=0 +void setPosition(int row, int column)=0 +int getRow()=0 +int getColumn()=0 +int getUpperRowBoundary()=0 +int getUpperColumnBoundary()=0 -void incrementRow(int val) -void incrementColumn(int val) -void setLowerRowBoundary(int lower) -void setUpperRowBoundary(int upper) -void setLowerColumnBoundary(int lower) -void setUpperColumnBoundary(int upper)
Marker (Concrete)	Remote Controlled Object	-enum MarkerPosition {UP, DOWN} -MarkerPosition its_marker_position	+Marker() +virtual ~Marker() +void setmarkerUp() +void setMarkerDown() +void toggleMarker() +bool isMarkerDown()
AbstractControlledRodent (Abstract) [Header File Only]	Rodent	None	+AbstractControlledRodent(){ +virtual ~AbstractControlledRodent(){ +virtual void setTailUp() = 0 +virtual void setTailDown() = 0;

Table 20-3: Robot Rat Project Classes

Class Name (Type)	Package	Attributes	Functions
RobotRat (Concrete)	Rodent	None	+RobotRat(int row_boundary = 20, int column_boundary = 20) +virtual ~RobotRat() +RobotRat(RobotRat& rhs) +RobotRat& operator=(RobotRat& rhs) +void setTailUp() +void setTailDown() +char* getObjectType()
RodentWorld (Concrete)	Rodent World	- AbstractControlledRodent** rodent_array -int its_rows -int its_columns -int number_or_rodents -int rodent_number	+RodentWorld(int rows = 20, int columns = 20) +virtual ~RodentWorld() +void addRodent() +void deleteRodent() +void toggleRodent(); +void turnRodentLeft() +void turnRodentRight() +void moveRodent(int spaces) +void setRodentTailUp() +void setRodentTailDown() +char* getRodentType() +int getRodentRow() +int getRodentColumn() +bool isRodentTailDown()
Controller (Concrete)	Controller	-UserInterface* its_ui -RodentWorld* its_world -enum MenuActions {ADD = 1, TOGGLE, TURNRIGHT, TURNLEFT, TAILUP, TAILDOWN, MOVE, DISPLAYFLOOR, EXIT} -void move()	+Controller() +~Controller() +void run()
UserInterface (Concrete)	User Interface	-bool** floor -int its_rows -int its_columns -int spaces	+UserInterface() +~UserInterface() +void displayMenu() +int getMenuChoice() +void markFloor(int row, int column) +void clearFloor(int row, int column) +displayFloor() +int getRows() +int getColumns() +int getSpaces()

Table 20-3: Robot Rat Project Classes

The completed overall class diagram is given in figure 20-18. The overall class diagram gives a high-level view of the robot rat project classes and their static relationships. At this point in the design process you would continue to iterate through the steps of selecting a class and adding attributes and functions.

A suggested approach is to concentrate on the classes in a particular package, getting all their attributes and functions defined. For example, a good place to start would be the classes in the Remote Controlled Object package. What you will notice as you iterate through your design is that the design will be unstable. By this I mean you will find yourself making changes to almost all the classes in the RemoteControlledObject package as you begin to better understand the needs and issues surrounding the design. Since subclasses in an inheritance hierarchy depend on their base classes, you will find that changes to base classes have a definite effect on dependent subclasses. For example, if you add a pure virtual method to a base class you must then implement that function in a subclass somewhere down the line. However, as the design progresses, the abstractions at the upper levels of the design will begin to stabilize. Soon enough you will reach a point where the necessity to modify the high-level abstractions drops way off. The design begins to set like Jello™ and adding functionality becomes a simple matter of extending the design. (see chapter 19 for a discussion of the open-closed principle)

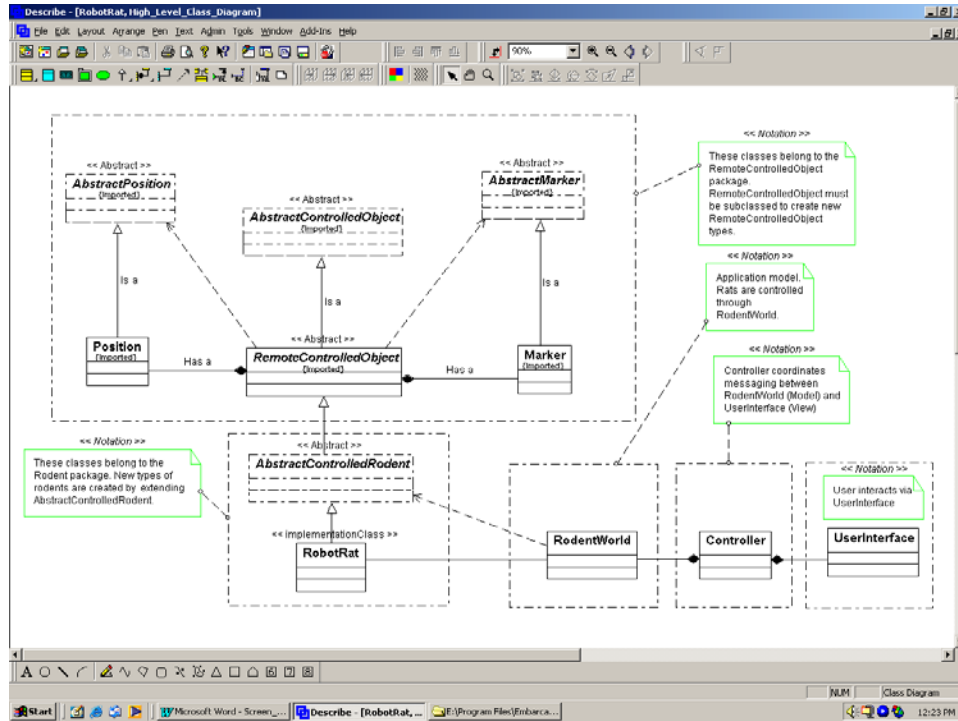


Figure 20-18: Completed and Annotated Overall Class Diagram

CREATING SEQUENCE DIAGRAMS

Sequence diagrams are used to present views of dynamic message flow between system objects. You do not have to have all your classes and operations defined to create sequence diagrams. In fact, Describe™ will let you define new classes, and add new operations to existing classes, when you create sequence diagrams.

PROPER USE OF SEQUENCE DIAGRAMS

Sequence diagrams are most effective when they focus on one particular thread of application execution or functionality. A sequence diagram that attempts to model all the messaging in a large application would be very hard to read.

ADDING OBJECTS TO SEQUENCE DIAGRAMS

Figure 20-19 shows the start of a sequence diagram for robot rat application launch. Objects are added to the sequence diagram by dragging them from the Systems Hierarchy pane, the large pane open to the left of the sequence diagram. Objects can also be added to the sequence diagram by selecting the desired object in the tool bar and adding it to the diagram.

If you have identified some or all of the classes for the system you are designing, they will be available in the System Hierarchy pane. If you add a new class to the sequence diagram, (a class that does not currently exist in your system), Describe™ will prompt you to add the new class to the current system if you desire. This is a good way to add classes to your design that you may not have thought of adding during the initial design phase.

In figure 20-19, the main application object, just to the right of the RobotRat User actor, represents the main.cpp file. I modeled the main.cpp file as an object in the system but chose not to add the main.cpp object to the system.

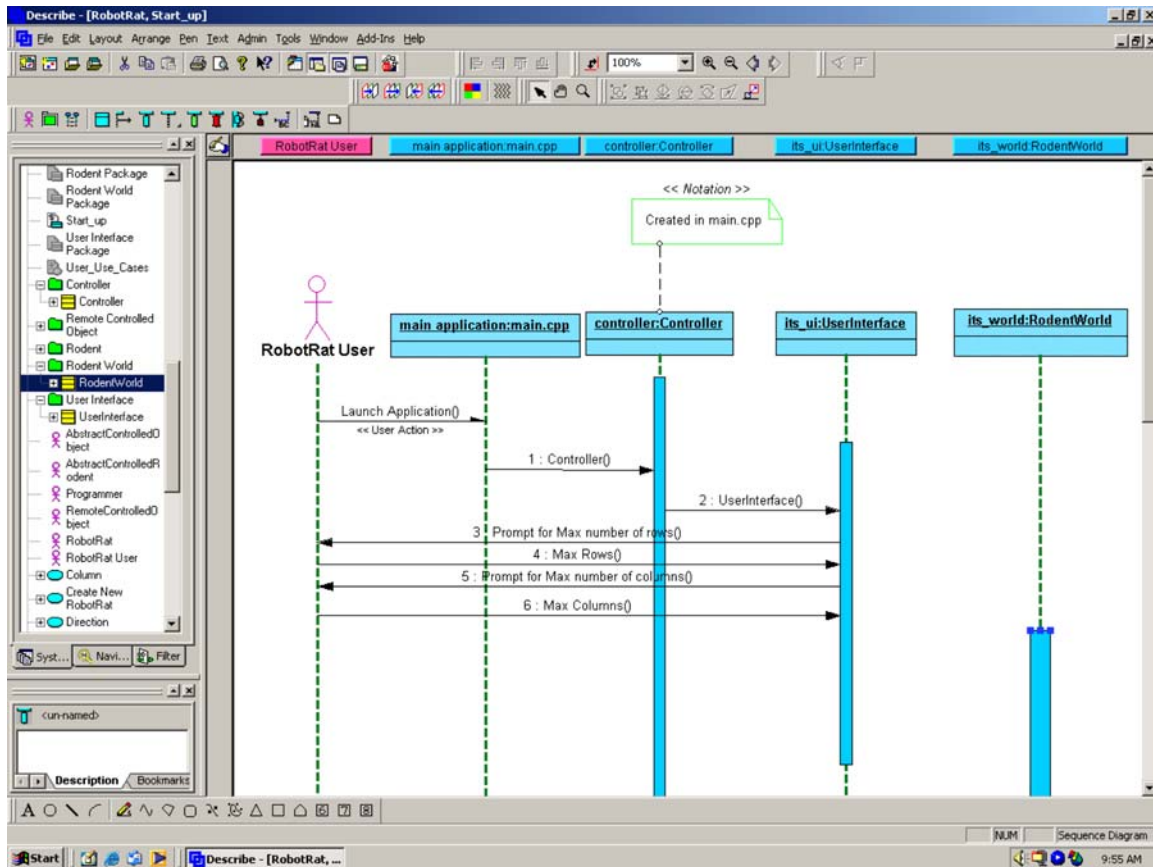


Figure 20-19: Start of Sequence Diagram for Robot Rat Application Launch

Adding MESSAGES TO SEQUENCE DIAGRAMS

Sequence diagrams model both outside actor interaction with an application and the interaction between application objects. Figure 20-19 shows the RobotRat User actor sending a Launch Application message to the main.cpp object. When the application is launched the main() function creates a new Controller object. This is indicated by the message labeled 1: Controller(). The number represents the message sequence; the name represents the object function name or operation.

To add a message to a sequence diagram click the message icon in the tool bar, click on the sequence diagram to indicate the message source, and finish by clicking on the diagram to indicate the message target. This action will result in a blank message line being generated from one object line to another. To add a name and other attributes to a message you can edit the message properties. Figure 20-20 shows the Properties Editor for the Controller() message sent from the main.cpp object to the controller object.

If you try and add a message to a sequence diagram and the target object does not yet declare the function, Describe™ will prompt you to add the operation to the target object. This is an excellent way to incrementally build your class interfaces.

Figure 20-21 shows the completed sequence diagram for the robot application start-up sequence. Figure 20-21 shows the sequence of events from the time a user starts the robot rat application through to menu display and menu handling. This is an example of one particular thread of application execution. Other logical choices for sequence diagrams would be one for each of the robot rat user use cases shown in figure 20-9. For example, figure 20-22 shows the completed sequence diagram for the Create New RobotRat use case. This sequence diagram, along with all supporting class diagrams, can be linked to the Create New RobotRat use case. See the Linking Diagrams section to learn how to link different diagram elements to use case diagrams.

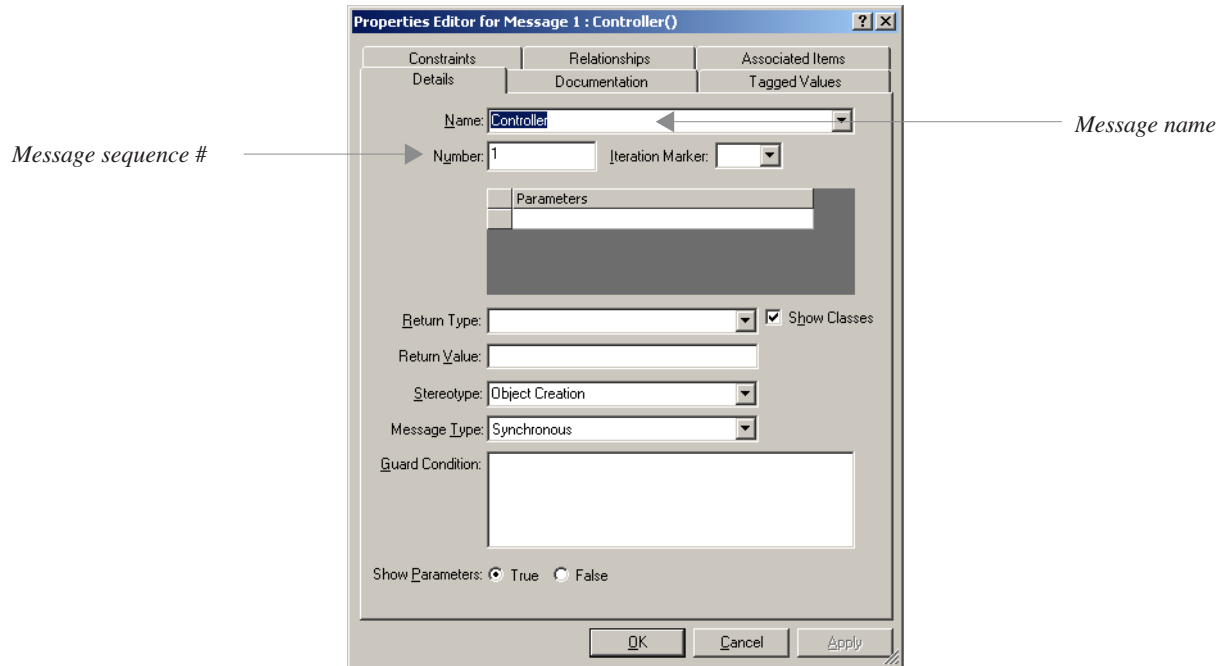


Figure 20-20: Editing Controller() Message Properties

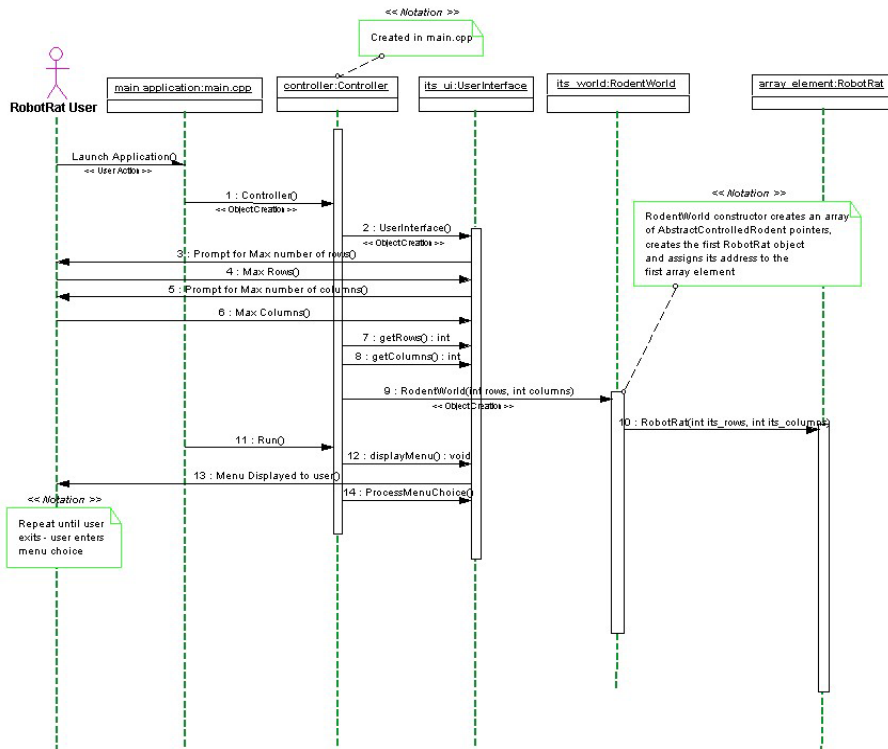


Figure 20-21: Completed Robot Rat Application Launch Sequence

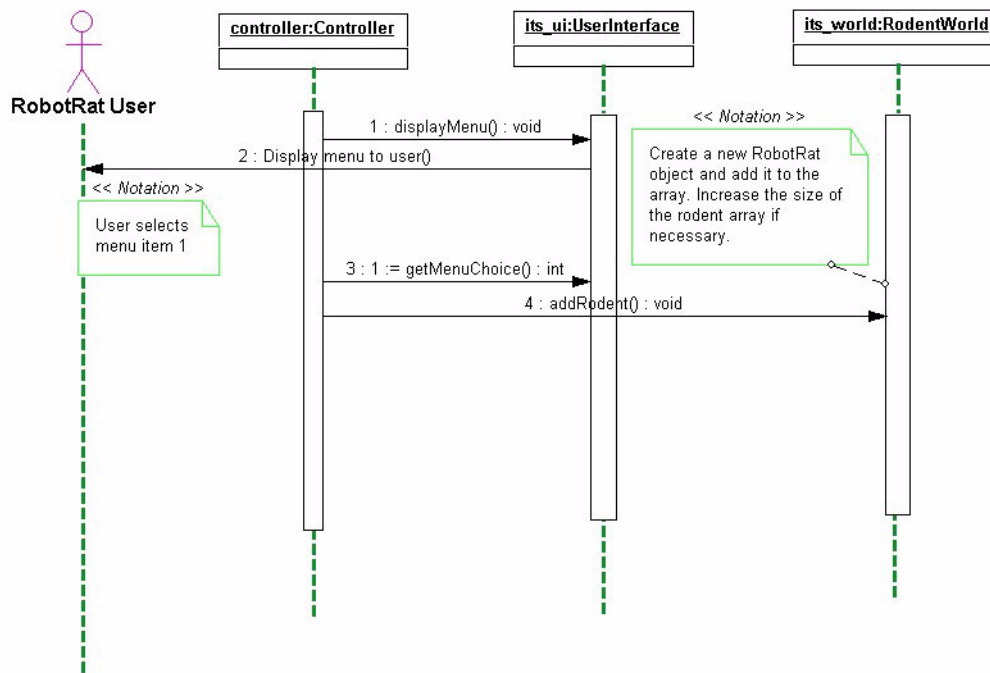


Figure 20-22: Create New RobotRat Sequence Diagram

GENERATING SOURCE CODE

Describe™ lets you generate C++ source code from class diagrams. I recommend generating code early in the design process to ensure your class properties are properly set. Once you get the settings just right you can generate your code framework with confidence.

Start the code generation process by opening a class diagram and selecting the classes you want to generate code for. Figure 20-23 shows a detailed robot rat application class diagram opened with all the classes belonging to the Remote Controlled Object package selected. When you have selected the classes go to the Tools menu and select Generate Code as is shown in figure 20-24. You will be presented with the Code Generation dialog shown in figure 20-25.

From the Code Generation dialog you can select the target language, which classes to generate code for, and Script Options. Take note of the Script Options portion of the Code Generation dialog. Script Options lets you check or uncheck different code generation options. Some of the options include the automatic insertion of a copyright comment, automatic generation of constructor and destructor functions, and automatic generation of get and set methods for class attributes. Getting code generation just right for a complex C++ application takes some skill and a little trial-and-error. Once you get the settings of your classes, their related attributes and functions, and the code generation options just right, Describe™ does a pretty good job of generating the application framework code. You, of course, must add the guts to the operations to get everything running properly.

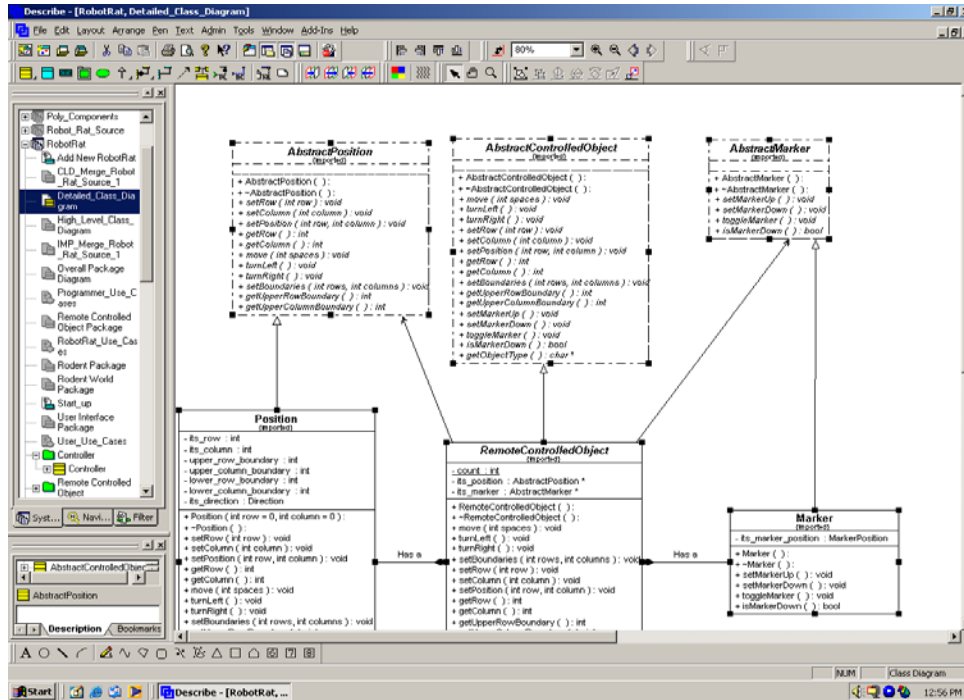


Figure 20-23: First Step to Generating Code: Select Class Diagrams

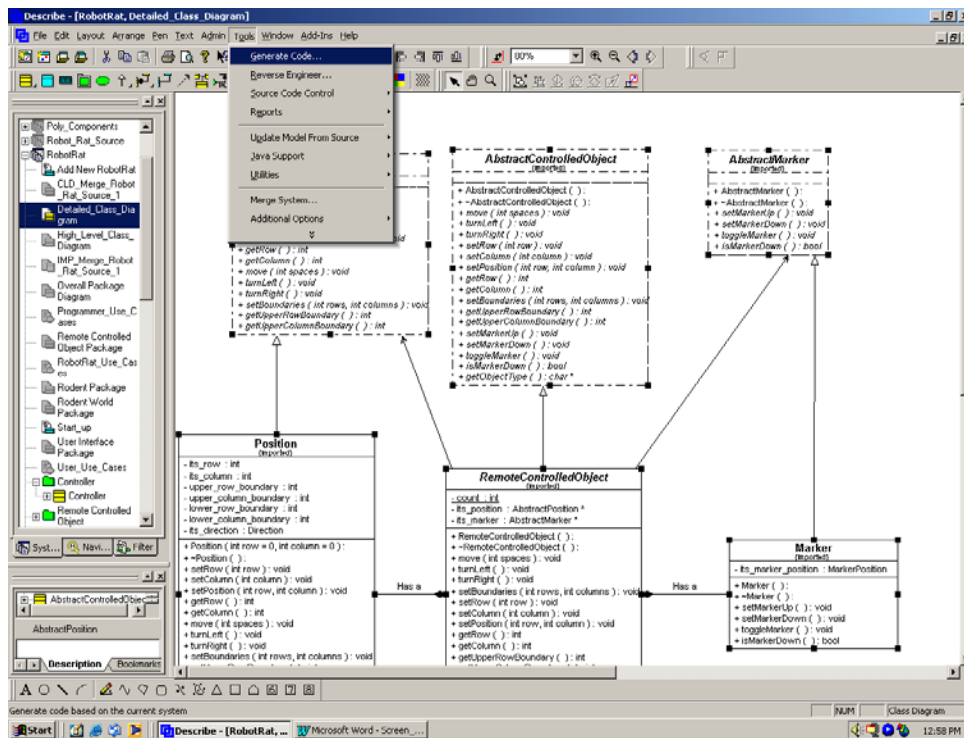


Figure 20-24: Generate Code Menu Item

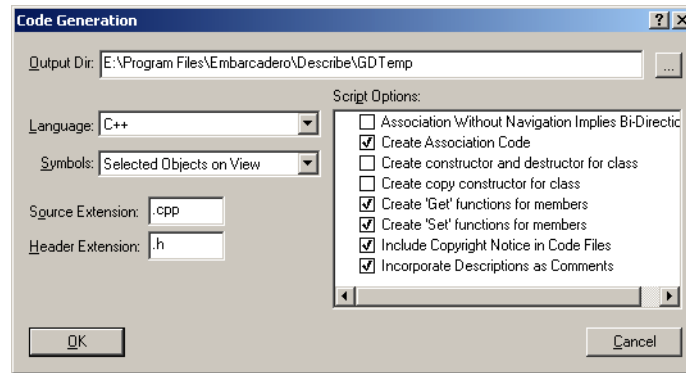


Figure 20-25: Code Generation Dialog

REVERSE ENGINEERING

Reverse engineering is a Describe™ feature that enables you to create class diagrams from existing source code. Reverse engineering can be used to start the design documentation process for an existing system. After all, a class diagram is worth a thousand raw source files.

More importantly, however, the reverse engineering feature can be used to update existing Describe™ systems when changes to source code have been made. In this section I will show you how to reverse engineer a set of source files and use the newly reverse engineered system to update an existing system.

Here is the scenario: Assume you have created all the robot rat classes necessary to implement the robot rat application. From the class diagram you generate your application framework code and then edit the framework code directly to complete the robot rat application. During implementation you may find it necessary to add class operations and attributes not captured in the initial design phase. At anytime during the implementation phase you can update the robot rat design by reverse engineering the modified source files into a new system, and then merging the new system into the robot rat system. To start the reverse engineering process from the Tools menu select Reverse Engineer. This

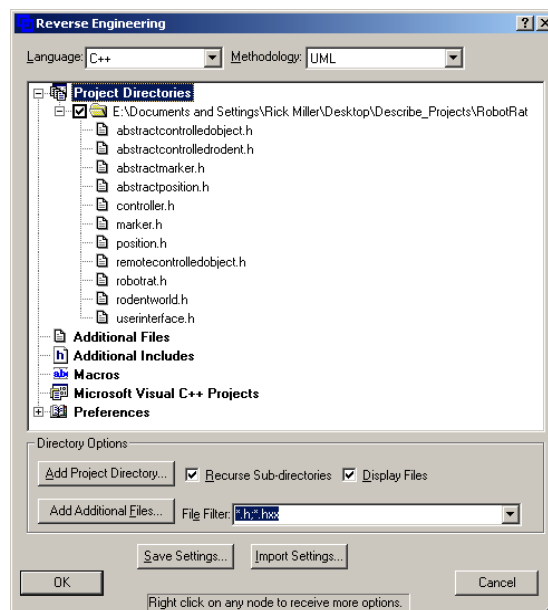


Figure 20-26: Reverse Engineering Dialog

results in the Reverse Engineering dialog shown in figure 20-26. In the Reverse Engineering dialog you select the language, methodology, and add any source files you want to reverse engineer. When you have added all the necessary files click the OK button. You will then be asked to name the new system and set various other system properties as shown in figure 20-27.

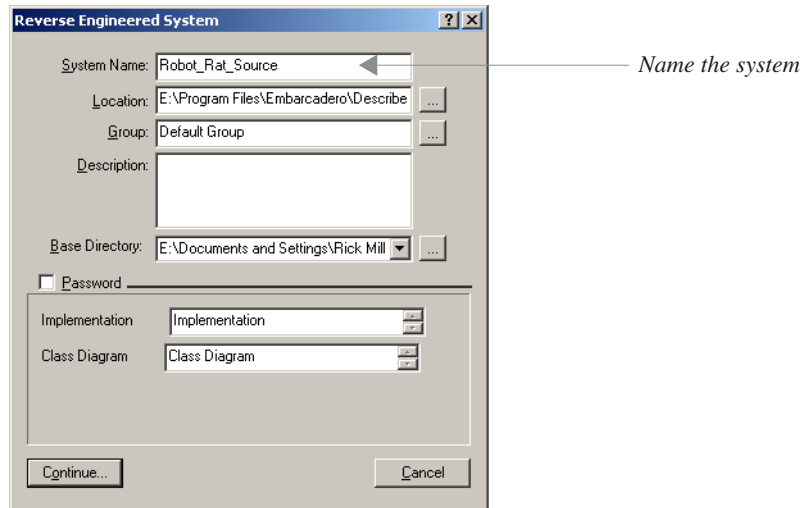


Figure 20-27: Step 2 in the Reverse Engineering Process: Naming the New System and Setting Various System Properties

When you have named the new system and set the other system properties according to your needs, click the continue button. Describe™ will start to reverse engineer your source files. Progress is shown in the REProgress window shown in figure 20-28.

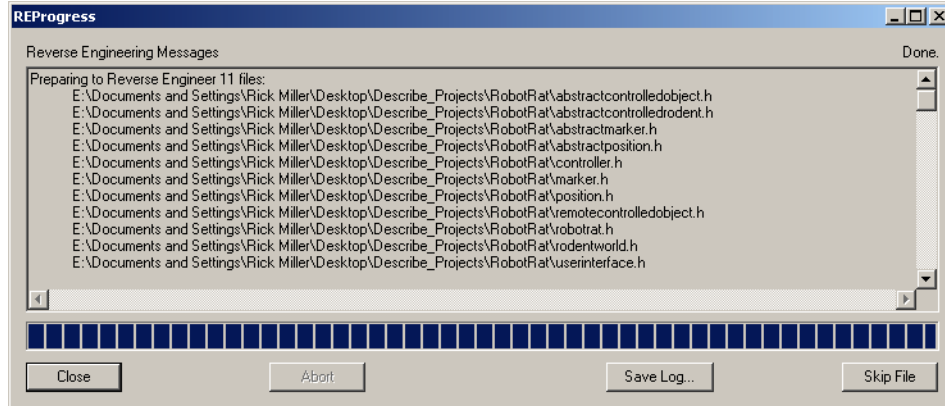


Figure 20-28: REProgress Window

MERGING SYSTEMS

When you have finished reverse engineering your updated source code you can merge the newly created system into the existing robot rat system. To merge systems, both systems must be open. You want to merge the newly reverse engineered system into the existing system, so, select a class diagram in the robot rat system and from the Tools menu select Merge System. This will result in the Merge System dialog shown in figure 20-29. Select the system you want to merge into the existing system and click the OK button. As the system merge takes place you will see a dialog giving you a report of each new system object being added to the existing system.

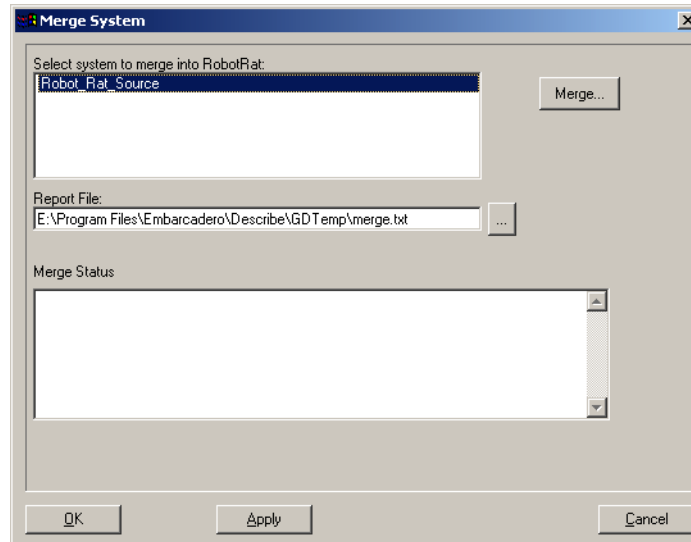


Figure 20-29: Merge System Dialog

LINKING DIAGRAM OBJECTS TO DIAGRAMS

Describe™ allows you to associate diagram objects with other diagrams within a system. This is an excellent way to create a comprehensive documentation chain. You have already seen how to link diagram objects with external documentation. In this section I will show you how to link a use case with a sequence diagram.

To associate a diagram object with another diagram, right click on the diagram object, click Associated Diagrams, and choose from the list of system diagrams to complete the association. Figure 20-30 shows the Create New RobotRat use case being associated with the Create New RobotRat sequence diagram.

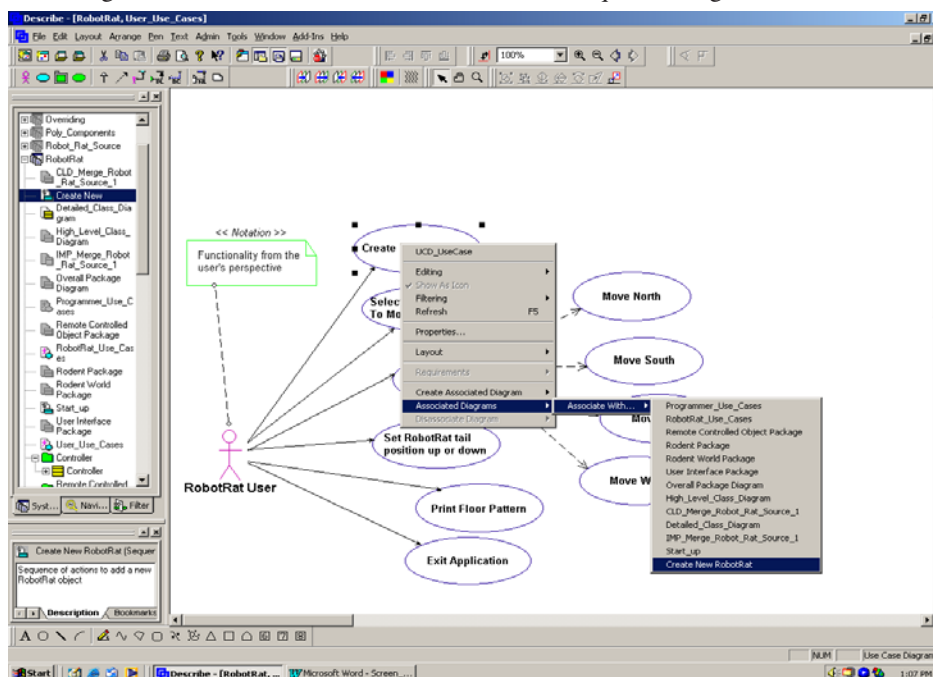


Figure 20-30: Associating Diagram Object with System Diagrams

When a diagram object has been linked to a system diagram, the linked diagram can be viewed by double clicking the diagram object. For example, to view the diagram or diagrams linked to the Create New RobotRat use case, double click the Create New RobotRat use case. This will display a dialog as shown in figure 20-31.

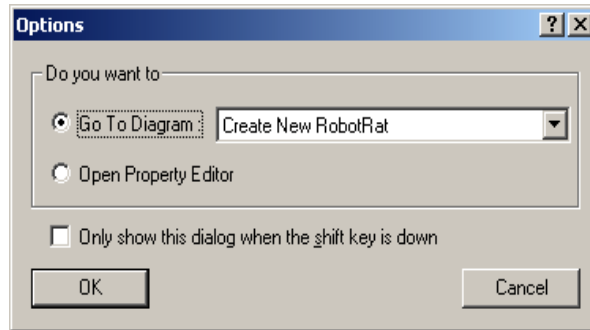


Figure 20-31: Navigating to Linked Diagram

GENERATING WEB PROJECT REPORTS

Describe™ provides an excellent reporting mechanism that enables you to share your system designs with others via the web. All system documentation captured during the design process is processed into a comprehensive web report. To generate a web report select Tools->Reports->Web Viewer. This will bring up the web viewer wizard as shown in figure 20-32.

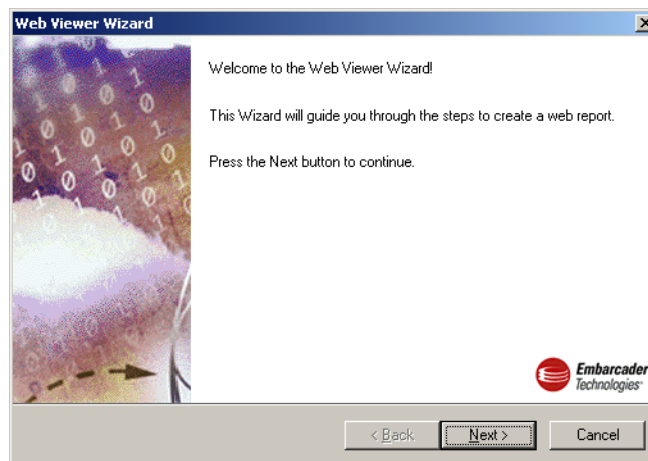


Figure 20-32: Web Viewer Wizard

Click the Next button and go to the next wizard window as shown in figure 20-33. Select the system on which to generate a web report and click the Next button. The last two wizard windows allow you to select the desired output directory and confirm your web viewer settings.

When the web viewer wizard completes it will automatically launch Microsoft Internet Explorer™ and bring up the main web report viewer window for the RobotRat project as shown in figure 20-34. The main web view screen contains a symbology key. This helps non-technical folks or those not familiar with UML to understand your diagrams.

In addition to generating great web reports, the Describe™ web viewer report function provides an excellent way to rapidly create JPEG images of all your system diagrams for inclusion into other system documentation and presentations.

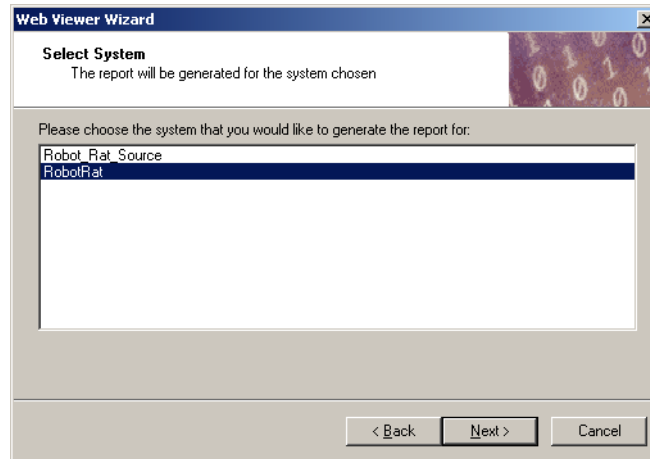


Figure 20-33: Selecting System for Web Report Generation

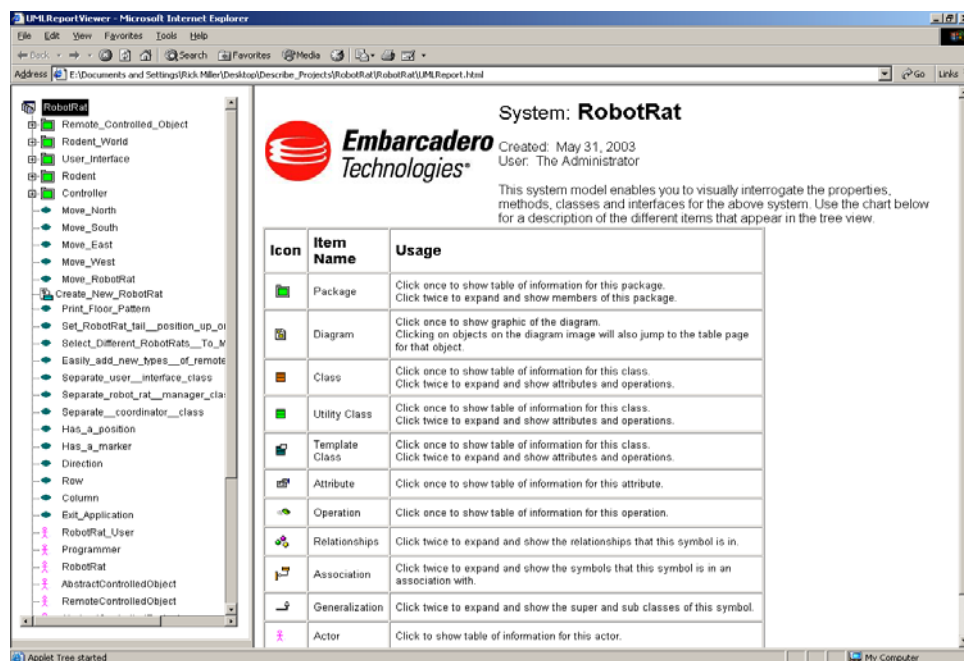


Figure 20-34: Main Screen - RobotRat Project Web View

SUMMARY

The purpose of a UML modeling tool is to assist the object-oriented analyst, designer, or programmer in the analysis, design and implementation of their object-oriented software system. A modeling tool also plays a critical role in communicating software designs to other analysts, designers, programmers, managers, and customers. A good UML modeling tool is therefore both a design and communication tool at the same time.

Regardless of how good the UML modeling tool is, it will not design or program your application by itself. You need to supply the brainpower. However, a good modeling tool can play a critical role in helping you craft a robust design.

ROBOTRAT SOURCE CODE

This section presents the complete source code listing for the robot rat project discussed in this chapter.

ABSTRACTPOSITION.H

abstractposition.h

```
#ifndef ABSTRACT_POSITION_H
#define ABSTRACT_POSITION_H

class AbstractPosition {
public:
    AbstractPosition();
    virtual ~AbstractPosition();
    virtual void setRow(int row) = 0;
    virtual void setColumn(int column) = 0;
    virtual void setPosition(int row, int column) = 0;
    virtual int getRow() = 0;
    virtual int getColumn() = 0;
    virtual void move(int spaces) = 0;
    virtual void turnLeft() = 0;
    virtual void turnRight() = 0;
    virtual void setBoundaries(int rows, int columns) = 0;
    virtual int getUpperRowBoundary() = 0;
    virtual int getUpperColumnBoundary() = 0;
};
#endif
```

ABSTRACTMARKER.H

abstractmarker.h

```
#ifndef ABSTRACT_MARKER_H
#define ABSTRACT_MARKER_H

class AbstractMarker {
public:
    AbstractMarker();
    virtual ~AbstractMarker();
    virtual void setMarkerUp() = 0;
    virtual void setMarkerDown() = 0;
    virtual void toggleMarker() = 0;
    virtual bool isMarkerDown() = 0;
};
#endif
```

ABSTRACTCONTROLLEDOBJECT.H

abstractcontrolledobject.h

```
#ifndef ABSTRACT_CONTROLLED_OBJECT_H
#define ABSTRACT_CONTROLLED_OBJECT_H

class AbstractControlledObject {
public:
    AbstractControlledObject();
    virtual ~AbstractControlledObject();
    virtual void move(int spaces) = 0;
    virtual void turnLeft() = 0;
    virtual void turnRight() = 0;
    virtual void setRow(int row) = 0;
    virtual void setColumn(int column) = 0;
    virtual void setPosition(int row, int column) = 0;
    virtual int getRow() = 0;
    virtual int getColumn() = 0;
    virtual void setBoundaries(int rows, int columns) = 0;
    virtual int getUpperRowBoundary() = 0;
    virtual int getUpperColumnBoundary() = 0;
    virtual void setMarkerUp() = 0;
    virtual void setMarkerDown() = 0;
    virtual void toggleMarker() = 0;
};
```

```

    virtual bool isMarkerDown() = 0;
    virtual char* getObjectType() = 0;

};
#endif

```

position.h

position.h

```

#ifndef POSITION_H
#define POSITION_H
#include "abstractposition.h"

class Position : public AbstractPosition {

public:
    Position(int row = 0, int column = 0);
    ~Position();
    void setRow(int row);
    void setColumn(int column);
    void setPosition(int row, int column);
    int getRow();
    int getColumn();
    void move(int spaces);
    void turnLeft();
    void turnRight();
    void setBoundaries(int rows, int columns);
    int getUpperRowBoundary();
    int getUpperColumnBoundary();

private:
    int its_row;
    int its_column;
    int upper_row_boundary;
    int upper_column_boundary;
    int lower_row_boundary;
    int lower_column_boundary;
    enum Direction {NORTH, SOUTH, EAST, WEST};
    Direction its_direction;
    void incrementRow(int val);
    void incrementColumn(int val);
    void setLowerRowBoundary(int lower);
    void setUpperRowBoundary(int upper);
    void setLowerColumnBoundary(int lower);
    void setUpperColumnBoundary(int upper);
};
#endif

```

marker.h

marker.h

```

#ifndef MARKER_H
#define MARKER_H
#include "abstractmarker.h"

class Marker : public AbstractMarker {
public:
    Marker();
    virtual ~Marker();
    void setMarkerUp();
    void setMarkerDown();
    void toggleMarker();
    bool isMarkerDown();

private:
    enum MarkerPosition {UP, DOWN};
    MarkerPosition its_marker_position;
};
#endif

```

REMOTECONTROLLEDOBJECT.H*remotecontrolledobject.h*

```

#ifndef REMOTE_CONTROLLED_OBJECT_H
#define REMOTE_CONTROLLED_OBJECT_H
#include "AbstractPosition.h"
#include "AbstractMarker.h"
#include "AbstractControlledObject.h"

class RemoteControlledObject : public AbstractControlledObject {
public:
    RemoteControlledObject();
    virtual ~RemoteControlledObject();
    void move(int spaces);
    void turnLeft();
    void turnRight();
    void setBoundaries(int rows, int columns);
    void setRow(int row);
    void setColumn(int column);
    void setPosition(int row, int column);
    int getRow();
    int getColumn();
    int getUpperRowBoundary();
    int getUpperColumnBoundary();
    void setMarkerUp();
    void setMarkerDown();
    void toggleMarker();
    bool isMarkerDown();

private:
    static int count;
    AbstractPosition* its_position;
    AbstractMarker* its_marker;
};
#endif

```

ABSTRACTCONTROLLEDRODENT.H*abstractcontrolledrodent.h*

```

#ifndef ABSTRACT_CONTROLLED_RODENT_H
#define ABSTRACT_CONTROLLED_RODENT_H
#include "remotecontrolledobject.h"

class AbstractControlledRodent : public RemoteControlledObject {
public:
    AbstractControlledRodent(){}
    virtual ~AbstractControlledRodent(){}
    virtual void setTailUp() = 0;
    virtual void setTailDown() = 0;
};
#endif

```

ROBOTRAT.H*robotrat.h*

```

#ifndef ROBOT_RAT_H
#define ROBOT_RAT_H
#include "abstractcontrolledrodent.h"

class RobotRat : public AbstractControlledRodent {
public:
    RobotRat(int row_boundary = 20, int column_boundary = 20);
    virtual ~RobotRat();
    RobotRat(RobotRat& rhs);
    RobotRat& operator=(RobotRat& rhs);
    void setTailUp();
    void setTailDown();
    char* getObjectType();
};

```

```
#endif
```

RODENTWORLD.H

rodentworld.h

```
#ifndef RODENT_WORLD_H
#define RODENT_WORLD_H

#include "abstractcontrolledrodent.h"
#include <cstdlib>

class RodentWorld {
public:
    RodentWorld(int rows = 20, int columns = 20);
    virtual ~RodentWorld();
    void addRodent();
    void deleteRodent();
    void toggleRodent();
        void turnRodentLeft();
        void turnRodentRight();
        void moveRodent(int spaces);
        void setRodentTailUp();
        void setRodentTailDown();
    char* getRodentType();
        int getRodentRow();
        int getRodentColumn();
        bool isRodentTailDown();

private:
    AbstractControlledRodent **rodent_array;
    int its_rows;
    int its_columns;
    int number_of_rodents;
    int rodent_number;

};
#endif
```

USERINTERFACE.H

userinterface.h

```
#ifndef USER_INTERFACE_H
#define USER_INTERFACE_H

#include <iostream>
using namespace std;

class UserInterface {
public:
    UserInterface();
    ~UserInterface();
    void displayMenu();
    int getMenuChoice();
    void markFloor(int row, int column);
    void clearFloor(int row, int column);
    void displayFloor();
    int getRows();
    int getColumns();
    int getSpaces();

private:
    bool** floor;
    int its_rows;
    int its_columns;
    int spaces;

};
#endif
```

CONTROLLER.H*controller.h*

```

#ifndef CONTROLLER_H
#define CONTROLLER_H

#include "rodentworld.h"
#include "userinterface.h"

class Controller {
public:
    Controller();
    ~Controller();
    void run();

private:
    UserInterface* its_ui;
    RodentWorld* its_world;
    enum MenuActions {ADD = 1,
                     TOGGLE,
                     TURNRIGHT,
                     TURNLEFT,
                     TAILUP,
                     TAILDOWN,
                     MOVE,
                     DISPLAYFLOOR,
                     EXIT};

    void move();
};
#endif

```

position.cpp*position.cpp*

```

#include "position.h"
#include <assert.h>

Position::Position(int row, int column):its_row(row), its_column(column),
lower_row_boundary(0), upper_row_boundary(20),
lower_column_boundary(0), upper_column_boundary(20),
its_direction(EAST) {}

Position::~Position() {}

void Position::setRow(int row){
    its_row = row;
}

void Position::setColumn(int column){
    its_column = column;
}

void Position::incrementRow(int val){
    int new_row = its_row + val;
    //assert((lower_row_boundary <= new_row) && (new_row < upper_row_boundary));
    if(new_row < lower_row_boundary){
        its_row = lower_row_boundary;
    }else if(new_row > (upper_row_boundary-1)){
        its_row = (upper_row_boundary-1);
    }else its_row = new_row;
}

void Position::incrementColumn(int val){
    int new_column = its_column + val;
    //assert((lower_column_boundary <= new_column) && (new_column < upper_column_boundary));
    if(new_column < lower_column_boundary){
        its_column = lower_row_boundary;
    }else if(new_column > (upper_column_boundary-1)){
        its_column = (upper_column_boundary-1);
    }
}

```

```

        }else its_column = new_column;
    }

int Position::getRow(){
    return its_row;
}

int Position::getColumn(){
    return its_column;
}

void Position::setLowerRowBoundary(int lower){
    lower_row_boundary = lower;
}

void Position::setUpperRowBoundary(int upper){
    upper_row_boundary = upper;
}

void Position::setLowerColumnBoundary(int lower){
    lower_column_boundary = lower;
}

void Position::setUpperColumnBoundary(int upper){
    upper_column_boundary = upper;
}

void Position::move(int spaces){
    switch(its_direction){
        case NORTH: incrementRow(spaces * -1);
                    break;
        case EAST:  incrementColumn(spaces);
                    break;
        case SOUTH: incrementRow(spaces);
                    break;
        case WEST:  incrementColumn(spaces * -1);
                    break;
        default: break;
    }
}

void Position::turnLeft(){
    switch(its_direction){
        case NORTH: its_direction = WEST;
                    break;
        case EAST:  its_direction = NORTH;
                    break;
        case SOUTH: its_direction = EAST;
                    break;
        case WEST:  its_direction = SOUTH;
                    break;
        default:   its_direction = EAST;
    }
}

void Position::turnRight(){
    switch(its_direction){
        case NORTH: its_direction = EAST;
                    break;
        case EAST:  its_direction = SOUTH;
                    break;
        case SOUTH: its_direction = WEST;
                    break;
        case WEST:  its_direction = NORTH;
                    break;
        default:   its_direction = EAST;
    }
}
}

```

```

void Position::setBoundaries(int rows, int columns){
    upper_row_boundary = rows, upper_column_boundary = columns;
}

void Position::setPosition(int row, int column){
    its_row = row, its_column = column;
}

int Position::getUpperRowBoundary(){
    return upper_row_boundary;
}

int Position::getUpperColumnBoundary(){
    return upper_column_boundary;
}

```

MARKER.CPP*marker.cpp*

```

#include "marker.h"

Marker::Marker():its_marker_position(UP){}

Marker::~Marker(){}

void Marker::setMarkerUp(){
    its_marker_position = UP;
}

void Marker::setMarkerDown(){
    its_marker_position = DOWN;
}

void Marker::toggleMarker(){
    switch(its_marker_position){
        case UP: its_marker_position = DOWN;
                break;
        case DOWN: its_marker_position = UP;
                 break;
        default: its_marker_position = UP;
    }
}

bool Marker::isMarkerDown(){
    return (bool)its_marker_position;
}

```

REMOTECONTROLLEDOBJECT.CPP*remotecontrolledobject.cpp*

```

#include "remotecontrolledobject.h"
#include "position.h"
#include "marker.h"

int RemoteControlledObject::count = 0;

RemoteControlledObject::RemoteControlledObject(){
    its_position = new Position;
    its_marker = new Marker;
    count++;
}

RemoteControlledObject::~RemoteControlledObject(){
    count--;
    delete its_position;
    delete its_marker;
}

void RemoteControlledObject::move(int spaces){
    its_position->move(spaces);
}

```

```

void RemoteControlledObject::turnLeft() {
    its_position->turnLeft();
}

void RemoteControlledObject::turnRight() {
    its_position->turnRight();
}

void RemoteControlledObject::setRow(int row) {
    its_position->setRow(row);
}

void RemoteControlledObject::setColumn(int column) {
    its_position->setColumn(column);
}

void RemoteControlledObject::setPosition(int row, int column) {
    its_position->setPosition(row, column);
}

int RemoteControlledObject::getRow() {
    return its_position->getRow();
}

int RemoteControlledObject::getColumn() {
    return its_position->getColumn();
}

void RemoteControlledObject::setBoundaries(int rows, int columns) {
    its_position->setBoundaries(rows, columns);
}

int RemoteControlledObject::getUpperRowBoundary() {
    return its_position->getUpperRowBoundary();
}

int RemoteControlledObject::getUpperColumnBoundary() {
    return its_position->getUpperColumnBoundary();
}

void RemoteControlledObject::setMarkerUp() {
    its_marker->setMarkerUp();
}

void RemoteControlledObject::setMarkerDown() {
    its_marker->setMarkerDown();
}

void RemoteControlledObject::toggleMarker() {
    its_marker->toggleMarker();
}

bool RemoteControlledObject::isMarkerDown() {
    return its_marker->isMarkerDown();
}

```

ROBOTRAT.CPP

robotrat.cpp

```

#include "robotrat.h"

RobotRat::RobotRat(int row_boundary, int column_boundary) {
    setBoundaries(row_boundary, column_boundary);
}

RobotRat::~RobotRat() {}

RobotRat::RobotRat(RobotRat& rhs) {
    setBoundaries(rhs.getUpperRowBoundary(), rhs.getUpperColumnBoundary());
    if (rhs.isMarkerDown()) {
        this->setMarkerDown();
    } else {
        this->setMarkerUp();
    }
}

```



```

    }

    this->setPosition(rhs.getRow(), rhs.getColumn());
}

RobotRat& RobotRat::operator=(RobotRat& rhs) {

    this->setBoundaries(rhs.getUpperRowBoundary(), rhs.getUpperColumnBoundary());
    if (rhs.isMarkerDown()) {
        this->setMarkerDown();
    } else {
        this->setMarkerUp();
    }

    this->setPosition(rhs.getRow(), rhs.getColumn());
    return *this;
}

void RobotRat::setTailUp() {
    setMarkerUp();
}

void RobotRat::setTailDown() {
    setMarkerDown();
}

char* RobotRat::getObjectType() {
    return "RobotRat";
}

```

RODENTWORLD.CPP

rodentworld.cpp

```

#include "rodentworld.h"
#include "robotrat.h"

RodentWorld::RodentWorld(int rows, int columns):its_rows(rows),its_columns(columns),
                                                number_of_rodents(1), rodent_number(0) {
    rodent_array = new AbstractControlledRodent*[number_of_rodents];
    rodent_array[0] = new RobotRat(its_rows, its_columns);
}

RodentWorld::~RodentWorld() {
    for(int i = 0; i<number_of_rodents; i++){
        delete rodent_array[i];
    }

    delete[] rodent_array;
}

void RodentWorld::addRodent() {
    //save current number of rodents
    int old_number_of_rodents = number_of_rodents;

    //create temp array same size as current array
    AbstractControlledRodent **temp_array = new AbstractControlledRodent*[number_of_rodents];

    //assign rodents to temp array
    for(int i=0; i<number_of_rodents; i++){
        temp_array[i] = rodent_array[i];
    }

    //then delete rodent array
    delete[] rodent_array;

    //now, create new rodent array one bigger than old rodent array
    rodent_array = new AbstractControlledRodent*[++number_of_rodents];

    //copy temp rodent objects into new rodent array
    for(int i=0; i<old_number_of_rodents; i++){
        rodent_array[i] = temp_array[i];
    }
}

```

```

//add new rodent object to the last array element
rodent_array[number_of_rodents-1] = new RobotRat(its_rows, its_columns);

//delete temp array
delete[] temp_array;
toggleRodent();
}

void RodentWorld::deleteRodent(){
//delete currently selected rodent object
delete rodent_array[rodent_number];

//then set pointer to null
rodent_array[rodent_number] = NULL;

//save previous number of rodents
int old_number_of_rodents = number_of_rodents;

//create temp array one smaller than rodent array
AbstractControlledRodent **temp_array = new AbstractControlledRodent*[--number_of_rodents];

//copy surviving rodent objects to temp array
{
int i(0), j(0);
while(i < old_number_of_rodents){
if(rodent_array[i] != NULL){
temp_array[j++] = rodent_array[i++];
}else{i++;}
}
}

//delete rodent_array
delete[] rodent_array;

//create new rodent array one smaller than old array
rodent_array = new AbstractControlledRodent*[number_of_rodents];

//Assign rodents from temp array
for(int i=0; i<number_of_rodents; i++){
rodent_array[i] = temp_array[i];
}

//delete temp array
delete[] temp_array;

toggleRodent();
}

void RodentWorld::toggleRodent(){
if(rodent_number < (number_of_rodents-1)){
rodent_number++;
}else{
rodent_number = 0;
}
}

void RodentWorld::turnRodentLeft(){
rodent_array[rodent_number]->turnLeft();
}

void RodentWorld::turnRodentRight(){
rodent_array[rodent_number]->turnRight();
}

void RodentWorld::moveRodent(int spaces){
rodent_array[rodent_number]->move(spaces);
}

void RodentWorld::setRodentTailUp(){
rodent_array[rodent_number]->setTailUp();
}

void RodentWorld::setRodentTailDown(){

```

```

    rodent_array[rodent_number]->setTailDown();
}

char* RodentWorld::getRodentType() {
    return rodent_array[rodent_number]->getObjectType();
}

bool RodentWorld::isRodentTailDown() {
    return rodent_array[rodent_number]->isMarkerDown();
}

int RodentWorld::getRodentRow() {
    return rodent_array[rodent_number]->getRow();
}

int RodentWorld::getRodentColumn() {
    return rodent_array[rodent_number]->getColumn();
}

```

USERINTERFACE.CPP

userinterface.cpp

```

#include "userinterface.h"
#include <stdlib.h>

UserInterface::UserInterface():spaces(0) {
    int rows(0), columns(0);
    cout<<"Enter Max Rows: ";
    cin>>rows;
    cout<<endl<<"Enter Max Columns: ";
    cin>>columns;
    its_rows = rows;
    its_columns = columns;
    floor = new bool*[rows];
    for(int i = 0; i<rows; i++){
        floor[i] = new bool[columns];
    }

    for(int i=0; i<rows; i++){
        for(int j=0; j<columns; j++){
            floor[i][j] = false;
        }
    }
}

UserInterface::~UserInterface() {
    for(int i=0; i<its_rows; i++){
        delete[] floor[i];
    }

    delete[] floor;
}

void UserInterface::displayMenu() {
    cout<<"    1. Add New Rat"<<endl;
    cout<<"    2. Toggle Rat"<<endl;
    cout<<"    3. Turn Right"<<endl;
    cout<<"    4. Turn Left"<<endl;
    cout<<"    5. Set Tail Up"<<endl;
    cout<<"    6. Set Tail Down"<<endl;
    cout<<"    7. Move"<<endl;
    cout<<"    8. Display Floor"<<endl;
    cout<<"    9. Exit"<<endl;
    cout<<endl;
    cout<<endl;
}

int UserInterface::getMenuChoice() {
    char c[1];
    cin>>c[0];
    while((c[0] < '1') || (c[0] > '9')) {
        cout<<"Please enter a valid menu choice"<<endl;
        cin>>c;
    }

    return atoi(c);
}

```

```

void UserInterface::markFloor(int row, int column){
    floor[row][column] = true;
}

void UserInterface::clearFloor(int row, int column){
    floor[row][column] = false;
}

void UserInterface::displayFloor() {
    for(int i = 0; i<its_rows; i++){
        for(int j = 0; j<its_columns; j++){
            if(floor[i][j]){
                cout<<'*';
            }else{
                cout<<'0';
            }
        }
        cout<<endl;
    }
}

int UserInterface::getRows() {
    return its_rows;
}

int UserInterface::getColumns() {
    return its_columns;
}

int UserInterface::getSpaces() {
    cout<<endl<<"Enter spaces to move: ";
    cin>>spaces;
    return spaces;
}

```

CONTROLLER.CPP

controller.cpp

```

#include "controller.h"

Controller::Controller() {
    its_ui = new UserInterface();
    its_world = new RodentWorld(its_ui->getRows(), its_ui->getColumns());
}

Controller::~Controller() {
    delete its_ui;
    delete its_world;
}

void Controller::run() {
    while(true){
        its_ui->displayMenu();
        switch(its_ui->getMenuChoice()) {
            case ADD: its_world->addRodent();
                    break;
            case TOGGLE: its_world->toggleRodent();
                    break;
            case TURNRIGHT: its_world->turnRodentRight();
                    break;
            case TURNLEFT: its_world->turnRodentLeft();
                    break;
            case TAILUP: its_world->setRodentTailUp();
                    break;
            case TAILDOWN: its_world->setRodentTailDown();
                    break;
            case MOVE: move();
                    break;
            case DISPLAYFLOOR: its_ui->displayFloor();
                    break;
            case EXIT: exit(0);
                    break;
            default: break;
        }
    }
}

```

```

}

void Controller::move(){
    int spaces = its_ui->getSpaces();
    while(spaces--){
        if(its_world->isRodentTailDown()){
            its_ui->markFloor(its_world->getRodentRow(), its_world->getRodentColumn());
            its_world->moveRodent(1);
        } else its_world->moveRodent(1);
    }
}
}

```

main.cpp*main.cpp*

```

#include <iostream>
using namespace std;
#include "controller.h"
#include "robotrat.h"

int main (){

    Controller controller;
    controller.run();
}

```

Skill Building Exercises

- 1. Procure and Install UML Modeling Tool:** Procure and install a UML modeling tool that is appropriate for the computer platform you are using. If you bought the printed edition or CD-ROM edition of C++ For Artists there are several UML tools included on the CD-ROM. If you purchased the ebook/PDF edition of C++ For Artists then you will have to download the tool from the web.
- 2. Reverse Engineering:** Use your UML modeling tool to reverse engineer an existing C++ project.
- 3. Create Use Case Diagram:** Use your UML modeling tool to create a use case diagram. Practice creating actors, use cases, and communication links. Learn how to add documentation to your use case diagrams.
- 4. Create Class Diagrams:** Use your UML modeling tool to create a class diagram. Practice creating classes and adding attributes and operations. Experiment with different class property settings.
- 5. Create Sequence Diagrams:** Use your UML modeling tool to create sequence diagrams. Practice creating object lifetime bars and messages.
- 6. Linking Diagram Elements to Other Diagrams:** If your UML modeling tool supports this feature, link a use case diagram element with another diagram. For example, link a use case diagram element to a class or sequence diagram.
- 7. Generating Documentation and Reports:** Practice generating documentation or reports from your UML modeling tool if it supports this feature. Experiment with different settings and note the results.
- 8. Reverse Engineering:** Reverse engineer the polymorphic engine component code given in chapter 16.

SUGGESTED PROJECTS

1. **Banking Application:** Use your UML tool to design and implement a banking application that allows customers to open different types of accounts and make deposits to and withdrawals from those accounts. Store account balance and transaction information in text files.
2. **Travel Agent Booking System:** Use your UML tool to design and implement a travel agent booking system that allows travel agents to book travel arrangements for customers.
3. **Gas Station Simulation:** Use your UML modeling tool to design and implement a gas station simulation. The simulation should model the fuel storage tanks, their imbedded pumps, and the gas nozzle handle that starts and stops the pump. Keep track of how much gas a user pumps and indicate the total liters or gallons purchased and the total price of the sale.
4. **University Course Enrollment System:** Use your UML modeling tool to design, document, and implement a university course enrollment system.
5. **Hospital Patient Management System:** Use your UML modeling tool to design, document, and implement a hospital patient management system.
6. **Pharmacy Robot Pill Dispenser:** Use your UML modeling tool to design, document, and implement a pharmacy robot pill dispenser.
7. **Health Club Membership System:** Use your UML modeling tool to design, document, and implement a health club membership system.
8. **Flower Shop Management System:** Use your UML modeling tool to design, document, and implement a flower shop management system.
9. **Auto Repair Shop Appointment System:** Use your UML modeling tool to design, document, and implement an auto repair shop appointment system.
10. **Rental Car Agency Management System:** Use your UML modeling tool to design, document, and implement a rental car agency management system. The rental agency rents different types of vehicles to include trucks, sport, and luxury models.

SELF TEST QUESTIONS

1. What is the purpose of a UML modeling tool?
2. In what two primary ways does a UML modeling tool help designers?
3. Why do you suppose it would be important for a UML modeling tool to support collaborative design and development.
4. What is meant by the term reverse engineering as it applies to UML modeling tools?
5. Why do you think web reporting is an important UML modeling tool feature?

6. What is the purpose of a use case diagram?
7. What is the purpose of a class diagram?
8. What is the purpose of a sequence diagram?
9. What piece of a design should one sequence diagram document?
10. (T/F) Describe™ can link diagram elements to outside documents and URLs.

REFERENCES

Sinan Si Alhir. *UML In A Nutshell: A Desktop Quick Reference*. O'Reilly & Associates, Inc. 101 Morris Street, Sebastopol, CA 95472. ISBN: 1-56592-448-7

Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts. ISBN: 0-201-57168-4

Embarcadero Technologies Describe™ online help.

Embarcadero Technologies Describe™ User Guide.

NOTES

APPENDICES

Appendix A

PROJECT APPROACH STRATEGY CHECKOFF LIST

Project Approach Strategy Checkoff List

Table Appendix A-1: Project Approach Strategy Checkoff List

	Strategy Area	Explanation
_____	Requirements	Determine and clarify exactly what purpose and features the finished project must have. Clarify your understanding of the requirements with your instructor if the project specification is not clear. The result of pursuing this strategy area should be a clear definition of what problem must be solved.
_____	Problem Domain	Study the problem until you have a firm understanding of how to solve it. Optionally, express your understanding of the solution by writing a pseudocode algorithm that describes, step-by-step, how the problem can be solved. The result of this strategy area should be a high-level solution statement that can be translated into a detailed application design.
_____	Language Features	Make a list of all the language features you must understand and use to draft a competent design and later implement your design. As you study each language feature check it off your list. Doing so will give you a sense of progress. The result of this strategy area should be a complete understanding of all C++ language features required to effect a good design and solve the problem.
_____	Design (Plan)	Sketch out a rough application design. The design should address issues such as data structures, Input/Output, and how you plan to execute the problem solution you derived in the Problem Domain strategy area. The result of this strategy area will be a clear understanding of what source code should be written.

Appendix B

ASCII Table

ASCII Table

Table Appendix B-1: ASCII Table

Decimal	Octal	Hex	Binary	Value	Comment
000	000	000	00000000	NUL	Null char
001	001	001	00000001	SOH	Start of Header
002	002	002	00000010	STX	Start of Text
003	003	003	00000011	ETX	End of Text
004	004	004	00000100	EOT	End of Transmission
005	005	005	00000101	ENQ	Enquiry
006	006	006	00000110	ACK	Acknowledgment
007	007	007	00000111	BEL	Bell
008	010	008	00001000	BS	Backspace
009	011	009	00001001	HT	Horizontal Tab
010	012	00A	00001010	LF	Line Feed
011	013	00B	00001011	VT	Vertical Tab
012	014	00C	00001100	FF	Form Feed
013	015	00D	00001101	CR	Carriage Return
014	016	00E	00001110	SO	Shift Out
015	017	00F	00001111	SI	Shift In
016	020	010	00010000	DLE	Data Link Escape
017	021	011	00010001	DC1	XON Device Control 1
018	022	012	00010010	DC2	Device Control 2
019	023	013	00010011	DC3	XOFF Device Control 3
020	024	014	00010100	DC4	Device Control 4
021	025	015	00010101	NAK	Negative Acknowledgement
022	026	016	00010110	SYN	Synchronous Idle
023	027	017	00010111	ETB	End of Trans. Block
024	030	018	00011000	CAN	Cancel
025	031	019	00011001	EM	End of Medium
026	032	01A	00011010	SUB	Substitute
027	033	01B	00011011	ESC	Escape
028	034	01C	00011100	FS	File Separator
029	035	01D	00011101	GS	Group Separator
030	036	01E	00011110	RS	Request to Send Record Separator

Table Appendix B-1: ASCII Table

Decimal	Octal	Hex	Binary	Value	Comment
031	037	01F	00011111	US	Unit Separator
032	040	020	00100000	SP	Space
033	041	021	00100001	!	
034	042	022	00100010	"	
035	043	023	00100011	#	
036	044	024	00100100	\$	
037	045	025	00100101	%	
038	046	026	00100110	&	
039	047	027	00100111	'	
040	050	028	00101000	(
041	051	029	00101001)	
042	052	02A	00101010	*	
043	053	02B	00101011	+	
044	054	02C	00101100	,	
045	055	02D	00101101	-	
046	056	02E	00101110	.	
047	057	02F	00101111	/	
048	060	030	00110000	0	
049	061	031	00110001	1	
050	062	032	00110010	2	
051	063	033	00110011	3	
052	064	034	00110100	4	
053	065	035	00110101	5	
054	066	036	00110110	6	
055	067	037	00110111	7	
056	070	038	00111000	8	
057	071	039	00111001	9	
058	072	03A	00111010	:	
059	073	03B	00111011	;	
060	074	03C	00111100	<	
061	075	03D	00111101	=	
062	076	03E	00111110	>	
063	077	03F	00111111	?	
064	100	040	01000000	@	
065	101	041	01000001	A	
066	102	042	01000010	B	
067	103	043	01000011	C	
068	104	044	01000100	D	
069	105	045	01000101	E	
070	106	046	01000110	F	
071	107	047	01000111	G	
072	110	048	01001000	H	
073	111	049	01001001	I	
074	112	04A	01001010	J	
075	113	04B	01001011	K	
076	114	04C	01001100	L	
077	115	04D	01001101	M	
078	116	04E	01001110	N	
079	117	04F	01001111	O	
080	120	050	01010000	P	
081	121	051	01010001	Q	
082	122	052	01010010	R	
083	123	053	01010011	S	
084	124	054	01010100	T	
085	125	055	01010101	U	
086	126	056	01010110	V	

Table Appendix B-1: ASCII Table

Decimal	Octal	Hex	Binary	Value	Comment
087	127	057	01010111	w	
088	130	058	01011000	x	
089	131	059	01011001	y	
090	132	05A	01011010	z	
091	133	05B	01011011	[
092	134	05C	01011100	\	
093	135	05D	01011101]	
094	136	05E	01011110	^	
095	137	05F	01011111	_	
096	140	060	01100000	`	
097	141	061	01100001	a	
098	142	062	01100010	b	
099	143	063	01100011	c	
100	144	064	01100100	d	
101	145	065	01100101	e	
102	146	066	01100110	f	
103	147	067	01100111	g	
104	150	068	01101000	h	
105	151	069	01101001	i	
106	152	06A	01101010	j	
107	153	06B	01101011	k	
108	154	06C	01101100	l	
109	155	06D	01101101	m	
110	156	06E	01101110	n	
111	157	06F	01101111	o	
112	160	070	01110000	p	
113	161	071	01110001	q	
114	162	072	01110010	r	
115	163	073	01110011	s	
116	164	074	01110100	t	
117	165	075	01110101	u	
118	166	076	01110110	v	
119	167	077	01110111	w	
120	170	078	01111000	x	
121	171	079	01111001	y	
122	172	07A	01111010	z	
123	173	07B	01111011	{	
124	174	07C	01111100		
125	175	07D	01111101	}	
126	176	07E	01111110	~	
127	177	07F	01111111	DEL	

Appendix C

ANSWERS TO SELF TEST QUESTIONS

Chapter 1

1. List at least seven skills you must master in your studies of the C++ programming language.

A development environment, computing platform, problem solving skills, how to approach a programming project, how to manage project complexity, how to put yourself in the mood to program, how to stimulate your creative abilities, object-oriented analysis and design, object-oriented programming principles...

2. What three development roles will you play as a student?

Analyst, Architect, Programmer

3. What is the purpose of the project approach strategy?

To help you maintain your development momentum

4. List and describe the four areas of concern addressed in the project approach strategy.

Requirements - Expected program functionality.

Problem Domain - Learn enough about the problem to be solved.

Programming language features - Learn the required C++ language features to write program.

Design - Design your solution

5. List and describe the five steps of the programming cycle.

Plan - Take a piece of the program design and formulate implementation

Code - Program the piece of the design you selected

Test - Ensure the piece of code you wrote works

Integrate - Fold newly tested code into the bigger program and test results

Factor - Improve the design where possible

Repeat - Goto Plan

6. What are the two types of complexity?

Physical and Conceptual

7. List several benefits to splitting even small projects into multiple files.

Helps manage physical complexity & helps separate interface from implementation

8. Discuss the concept of interface vs. implementation. How to you separate the interface of a class from its implementation?

Declare the class in a header file and implement the class in an implementation file.

9. What preprocessor directives can be used to allow multiple inclusion of header files?

#ifndef, #define, #endif

10. List at least three things that can be contained in header files.

class declarations, structure declarations, enum declarations, function declarations...

11. List three things that shouldn't be contained in header files.

variable declarations, class or function definitions, exported template definitions

12. Why do you think it would be helpful to write self-commenting source code?

It will make your source code easier to read and maintain.

13. What can you do in your source code to maximize cohesion?

Write well-named functions that have a singular purpose. Functions should not pull any surprises.

14. What can you do in your source code to minimize coupling?

Do not use global variables.

Chapter 2

1. Describe the program creation process.

*Write program in programming language like C++.
Save program in text file. This is known as a source file.
Compile source file into object file with program called a compiler.
Link object file to other object files with program called a linker. This results in an executable file.*

2. What is the purpose of the C++ preprocessor?

The preprocessor performs a transformation on source code files before they are compiled. The

preprocessor acts of special preprocessor commands like #include, #ifndef, #define, #endif, etc...

3. What is the purpose of the compiler?

The compiler transforms source code files into machine instruction object modules targeted for a specific hardware platform.

4. What is the purpose of the linker?

The linker binds a symbolic name to the address at which the code for the symbolic name is located. The linker allows you to call a function in your program that is actually contained in another code module (i.e., a static library perhaps)

5. What is the primary benefit of using an integrated development environment?

You get a text editor, compiler, linker, debugger, and project manager all in one integrated product.

6. List at least three features of an integrated development environment.

Text editor, compiler, linker, project management, debugger

7. What is the purpose of the UNIX make utility?

To build complex projects. The make utility uses a makefile that contains project build dependency information. You have to manually create makefiles unless you are using an integrated development environment.

Chapter 3

1. What is the purpose of the project approach strategy?

The project approach strategy will help you maintain a feeling of forward momentum and a sense of progress when working on difficult programming projects.

2. What is the purpose of the development cycle?

The development cycle provides you with a repeatable process that will result in the successful implementation of your project.

3. Describe how to apply the project approach strategy and development cycle in an iterative fashion.

When faced with your first, or difficult, project, use the approach strategy to clarify the project requirements, problem domain, language features, and project design approach. When you are ready to proceed with the implementation phase of your project you apply the development cycle by taking a piece of your design, planning its implementation, coding the implementation, testing the code, and integrating the results with previous project components if required. Repeat the development cycle as required until you complete the project.

4. Why is it a good idea to do just enough design to get started coding? Does this approach have practical application

in the real programming world? What future problems regarding application design does using this approach help to avoid?

*You want to test your design for soundness. (Is it a good design?) The idea is to catch design problems early in the project. (The earlier the better)
Yes, the approach has practical application in the real world.
This approach will help detect serious problems with application architecture.*

5. How is function stubbing used in the robot rat project?

Function stubbing is used to defer the implementation of certain functions until after the application framework has been tested.

6. Why is component testing important?

Component testing is important because you want to verify the proper operation of component before integrating them into the bigger program.

7. Why is frequent component integration important?

You want to detect any potential problems as soon as possible. Early and frequent integration will answer the question: "Did my new code break my good, tested code?"

8. What is the purpose of pseudocode?

To describe processing steps in natural language that can be easily translated into a programming language.

9. What is the purpose of a state transition diagram?

To illustrate valid object states and the events that result in object state changes.

10. What C++ flow control structure can be used to implement the functionality described by a state transition diagram?

switch statement

Chapter 4

1. List at least five components of a typical computer system.

System unit, speakers, monitor, hard disk drive, floppy disk drive, memory, etc....

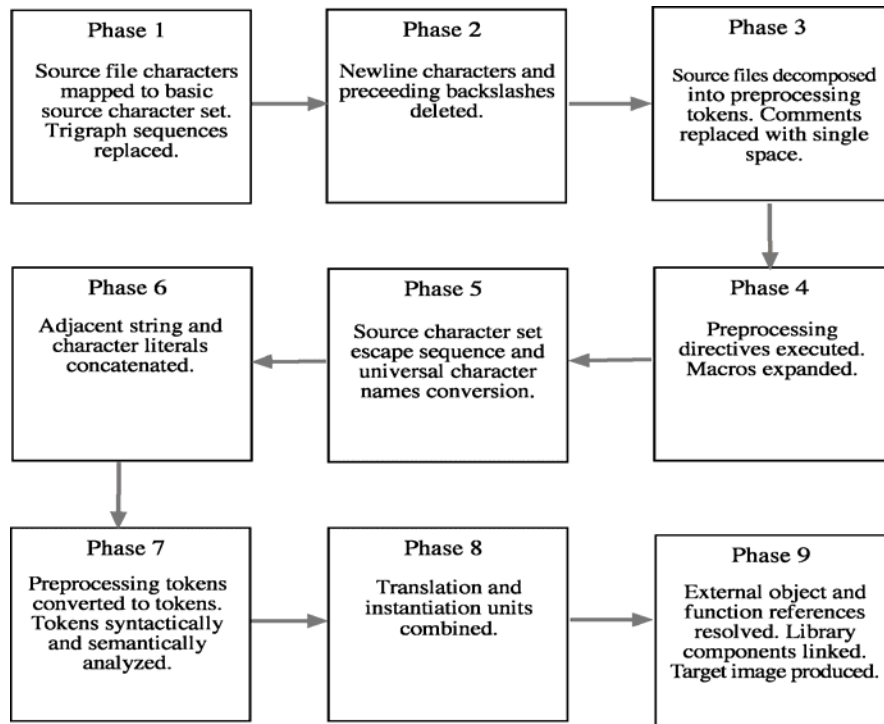
2. What device do the peripheral components of a computer system exist to support?

Processor or Microprocessor

3. From what two perspectives can programs be viewed? How does each perspective differ from the other?

The computer perspective and the human perspective

4. List the nine phases of the C++ translation process.



5. What are the function of trigraphs?

Trigraphs are used by programmers to write source code on terminals that lack the special characters required by the basic source character set.

6. What is a C++ translation unit?

A translation unit consists of a source file, its headers and any source files included via the #include preprocessor directive, minus any source lines skipped by conditional compilation.

7. List and describe the four steps of the processing cycle?

*Fetch - Fetch an instruction from memory
 Decode - Translate the instruction into electronic signals that control the processor
 Execute - Carry out the operations required by the instruction
 Store - Store the results of the instruction in memory*

8. State in your own words the definition of an algorithm.

A recipe for solving a problem.

9. How does a processor's architecture serve to implement its feature set?

A processor's architecture dictates what features the processor offers.

10. How can programmers access a processor's feature set?

Via the processor's assembly language.

Chapter 5

1. What is the purpose of the main() function? Why does every C++ program need one?

The main() function is the execution entry point for C++ programs. Every C++ program needs a main function so execution can be passed to it when the program is executed by the computer system.

2. Describe how disassembling your programs and studying the results can improve your understanding of the C++ language and your computing platform.

By disassembling C++ programs you get a feel for how program elements map to the processor's assembly language.

3. What is the purpose of the preprocessor directive #include?

The #include directive is replaced by the contents of the file it names during preprocessing.

4. (True/False) You can use the C++ reserved keywords to name your own program objects.

FALSE

5. The sizeof operator will return the size of data types in what unit of measure?

Allocation units

6. You can either memorize C++ operator precedence or use _____ to force precedence and make source code easier to read at the same time.

parentheses

7. Why can an unsigned data type represent a larger positive value than a signed data type?

Because the most significant bit is not used to indicate a negative value.

8. When negative numbers are shifted to the right using the >> operator what is the effect on the sign bit?

It is carried along with the shift.

9. How are string literals terminated?

With \0 or null termination.

10. Study the following code and answer the following questions:

```

1  #include <iostream>
2  using namespace std;
3
4  int val1 = 1;
5
6  int main() {
7      {
8          cout<<val1++<<endl;
9          int val1 = ::val1;
10         cout<<val1<<endl;
11     }
12     cout<<val1<<endl;
13     return 0;
14 }

```

-What value will the new variable named val1 be initialized to on line 9?

2

-Which val1 will be printed to the screen on line 10?

The val1 local to the block declared within the main() function.

-What is the value of val1 printed to the screen on line 12?

2

-Describe the effect of using the unary scope resolution operator :: on line 9. Why is its use necessary?

The val1 declared on line 9 conflicts or hides the global val1 declared on line 4. The :: operator is necessary to resolve the variable names.

Chapter 6

1. What characters are used to begin and end compound statements?

opening and closing braces { ... }

2. How does the if-else statement differ from the if statement?

The if...else statement offers an alternative course of action in the event the expression evaluates to false.

3. What control statement can be used in place of nested if-else statements?

switch

4. Why should a break statement be added to the end of each case of a switch statement?

To prevent execution falling through to the next case statement.

5. What is the purpose of a default case in a switch statement?

To provide a default course of action should no previous case apply.

6. What is the difference between a while and a do-while statement? When would you use a do-while instead of a while statement?

A do...while executes the body at least once regardless of the value of the expression. Use a do...while when you have to execute the body of the statement at least once.

7. Convert the following while statement to a for statement:

```
int i = 0;
while (i < 10) {
    //do something
    i++;
}
```

```
for(int i=0; i<10; i++){
    //do something
}
```

8. What will happen when the following code executes:

```
int i = 0;
while (i++ < 3) {
    cout << "i = " << i << endl;
    break;
}
```

The value of i is incremented to 1 and the string i = 1 is printed to the screen, then the break causes the while loop to terminate.

9. What happens when the following code executes:

```
int i = 0;
while (i < 3) {
    cout << "i = " << i << endl;
    continue;
    i++;
}
```

The string i = 0 is repeatedly printed to the screen. The continue forces the while loop to repeat forever because i is never allowed to increment.

10. True/False: Goto statements can be used to jump outside of functions.

FALSE

Chapter 7

1. What is an object? What is the difference between an object and the identifier name used to reference the object?

An object is a region of memory. The identifier name simply points to the object, or rather, the object is accessed via the identifier name.

2. Describe in general terms how computer memory is typically organized.

Computer memory systems are typically byte addressable and word aligned.

3. What operator do you use to determine an object's memory address? Give an example of its use.

The & operator. To get the address of an object, append the & operator to the object's identifier like so:

&objectName

4. What is a pointer?

A pointer is a variable that can contain a memory address.

5. What character do you use to declare a pointer? Give several examples of its use.

*Use the * operator. For example:*

```
int* i = new int(3); //declares a new int pointer and sets its value to 3  
ObjectName* objptr; //declares a new ObjectName pointer
```

6. What operator do you use to access an object via a pointer? Give an example of its use.

*To access an object via a pointer you have to dereference the pointer. Use the * operator to dereference pointers. For example:*

```
*i = 4; //set the value of the integer object pointed to by i to 4
```

7. What character is used to declare a reference?

The & operator.

8. What are the primary differences between a pointer and a reference?

A pointer is a variable and can be changed. A reference cannot be changed once initialized.

9. How do you access the object referred to by a reference? How is this different from accessing an object via a pointer?

You use references just like regular identifier names. A reference is like a nickname for an object.

10. (T/F) A pointer is a variable.

TRUE

Chapter 8

1. An array is a _____ allocation of memory to _____ objects.

contiguous / homogeneous

2. A static array name is what type of pointer?

const pointer

3. (T/F) An array name points to the first element of the array.

TRUE

4. How are multi-dimensional arrays stored in memory.

Row major order

5. Multi-dimensional arrays are arrays of _____.

arrays

6. What determines the size of an allocation unit?

The data type.

7. How does a dynamically allocated array differ from a statically allocated array?

The elements of a dynamically allocated array are stored in heap memory.

8. List at least four types of objects an array can contain.

Fundamental data types
User-defined data types
Pointers to fundamental data types
Pointers to user-defined types
Pointers to members
Enumerated types
Other arrays

9. Describe the two methods available to access array elements.

Subscript method
Pointer arithmetic method

10. To what values will the array elements be initialized to in the following declaration:

```
int int_array[25];
```

The array will contain garbage values.

11. To what values will each array element be initialized to in the following declaration:

```
int int_array[25] = {1, 2, 3};
```

The first three elements will be 1, 2, 3, and the rest will be initialized to 0.

12. What is the difference between the new operator and the new[] operator?

The new operator allocates memory for new objects; the new[] operator allocates memory for arrays

13. Why is it important to release dynamically allocated array memory with the delete[] operator?

Because the delete[] operator is the array equivalent to the delete operator. Any memory allocated with new[] should be released with delete[].

14. List and discuss the steps required to create a single-dimensional dynamic array.

*Declare a pointer to the type of array elements
Create the array with the new[] operator
Create each element object as required with the new operator
---- when done with array
Delete each element object with the delete operator
Delete the array pointer with the delete[] operator*

15. Discuss the two methods for creating multi-dimensional dynamic arrays. Which method would you prefer to use and why?

*The first method is to statically allocate the smallest set of elements and dynamically allocate the largest
The second method is to dynamically allocate all dimensions of the multi-dimensional array. A multi-dimensional array is just an array of arrays.*

Chapter 9

1. Describe in your own words the definition of a function.

A function is a logical, named grouping of related code statements designed to perform a specific processing activity.

2. List and describe the three characteristics of a good function.

Maximally cohesive (singular purpose), well-named, and minimally coupled.

3. What is the difference between a function's interface and its implementation? What constitutes a function's interface? What constitutes a function's implementation? Why is it important or desirable to separate a function's interface from its implementation?

The function interface consists of the function name, its parameter list (number and type of parameters), and return type. The function implementation consists of the required code to implement the interface. Function users should care only about the function interface. It's important to separate the interface from the implementation because the interface can be distributed in a header file and the implementation can be distributed as a library module, keeping the source code secrets safe from tampering.

4. What is the purpose of the #ifndef, #define, & #endif preprocessor directives as they apply to header files?

To prevent multiple header file inclusions. Header files must often be #include(d) in multiple source files. The use of #ifndef, #define, #endif directives allows the header file to be used in multiple source files but limits the actual importing of the contents to one time.

5. List at least three benefits to giving functions good names.

To aid code readability and to make it easy for users to understand the purpose of your function.

6. In what two ways can arguments be passed to functions? What is the difference between the two ways? What advantages or disadvantages are associated with each way?

Arguments can be passed to functions by value or by reference. When passed by value, a copy of the object is made and passed to the function. For large object, pass by value may be an intensive operation. When passed by reference, a pointer (or reference) to the object is passed and the object is manipulated via the pointer. (pointer or reference)

7. What is the difference between an automatic local variable and a static local variable?

An automatic local variable will be created and initialized each time the function is called, whereas the static local variable will be created once and preserved across function calls.

8. What is meant by the phrase, "Maximize Cohesion — Minimize Coupling"?

*Maximize Cohesion - A function's purpose should be singular. (A function call should result in no surprises)
Minimize Coupling - A function should use local variables and avoid the use of global variables.*

9. Describe how functions can be overloaded.

By changing the number and type of parameters. (changing the function signature)

10. Given the following function pointers describe what type of function each can point to:

```
(void) (*fun_ptr) ();
(float) (*fun_ptr) (int, char*, float);
(char*) (*fun_ptr) (float, float);
```

*Pointer to function that returns void and takes no arguments.
Pointer to function that returns float and takes three arguments (int, char*, and float).
Pointer to function that returns char* and takes two floats as arguments.*

Chapter 10

1. (T/F) The typedef keyword lets you create a new data type that better represents your problem domain.

FALSE. The typedef keyword lets you create a type synonym (create a new name for an existing type) that can then be used to better represent your problem domain.

2. (T/F) Enumerations are new data types.

TRUE.

3. What type of operator would you use to access a structure object's data or function members?

The dot operator.

4. What type of operator would you use to access a structure object's data or function members via a pointer?

The -> operator

5. Describe how to use the “.” operator in conjunction with the “*” operator to access a structure object's data or functions members via a pointer.

*Dereference the pointer with the * operator and then access the object's elements via the dot operator. For example: (*pointer).memberFunction()*

6. List and discuss the three major differences between structures and classes.

*Structures have default public access — classes have default private access
Structures use the struct keyword — classes use the class keyword
Structures represent the C way of thinking — classes represent the C++ way of thinking*

7. Describe the code changes that were necessary to convert the struct version of Person to the class version.

The keyword struct was changed to class.

8. Why do you suppose the default access for a class is private?

To support the notion of data encapsulation.

9. Define the term data encapsulation.

The act of declaring class attributes private and access them via public interface functions.

10. What is meant by the term interface?

The set of public interface functions used to access class functionality.

Chapter 11

1. What is the primary difference between primitive data type objects and class objects?

Class type objects will play an expanded role in software systems. (They will perhaps interact with other class type objects in the system)

2. List the four special class member functions and describe the function of each. Which three of the four special functions can be overloaded?

Constructor (can be overloaded), destructor (cannot be overloaded), copy constructor (can be overloaded), and copy assignment operator (can be overloaded)

3. List the three access specifiers. Describe how each access specifier affect horizontal access.

public, protected, and private. Public access allows horizontal member access; protected and private do not.

4. In your own words define the term horizontal access.

The ability of a client program the access the interface of member elements of an object.

5. What is the difference between accessor and mutator functions?

*Accessors return the value of an attribute but do not alter the attribute value leaving the object state unchanged.
Mutators will change the attribute value which results in an object state change.*

6. What is the purpose of the this pointer?

The this pointer is a special pointer that contains the address of the current object.

7. What is the difference between a static class-wide variable and an instance variable?

A static, class-wide variable is available to all class object, whereas an instance variable is specific to a particular object. (All objects share static, class-wide variables but have their very own copies of instance variables)

8. Describe how member functions can be overloaded.

Member functions are overloaded like regular functions.

9. List and discuss two benefits of separating class interface from class definition. Can you think of any other benefits?

*To hide implementation details.
To be able to distribute class interface specifications as header files and definitions as libraries.*

10. How would you access an instance variable masked by a local function variable of the same name?

By using the this pointer.

Chapter 12

1. A class built from other class types is referred to as an _____.

Aggregation

2. List the two types of aggregation.

Simple and composite

3. Discuss each of the types of aggregation you listed above in terms of what role part objects play in each.

In simple aggregation part objects exist outside the aggregation whereas a composite aggregate creates and destroys its part objects.

4. In this type of aggregation, part objects belong solely to the whole or containing class. Name the type of aggregation.

Composite aggregate

5. In this type of aggregation, part object lifetimes are not controlled by the whole or containing class. Name the type of aggregation.

Simple aggregate

6. What does the line drawn between classes in a UML class diagram denote?

An association

7. What type of aggregation does a solid diamond indicate when attached to one end of an association line?

Composite aggregation

8. What type of aggregation does a hollow diamond indicate when attached to one end of an association line?

Simple aggregation

9. In a UML class diagram, the aggregation diamond is drawn closest to which class, the whole or the part?

The whole or aggregate class

10. What is the purpose of a UML sequence diagram?

To graphically represent object message sequencing.

Chapter 13

1. Describe the differences between horizontal and vertical access. Describe how to use the keywords public, protected, and private to control horizontal and vertical access.

Horizontal access is the access enjoyed by a client program to an object's members. The public access specifier allows horizontal access but protected and private prevent horizontal access. Vertical access is the access a derived class object has to base class object members. The access specifiers are used to specify what type of horizontal access inherited members will have in the derived class. (public inheritance, protected inheritance, or private inheritance)

2. When can you get away with forward declaring a class name vs. including the whole header file?

When you only need the name of the class and not the names of member functions or attributes.

3. Describe, in your own words, the purpose of inheritance. In what two primary ways is inheritance used?

Inheritance is used to extend the functionality of a class. Inheritance is used to evolve software design and to facilitate code reuse by having common class behavior located in base classes and inherited by derived classes.

4. Describe how to hide a base class function with a derived class function.

The base class function is non-virtual and the derived class function has the same name function signature as the base class function.

5. How do you override a base class function in a derived class?

The base class function is virtual and the derived class function has the same signature as the base class function. The overriding function is accessed via a base class pointer to a derived class object. (dynamic polymorphic behavior)

6. What is a virtual function?

A virtual function is one that may have an overriding implementation in a derived class

7. What is a pure virtual function?

A pure virtual function has no implementation in the class in which it's declared and must be implemented somewhere down the inheritance chain.

8. What is the purpose of a virtual destructor?

To ensure the destruction of a derived class object via a base class pointer.

9. Describe how to achieve dynamic polymorphic behavior.

Declare a base class pointer and assign it the address of a derived class object. Call a base class interface function and the derived class version will be called instead.

10. In what part of a derived class constructor do you call the base class constructor?

Initializer list

Chapter 14

1. In your own words describe the goal of operator overloading.

To tailor the behavior of operators in the context of your programs.

2. Describe the difference between a shallow copy and a deep copy.

A deep copy will take into account pointers and the objects they point to whereas a shallow copy will not.

3. What type of copy does a default, compiler-provided assignment operator perform?

Shallow

4. Describe how operator overloading is related to function overloading.

Operator overloading is function overloading!

5. (T/F) Operators overloaded in base classes can be overridden in derived classes.

TRUE

6. What is the purpose of the dummy integer parameter in the postfix version of the increment and decrement operators?

The dummy parameter is used by the compiler to distinguish between the two different versions of the operator.

7. What level of access does a friend function of a class enjoy?

The same level of access as an ordinary member function.

8. How would non-member operator functions gain access to private class data members?

Declare it to be a friend of the class.

9. When overloading arithmetic operators, to what type of error checking should you pay particular attention?

That the result of the operation does not exceed the capacity of the return type.

10. What is the ultimate result and benefit of operator overloading?

Cleaner, easier to read code.

Chapter 15

1. In your own words explain what static polymorphism is and how it is achieved in C++ using function and class templates.

Static polymorphism happens at compile time when a parameterized function, structure or class is used to specify a new function or class type.

2. In your own words give a definition of a template.

A template is a generic specification of a function, structure, or class that can be used to create a specific function, structure, or class type.

3. How are function templates related to overloaded function?

Function templates can often be used in place of function overloading because one generic function can be declared and defined to work on multiple parameter data types.

4. How are places reserved in source code for type substitution?

By using type placeholders.

5. How many type placeholders can a function or class template declare?

As many as are required.

6. If the number of type placeholders declared in a function template is greater than the number of function parameters, how can the type of extra placeholders be specified when the function is called?

Using the template specialization syntax.

7. Why is it a good idea to group the declaration and definition of a class or function template in one header file?

To ensure compatibility.

8. Is it necessary to group the declaration and definition of a class or function template in one header file?

It depends on the compiler.

9. What is the purpose of an STL container adapter?

STL container adapters use STL containers to implement their functionality.

10. What are iterators and how are they related to pointers? What is the purpose of an iterator? What benefits do you gain from using iterators to manipulate container elements vs. other iterative methods.

Iterators provide a uniform way to manipulate elements in a container component regardless of the container type.

Chapter 16

1. Describe in your own words what is meant by the term dynamic polymorphism.

Dynamic polymorphism mean being able to treat different types of objects as if they were the same type using a uniform interface to those objects. A derived class object is substituted for and used where base class objects are specified.

2. How does dynamic polymorphism differ from ad hoc and static polymorphism?

Ad hoc polymorphism is operator overloading, meaning operators are given special functionality relevant to the context of a design.

Static polymorphism is achieved by writing generic code that then can be used for different purposes based on the context of your design.

Dynamic polymorphism enables you to substitute different implementation objects at runtime.

3. What is a pure virtual function?

A virtual function is a function that can be overridden by a derived class function having the same signature.

4. How is a pure virtual function different from an ordinary virtual function?

A pure virtual function has no implementation in the class or struct in which it is declared and must be overridden eventually in a derived class.

5. A base class that declares one or more pure virtual functions is known as what type of class?

Abstract base class

6. What happens to a derived class if it inherits a pure virtual function but fails to override it?

It becomes an abstract class.

7. Describe the three different inheritance behaviors achieved through the use of pure virtual, ordinary virtual, and non-virtual functions.

A class that declares pure virtual functions is serving as an interface specification only and provides no behavior. A derived class can either declare and implement the function or defer the implementation to yet another derived class down the inheritance chain.

An ordinary virtual function provides some sort of behavior that will either be accepted by the derived class but can be overridden if desired.

Non-virtual functions provide behavior and cannot be overridden in a derived class. However, they can be hidden by a redeclaration of a derived class member function with the same name.

8. Consider the relationship between a base class and a derived class. What type of behavior should the base class implement as compared to the derived class's behavior?

The base class should implement behavior that is common to all derived classes.

9. How does public inheritance differ from private inheritance?

Public inheritance means the derived class implements an "is a" relationship.

Private inheritance means there is no conceptual relationship between the base class and derived class since all public functions of the base class are now private in the derived class.

10. What happens to public base class functions when a derived class privately inherits them?

They become private in the derived class.

Chapter 17

1. Describe in your own words why it is important to have well-behaved objects in an object-oriented software application.

So you can accurately predict the behavior of objects in your program. This will lead to more reliable code.

2. List and describe the four basic object usage contexts.

*Object creation
Object copying
Object assignment
Object destruction
other object contexts by design,,,*

3. What is the orthodox canonical class form?

A recipe for writing well-behaved objects.

4. What four special functions support the OCCF?

*Default constructor
Copy constructor
Destructor
Copy assignment operator*

5. Why is it important to be aware of all the contexts an object will participate?

So your object behave predictably when they are used in a program.

6. List several examples of object context participation that have not been discussed in this chapter.

*Comparing objects in a sorting program
Comparing contents of text message objects
etc....*

7. List several examples of why it would be a mistake to rely on a compiler generated constructor and destructor.

The default constructor and destructor will not properly initialize complex object. For instance, a default constructor will not automatically initialize a pointer, and the default destructor will not destroy the pointer either.

8. What is the definition of a default constructor?

A default constructor has default values for all parameters. It can therefore be called with no arguments (setting all object attributes to default values) or with arguments as required. A default constructor generally alleviates the need to overload the constructor.

9. How does a copy constructor differ from a copy assignment operator?

A constructor creates a new object.

A copy constructor creates a new object using an existing object as a guide.

10. What is the purpose of a destructor?

To tear down the object in memory and release any system resources back to the operating system.

Chapter 18

1. (T/F) C enables programmers to overload function names.

FALSE

2. What is the purpose of the extern linkage specification?

To prevent the C++ compiler from name mangling a C (or other) language function declaration.

3. What is name mangling and why do C++ compilers employ this technique?

All identifier names in memory must be unique. In order to overload functions C++ must mangle the names of functions.

4. List and describe the steps required to create a C function library.

***Step 1** - Put the function declarations for any functions you want in the library in a header file. (.h file)*

***Step 2** - Put the definitions for the library functions in a separate implementation file. (.c file)*

***Step 3** - Create an empty project in your Integrated Development Environment and add the implementation file to it.*

***Step 4** - Add any library files to the project required to support the implementation file.*

***Step 5** - Set the required target settings for the project.*

***Step 6** - Name the library output file and set project type.*

***Step 7** - Compile the project.*

***Step 8** - Use the library!*

5. List and describe the four issues to consider before using assembly language in your programs.

***Efficient Processor Usage** - Old assembly language routines may not use modern processors efficiently. As a rule, new processor versions introduce new instructions. Old versions of assembly language will most certainly not support these new instructions.*

***Code Optimization** - Face it...compilers generate better optimized code than do humans. If your reasons for using assembly include the one "I can do it better..." I recommend letting the compiler have a go at it first.*

***Portability** - Adding assembly to your C++ program will lock it into a specific processor and severely limit your code portability.*

***Maintenance** - As time passes, assembly routines embedded directly in your C++ code will have to be upgraded more frequently than the C++ instructions to take advantage of new processor features.*

6. What C++ keyword allows you to embed assembly language in the body of C++ functions?

asm (but if and how you can embed assembly language is compiler dependent, check with your compiler documentation)

7. (T/F) All C++ compilers implement the capability to include assembly language in the body of functions the exact same way.

FALSE

8. List and describe the steps required to create, compile, and link an assembly language module to a C++ program.

*Step 1 - Create assembly language file with a text editor and save it with the .asm extension,
Step 2 - Assemble the file to create a Portable Executable/Common Object File Format (PE/COFF) object file,
Step 3 - Create a C++ project using the development system of your choice,
Step 4 - Create a header file that declares the name of the function contained in the object file. The function declaration must be declared to have extern "C" language linkage.
Step 5 - Add the object file to the C++ project,
Step 6 - Compile and run the project and do a victory dance!*

9. What is the purpose of the javah command line tool?

To automatically generate header files for native functions from the Java class specification.

10. List and describe the steps required to call C++ functions from a Java program using the Java Native Interface (JNI).

*Step 1 - Create a Java source file that declares a class with one or more native methods. In addition to any native methods this class requires it also must also load the native dynamic library module using the System.loadLibrary() function.
Step 2 - Compile the Java source file to create a .class file.
Step 3 - Use the javah compiler with the -jni switch to automatically create a header file for use in your C++ program.
Step 4 - Create a C++ source file that implements the native method.
Step 5 - Compile the C++ source file to create a dynamic library that exports the native method.
Step 6 - Run the Java program using the java virtual machine and do a victory dance!*

11. Under what circumstances would it not be a good idea to call a native C++ function from a Java program?

If the native function did something small and needed to be called often.

Chapter 19

1. List and describe the preferred characteristics of an object-oriented architecture.

*Easy to understand - Programmers can get their heads around the design with minimal effort
Easy to reason about - It's easy to predict the effects of change
Easy to extend - It's obvious how to make enhancements to the system.*

2. State the definition of the Liskov substitution principle.

Subtype objects must be behaviorally substitutable for supertype objects. Programmers must be able to reason correctly about and rely upon the behavior of subtypes using only the supertype behavior specification.

3. Define the term class invariant.

A condition associated with a particular class or structure property that must always hold true in all valid object states.

4. What is the purpose of a function precondition?

A function precondition must hold true before the function can be guaranteed to work properly.

5. What is the purpose of a function postcondition?

A function postcondition must preserve the class invariant.

6. List and describe the three rules of the substitution principle.

***Signature rule** - a subtype must support all functions published by the supertype*

***Methods rule** - calls to overriding functions should behave like the base class functions they override.*

***Properties rule** - Subclasses must preserve any properties that are provable about the base class*

7. Write the definition and goals of the open-closed principle.

Software modules must be designed and implemented in a manner that opens them for extension but closes them for modification.

8. Explain how the open-closed principle uses the Liskov substitution principle and Meyer design by contract programming to achieve its goals.

Code written with the OCP in mind depends upon behavior promised by abstract base class specification. Depending upon promised behavior enables the subtype reasoning process. The LSP/DbC is used to achieve the proper subtype behavior within a type hierarchy thereby enabling the OCP

9. Write the definition and goals of the dependency inversion principle.

A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

10. Explain how the dependency inversion principle builds upon the open-closed principle and the Liskov substitution principle/Meyer design by contract programming.

The OCP and the LSP/DbC, when applied together, result in the realization of a third design principle known as the Dependency Inversion Principle (DIP). The key to the DIP is that high-level software modules should not rely on low-level details, and that software modules at all hierarchy levels should rely only upon abstractions. When a software architecture achieves the goals of the DIP it is easier to extend and maintain (flexible and non-rigid). Software modules that conform to the DIP are easier to reuse in other contexts (mobile)

Chapter 20

1. What is the purpose of a UML modeling tool?

The purpose of a UML modeling tool is to assist the object-oriented analyst, designer, or programmer in the analysis, design and implementation of their object-oriented software system. A modeling tool also plays a critical role in communicating software designs to other analysts, designers, programmers, managers, and customers. A good UML modeling tool is therefore both a design and communication tool at the same time.

2. In what two primary ways does a UML modeling tool help designers?

Helps them design systems and helps them communicate the system design to themselves and others.

3. Why do you suppose it would be important for a UML modeling tool to support collaborative design and development.

Because large software systems are designed by more than one person.

4. What is meant by the term reverse engineering as it applies to UML modeling tools?

The ability to generate system diagrams from existing source code.

5. Why do you think web reporting is an important UML modeling tool feature?

A web reporting tool will let you communicate your system design to others who do not have access to the UML modeling tool you are using.

6. What is the purpose of a use case diagram?

To illustrate high-level system functionality from the point of view of system actors.

7. What is the purpose of a class diagram?

To illustrate application classes and their static relationship to other application classes.

8. What is the purpose of a sequence diagram?

To illustrate the sequence of object message passing for a particular thread of execution.

9. What piece of a design should one sequence diagram document?

A sequence diagram should concentrate on one particular thread of execution, otherwise it will become too cluttered.

10. (T/F) Describe™ can link diagram elements to outside documents and URLs.

TRUE

INDEX

Symbols

- ! 114
- 114
- 114
- != 115, 121
- #define 13, 27, 207
- #endif 13, 27, 207
- #ifndef 13, 27, 207
- #include 13, 27, 103
- % 114
- %= 115
- & 114, 115, 122
- & operator 172, 174
- && 115
- &= 115
- () 114
- (type) 114
- * 114
- * operator 172, 174
- *= 115
- + 114
- ++ 114
- += 115
- . 114
- . * 114
- / 114
- /= 115
- < 114
- <> 103
- << 114
- <<= 115
- <= 115
- = 115
- = 106, 115
- == 115, 121
- > 114
- > 115
- > operator 256
- >* 114
- >= 115
- >> 114
- >>= 115
- ? : 115
- [] 114
- ^ 115, 122
- ^= 115
- | 115, 122
- |= 115
- ~ 114

Numerics

- 64 bit processor 104

A

- abstract base class 266, 420, 434
- abstract base classes 328, 416, 417
- abstract class 343
 - definition of 343
- abstract classes 343
- abstract data types 167
- abstract machine
 - targeting 83
- abstract thinking 7, 47
- abstract thought 84
- abstraction 83, 416
 - problem 7
- abstractions
 - getting them right 416
 - selecting the right kinds of 501
- access
 - member
 - horizontal 274
- access specifier
 - private 260, 274
 - protected 274
 - public 259, 274
- access specifiers 274, 328, 332, 362
 - private 289
 - protected 289
 - public 289
- accessing member elements via pointers 256
- accessor 374
- accessor functions 287, 331
- ad hoc polymorphism 372
- Ada 36
- adapters 406
- addition operator 117
- additive operators 117
- aggregation 302, 322
 - relationship to object lifetime 302
 - simple 305
 - simple and composite 302
 - simple vs. composite 302
 - two forms 322
- alert 109
- algorithm
 - growth rate 93
- algorithms 80, 91, 412
- American National Standards Institute (ANSI) 104
- analysis 50
- application
 - graceful recovery 48
- architecture
 - extension via inheritance 328
 - improving with three design principles 482
- arguments
 - passing to a function by reference 219
 - passing to a function by value 216
- array
 - accessing elements 186
 - addressing formula 185
 - arrays of pointers 188
 - automatic initialization of multi-dimensional 193
 - brace map 194
 - combining declaration and definition 187
 - danger of uninitialized elements 187
 - declaration
 - four-dimensional array 192
 - two-dimensional array 190
 - declaring 185
 - declaring two-dimensional 189
 - definition of 184
 - dimensional grouping at declaration 191
 - dynamic array declaration 196
 - dynamically allocated multi-dimensional 197
 - element addressing 184
 - locating array elements 185
 - memory representation of two-dimensional array 191
 - multi-dimensional 189
 - of function pointers 236
 - passing as argument to function

- 222
 - passing multi-dimensional to function 223
 - pointer arithmetic 187
 - representation in memory 184
 - single dimension
 - declaration of 185
 - strings 200
 - subscript method of element accessing 186
 - three and more dimensions 191
 - two-dimensional 189
 - types they can contain 185
 - use of braces 194
 - visualizing multi-dimensional 189
 - What is an array? 184
 - array elements 63
 - array processing 48
 - arrays
 - intro to 184
 - using constants to set array size 186
 - assembler 83
 - assembly 83
 - assembly language 83, 458
 - adding object module to C++ project 469
 - assembling 467
 - creating source file 467
 - inline assembly in Macintosh environment 469
 - inline in C++ function 465
 - issues to consider before using 465
 - linking object file 467
 - linking object modules with C++ 465
 - steps to create object file 467
 - to machine code translation 83
 - assignment operator 378, 394
 - assignment operators 124
 - associating an identifier with a memory location 168
 - asterisk
 - overloaded use of 173
 - placement in pointer declaration 172
 - attribute candidates 49
 - attributes 47, 48
 - auto increment and decrement operators 125
 - auxiliary storage devices 81
 - avoiding break and continue 158
- B**
- backslash 109
 - backspace 109
 - bad programming habits
 - avoiding 100
 - bad software architecture
 - characteristics of 500
 - base class 266, 328, 331, 362
 - base class attributes
 - initialized from derived class 337
 - base class constructor 328
 - base class constructor call 362
 - base class constructors
 - calling from derived classes 335
 - base class pointer 420
 - base class pointers 416
 - base classes
 - preventing multiple instances of 360
 - behavior 328
 - defining common in base classes 328
 - predictable object 446
 - Bertrand Meyer 482, 497
 - Bertrand Meyer's Design by Contract (DbC) 483
 - Bertrand Meyer's Design by Contract programming 482
 - binary 26
 - binary instructions 26
 - bit 89
 - bitwise AND operator 121
 - bitwise exclusive OR operator 122
 - bitwise inclusive OR operator 122
 - block statement 143
 - bool 104
 - boolean literals 111
 - branch link register 101
 - break 157
 - buses 81
 - byte 90
 - byte addressable memory 90
- C**
- C 36
 - C library function header 459
 - C library function implementation file 459
 - C library routines
 - adding files to CodeWarrior project 460
 - CodeWarrior project target settings 461
 - compiling CodeWarrior project 462
 - using 462
 - C vs. C++
 - important difference 458
 - C++ 36
 - power and danger of 484
 - C++ & Java 470
 - C++ class construct 274
 - C++ implementation dependencies 465
 - C++ Man 167
 - C++ preprocessor 27
 - C++ program
 - smallest 100
 - C++ standard 19
 - C++ Standard Template Library 412
 - C.A.R. Hoare 233
 - cache 83
 - cache memory 88
 - callback function 237
 - callback functions 237
 - called routine 27
 - calling conventions 27
 - calling routine 27, 101
 - carriage return 109
 - case label 150
 - cfloat.h 104
 - challenges 4
 - char 104
 - character 104
 - character literals 108
 - cin 134
 - CISC 83
 - CIWS class 345
 - class 266, 274
 - constants 275
 - declaration 330
 - declaring new data types 274
 - class attributes 278
 - class behavior 322
 - class constants
 - initializing via initializer list 280
 - class declaration
 - body 275
 - don't forget the semicolon! 275
 - how to name classes 274
 - minimum 275
 - opening and closing braces 275
 - parts of 274
 - placement of public functions 275
 - class declarations
 - viewed as behavior specifications 484
 - class diagram 274, 276, 506
 - compartments 276
 - indicating private class members 276
 - indicating public class members 276
 - class diagrams
 - adding attributes in Describe™ 517
 - adding operations in Describe™ 517
 - creating in Describe™ 515
 - Describe™ properties editor window 518
 - class invariant 484, 486
 - defined 484

- class invariants 484
 - class member functions 276
 - class template 398
 - class type 362
 - classes 263
 - difference between classes and structs 265
 - introduction to 263
 - polymorphic 417
 - static relationship modeling 506
 - limits.h 104
 - code reuse 328
 - CodeBuilder 36
 - CodeWarrior
 - disassembly tool 101
 - COFF library file format 467
 - cohesion 19
 - comma operator 125
 - comment clutter 17
 - comments 103
 - bad use of 17
 - c++-style 17
 - creating in source code 103
 - C-style 103
 - c-style 16, 17
 - self commenting code 17
 - source code 16
 - communication devices 81
 - compiler 26, 27
 - compiler errors
 - caused by missing class declaration semicolon 275
 - fixing 11
 - complex application behavior 302
 - complexity
 - conceptual 416
 - managing physical 302
 - physical
 - on larger projects 309
 - complexity management 293
 - composite aggregate
 - definition 302
 - composite aggregation 322
 - example 303
 - compound assignment operators 125
 - compound statements 143
 - computer
 - definition of 80
 - general purpose machine 80
 - vs. computer system 80
 - computer architecture 83
 - computer program 250
 - computer system 80
 - bus 81
 - definition of 80
 - interface components 81
 - main logic board 81
 - main logic board block diagram 81
 - system unit 81
 - typical 80
 - computers 80
 - conceptual complexity 416
 - concrete class 343
 - concrete classes 434
 - conditional operator 123
 - constant 49, 100
 - constants 103, 128
 - declaration 100
 - naming convention 18
 - constructor 275, 278, 331
 - initializer list 279
 - purpose 279
 - constructors 274
 - containers 406, 412
 - continue 158
 - control flow 142
 - control statement usage guide 160
 - copy
 - deep 379
 - deep vs. shallow 379
 - shallow 379
 - copy assignment operator 282
 - purpose 283
 - purpose of 449
 - copy constructor 281
 - example code 281
 - purpose 281
 - purpose of 449
 - coupling 19
 - cout 135
 - Cplusplusified C code 482
 - creating objects 167
 - creating objects via definition 168
 - C-style comment 103
 - current position 50
 - CVS 507
- ## D
- dangling reference 229
 - data 250
 - data encapsulation 267, 274, 289
 - data modeling 250
 - data structure 49
 - data structures 48
 - data types
 - determining size with sizeof 106
 - DbC 483
 - debugger 27
 - decimal 107
 - decimal integer literal 106
 - declaration statement 142
 - declaring
 - overloaded functions 231
 - decode 88
 - deep copy 379
 - default constructor 278, 279
 - definition 279
 - purpose of 449
 - delete 114
 - delete operator 176
 - dependency inversion principle 482, 500
 - dependency inversion principle (DIP) 328
 - derived class 266, 328, 362
 - derived class initializer list 328
 - derived class object 362
 - derived class objects 416
 - Describe™ 276, 506
 - feature set 507
 - Design by Contract 483
 - destructor 275, 283, 331
 - purpose 284
 - purpose of 449
 - virtual 330
 - destructors 274
 - effects of non-virtual destructors
 - 341
 - virtual 340, 341
 - development cycle
 - applying 45
 - code 45
 - deploying 45
 - factor 45
 - integrate 45
 - plan 45
 - test 45
 - using 45
 - direction 49
 - disassemble 101
 - disassembled code listings 101
 - disassembly 101
 - division operator 116
 - DLL 293
 - creating Mac OSX DLL with GNU C++ compiler 476
 - do statement 155
 - document-as-you-go 511
 - doing something forever 152
 - DOORS™ 507
 - dot operator 255
 - double 105
 - double quote 109
 - Dr. Barbara Liskov 483
 - Dr. Bertrand Meyer 483
 - Droning Professor 166
 - dumb sort 92
 - dynamic linked library 293
 - dynamic object creation 174, 420
 - dynamic polymorphic behavior 362
 - dynamic polymorphism 416
 - complex example 422
 - defined 417
 - example 419
 - dynamically allocated resources 379

E

- Eiffel 483
- electronic pathways 81
- Emacs 37
- Embarcadero Technologies 507
- empty project
 - creating in CodeWarrior 459
- encapsulation 267
- enum 250, 252, 254
 - default state values 252
 - state name conflicts 253
- enumerated data types 252
- enumerated types 250, 252
- enums 252
- equal To operator 121
- equality operators 121
- error checking 48, 70
- error handling
 - trapping bad input 134
- errors
 - compiler 11
- escape sequence table 109
- escape sequences 109
 - using in strings 111
- essentiality 416
- ethernet 81
- evolving software architecture 458
- Example 392
- executable image 26
- executable program 84
- execute 88
- EXIT_SUCCESS 133
- explicit template specialization 451
- expression forms
 - additive operators 112
 - assignment operators 113
 - bitwise AND operator 113
 - bitwise exclusive OR operator 113
 - bitwise inclusive OR operator 113
 - comma operator 113
 - conditional operator 113
 - constant expressions 113
 - equality operators 113
 - explicit type conversion 112
 - logical AND operator 113
 - logical OR operator 113
 - multiplicative operators 112
 - pointer-to-member operators 112
 - postfix 112
 - class member access 112
 - const cast 112
 - decrement 112
 - dynamic cast 112
 - explicit type conversion 112
 - function call 112
 - increment 112
 - pseudo destructor call 112
 - reinterpret cast 112
 - static cast 112
 - subscripting 112
 - type identification 112
- primary 112
- relational operators 113
- shift operators 113
- unary 112
 - decrement 112
 - delete 112
 - increment 112
 - new 112
 - sizeof 112

- expressions 103, 111
 - composition 111
 - table of expression forms 112
- expression-statements 142
- extended mnemonic 101
- extern “C” language linkage specification 478
- extern keyword 458

F

- false 111
- fetch 87, 88
- file scope 130
- firewire 81
- fleet simulation 343
- float 105
- floating point 104
- floating point arithmetic 83
- floating point literals 110
- floor 50
- flow 9
 - achieving 9
 - concept of 9
 - stages 9
- for statement 155
- force multipliers
 - abstraction and inheritance 416
- form feed 109
- Fortran 36
- frustrations 4
- function
 - array of pointers to 236
 - callback via function pointer 237
 - called function responsibilities 216
 - calling 209
 - calling function responsibilities 216
 - characteristics of a well written 208
 - code module 206
 - declaration 209
 - declaring 208
 - default argument values 227
 - defining 208
 - definition 209
 - definition of 206

- definitions 207
- description of 206
- function parameter scope 214
- grouping related statements into 206
- hiding 337
- hiding global variables 212
- implementation files 207
- interface vs. implementation 207
- local variable scoping 212
- member function 206
- naming 208
- overloading 231, 328, 337
- overriding 328, 420
- passing arguments by reference 219
- passing arguments to 215
- passing arrays as arguments 222
- passing multi-dimensional arrays to 223
- passing pointers as arguments 220
- passing references 221
- placing declarations in header files 207
- pointer
 - syntax 235
- pointers 234
- purpose of 206
- recursion 232
- recursive calling 232
- return keyword usage 226
- return values 225
- returning objects 226
- returning pointers 227
- returning reference 229
- returning void 225
- signature 231
- static variables 213
- use of scoping blocks 212
- using return values 225

- function access rights
 - as preconditions 496
- function argument types
 - as preconditions 494
- function declaration in header file 207
- function libraries 207
- function library 239
 - steps to creating 239
- function overloading 328
- function overriding 416
- function parameter scope 214
- function parameters 212
- function pointer
 - array of 236
 - assigning the address of a function to 235
 - calling function via 236
- function pointers 234, 235, 237
- function return types

- as postconditions 496
- function return values 225
- function scope 130
- function signature 231, 340, 362
- function signatures 290
- function stubbing 11, 56
- function template 398
- functions 48, 230
 - accessor 274
 - class members 276
 - declaring pure virtual 342
 - declaring virtual 340
 - introduction to 206
 - main() 100
 - member functions 206
 - multiple with same name 231
 - mutator 274
 - naming convention 18
 - overloading 231, 290, 372
 - protected 275
 - pure virtual 342
 - purpose of pure virtual 342
 - virtual 340
- fundamental data types 250
- fundamental language features 48
- fundamental types 104
 - determining value ranges 105
 - range of values 104
 - value range table 105

G

- G4 82
- G4 7400 82
- generalized component behavior 434
- generic class definition 398
- generic code 398
- generic function 398
- global variable
 - hiding with local variables 212
- global variables 131
- GNU C++ compiler 476
- GNU C++ compiler command 38
- good software architecture
 - characteristics of 501
- goto statement 159
- greater than operator 120
- greater than or equal to operator 121
- growth rate 93

H

- hardest thing about learning to program 4
- hardware 83
- header file 14, 207
 - structure of 207
 - typical 207
- header files
 - contents of 14

- heap 175
- heat sink 82
- heterogeneous objects 184
- hexadecimal 107
- hexadecimal escape sequences 110
- hiding outer block variables 213
- high-level program modules 52
- homogeneous objects 184
- horizontal access 330
 - concept 289
 - controlling via access specifiers 288
- horizontal member access 274
- horizontal tab 109
- HP 36
- hungarian notation 126

I

- IComponent class 422
- IDE 27
 - creating new project 28
 - creating project with CodeWarrior 28
 - project 28
 - project stationary 28
 - tracking project changes 35
- identifier
 - naming conventions 126
- identifiers 126
- if statement 143
- if statements and compound statements 144
- if-else statement 145
- IMDE 506
- implementation file
 - contents of 15
- incremental code evolution 361
- inheritance 266, 331, 332, 362, 416
 - benefits 328
 - effects of using different types 332
 - multiple 353
 - private 333
 - protected 333
 - public 333
 - purpose and use 328
 - uses of 361
 - virtual 357
- inheritance hierarchy 422
- inheriting 328
- initializer list 274
 - definition 279
 - example code 279
- initializer lists 274
 - line wrapping for readability 280
- input error checking 134
- input/output 83
- insertion operator 375
- instance attributes 278

- instruction 83
- instruction decode 87
- instruction execution 87
- instruction execution stages 82
- instructions 26, 250
- int 105
- integer 104
- integer literals 107
- Integrated Development Environment 27
- Integrated Modeling and Development Environment 506
- integration
 - robot rat project 60
- interface 266, 311, 434
 - benefits of separating 12
 - separating from implementation 302
 - separation from implementation 293
 - separation of 12
- interface components 81
- interface vs. implementation 207
- internal hard disk drives 81
- invoking a method on an object 267
- iostream 54, 103, 374
- iostream.h 103
- iteration statements 150
- iterators 412

J

- Java 36
 - compiling Java source file 472
 - creating Java source file 471
 - steps to create JNI C++ program 470
 - using javah command line tool 473
- Java & C++ 470
- Java Native Interface
 - creating C++ Win32 DLL 474
 - creating header file 472
 - Mac OSX example 475
 - Win32 JNI example 471
- Java Native Interface (JNI) 458, 470
- javah command line tool 473

K

- keyboard 80, 81
- keywords 104
 - listing 104
 - reserved identifiers 104
- Knuth 233

L

- labeled statements 159
- language features 44, 50
 - strategy area check-off list 51

- language linkage specification 458
 - left shift operator 118
 - legacy application code 458
 - legacy C code 458
 - using 458
 - less than operator 120
 - less than or equal to operator 121
 - libraries 103
 - adding to programs 103
 - library
 - function
 - creating 239
 - library code written in C 458
 - library routines
 - building in C 458
 - steps to creation in C 459
 - limits.h 104
 - link register 101
 - linker 26, 27
 - Linux 36
 - Liskov Substitution Principle 482
 - relationship to Meyer Design by Contract Programming 483
 - three rules of 497
 - Liskov Substitution Principle (LSP) 328, 483
 - literal
 - hexadecimal 107
 - octal 107
 - literals 106
 - boolean 111
 - characters 108
 - decimal 107
 - floating point 110
 - dissected 110
 - parts of 110
 - integer 107
 - multiple characters 108
 - single characters 108
 - strings 111
 - using escape sequences in character literals 109
 - load immediate 101
 - local function variables 212
 - local variable 212
 - local variables 212
 - declaring 212
 - logical AND operator 123
 - logical OR operator 123
 - long double 105
 - long int 105
 - long long int 105
 - looping
 - forever 152
 - lower-level modules 52
 - LSP 483
 - LSP & DbC
 - C++ support for 483
 - common goals 483
 - designing with 483
 - lvalue vs. rvalue 124
- ## M
- machine code 26, 83
 - machine instructions 26
 - MachTen 36
 - Macintosh OSX 476
 - Macintosh™ 100, 506
 - main file
 - main() function 15
 - main logic board 81
 - main memory 81
 - main() function 100, 103, 132
 - calling functions upon exit 133
 - disassembled 100
 - EXIT_SUCCESS 133
 - exiting 133
 - purpose 132
 - returning integer value 100
 - two forms 132
 - makefile 37, 38
 - contents of 38
 - creating 38
 - dependencies 38
 - for firstprog 38
 - managing physical complexity 293
 - member access via this pointer 278
 - member function 206
 - member function access 278
 - member function overloading 274
 - member functions
 - access to class attributes 278
 - memory 81
 - addressability 90
 - alignment 90
 - arrangement 168
 - boundaries 90
 - cache 89
 - miss performance penalty 89
 - contiguous allocation for array 184
 - non-volatile 88
 - stack and heap 175
 - sub systems 89
 - volatile 88
 - memory address of objects 169
 - memory addressing 169
 - memory hierarchy 89
 - memory organization 88
 - memory region
 - size occupied by an object 168
 - menu 49
 - Mercury Test Director™ 507
 - methods rule 497
 - Metrowerks CodeWarrior 28, 401, 459
 - adding files to projects 30
 - Groups 29
 - projects 29
 - running projects 31
 - setting project search paths 29
 - Sources group 29
 - microphones 81
 - microprocessor 82
 - Microsoft Visual C++ 32, 507
 - creating new workspace 32
 - StdAfx.h 32
 - Workspace 32
 - millions of instructions per second 82
 - minimal C++ program 100
 - minimal development environment 27
 - MIPS 82
 - mnemonic
 - extended 101
 - model 47
 - modeling 47
 - modem 81
 - modulus operator 117
 - monitor 80
 - mouse 80, 81
 - multi-file projects 12
 - multi-file variable usage 131
 - multiple character literals 108
 - multiple file project 48
 - multiple file projects 48
 - multiple inheritance 353
 - multiplication operator 116
 - multiplicative operators 116
 - mutator 374
 - mutator functions 287
- ## N
- name mangling 458
 - name spaces 253
 - name table 101
 - naming
 - functions 208
 - naming convention
 - sticking with 19
 - naming conventions
 - adopting 19
 - concepts of 17
 - constants 18
 - functions 18
 - hungarian notation 126
 - variables 18
 - nested while loop 153
 - nesting for statements 157
 - nesting if-else statements 146
 - nesting while statements 152
 - new 114
 - new class types 328
 - new line 109
 - new operator 175
 - NeXT 36
 - non-static class attribute 278

- non-virtual inheritance 360
 - not an lvalue... 124
 - not equal to operator 121
 - noun 49
 - nouns 48, 49
 - mapping to data structures 49
 - null statements 142
 - null termination of strings 111
 - numeric_limits 105, 106
 - how to use 105
 - numeric_limits<> 106
- O**
- object 266, 274
 - accessing via pointer 173
 - addressing 169
 - behavior 446
 - definition 167
 - dynamic creation using new operator 174
 - state and behavior 276
 - three ways to create 167
 - object behavior 276
 - object code 26
 - object creation 167
 - object interaction 84
 - Object Management Group 506
 - object modules 27
 - object state 276
 - Objective-C 36
 - object-oriented analysis & design 267
 - object-oriented architecture
 - extending 482
 - preferred characteristics 482
 - reasoning about 482
 - understanding 482
 - object-oriented design 7
 - reasoning about 361
 - object-oriented programming 416
 - C++ language feature support for 417
 - defined 417
 - object-oriented thinking 266, 274
 - ObjectPlant™ 276, 506
 - objects 331
 - accessing via address 169
 - assignment 447
 - copying 447
 - creation 446
 - destruction 447
 - lifetime control in composite aggregates 322
 - other design usage contexts 447
 - usage contexts 446
 - observable behavior 84
 - definition 84
 - OCCF 448
 - OCCF class functions 449
- OCP 498
 - defined 498
 - example 498
 - octal 107
 - octal escape sequences 110
 - ofstream 374
 - overloading 374
 - OMG 506
 - open-closed principle 482, 497
 - achieving 498
 - conventions 498
 - relationship to the LSP and DbC 498
 - open-closed principle (OCP) 328
 - operands 101
 - operating system 80, 100
 - operator 100, 176, 177, 184
 - insertion 375
 - operator * 186
 - operator overloading
 - cool things C++ programmers can do 372
 - goal 372
 - in the context of your design 372
 - operator precedence 114
 - operators 113
 - > 256
 - addition 117
 - additive 117
 - assignment 124
 - division 116
 - equal to 121
 - equality 121
 - greater than 120
 - greater than or equal to 121
 - left shift 118
 - less than 120
 - less than or equal to 121
 - modulus 117
 - multiplication 116
 - multiplicative 116
 - not equal to 121
 - overloading 372, 373
 - precedence table 114
 - right shift 119
 - shift 118
 - shorthand member access 256
 - subtraction 118
 - optimized routine 458
 - organizational complexity 54
 - orthodox canonical class form 448
 - extending 451
 - four required functions 448
 - ostream 374
 - overloadable operators 372
 - overloaded function
 - defined 231
 - return types 231
 - overloaded operator 394
- overloaded operators
 - leading to cleaner code 372
 - parameter list 373
 - overloading
 - arithmetic operators 383
 - assignment operator 378
 - comma operator 392
 - compound assignment operator 386
 - constructors 279
 - decrement operator 387
 - equality operator 381
 - function operator 392
 - increment operator 387
 - iostream operators 374
 - member operator 392
 - relational operators 380
 - subscript operator 384
 - virtual overloaded operators 392
 - overloading functions 290
 - overloading iostream operators 374
- P**
- parentheses 115
 - using to control operator precedence 115
 - part class 302
 - pass by value 216
 - pen 49
 - Perplexed One 166
 - pipeline 82
 - pipelined instructions 82
 - placeholders
 - in templates 398
 - placement of the asterisk 172
 - Plant class 343
 - pointer
 - accessing object via 173
 - array of 188
 - callback functions 237
 - declaration
 - use of asterisk 172
 - declaration of 172
 - definition of 171
 - dereferencing 173
 - how not to return from a function 229
 - using to access array elements 187
 - vs. reference 177
 - pointer arithmetic 187
 - pointer dereferencing 173
 - pointer vs. reference 177
 - pointers 166, 328, 340
 - base class to derived class objects 340
 - to functions 234
 - uses of 166
 - what are they good for? 166

- polymorphic class 340
 - polymorphic classes 417
 - polymorphism
 - ad hoc 372
 - defined 417
 - dynamic 416, 436
 - static 398, 402, 412
 - post condition
 - example 485
 - postcondition 486
 - defined 484
 - postconditions 484
 - changing in derived class functions 493
 - power supply 81
 - PowerDesigner™ 506
 - PowerPC 100
 - PowerPC G4 82
 - PowerPC mnemonic 101
 - PPC Std C++ Console Settings 101
 - precondition 486, 487
 - defined 484
 - example 485
 - preconditions 484
 - changing preconditions of derived class functions 488
 - weakening 489
 - predictable object behavior 446
 - preprocessor directive 103
 - preprocessor directives 13, 27
 - behavior of 14
 - previously developed assembly modules 458
 - private 328, 332, 334
 - private access specifier 260
 - problem abstraction 7, 84
 - problem domain 6, 44, 48
 - procedural design 7
 - procedural programming paradigm 49
 - processing cycle 87
 - processor 81, 82
 - program
 - definition of 83
 - two aspects 83
 - program creation process 26
 - program termination
 - indicating with result code 100
 - program transformation process 85
 - programming 4
 - programming as art 4
 - programming cycle 10
 - code 10
 - factor 10
 - integrate 10
 - plan 10
 - repeating 11
 - summarized 11
 - test 10
 - programming skills required 4
 - programs 80
 - project approach strategy 6
 - design 7
 - in a nutshell 7
 - language features 7
 - problem domain 6
 - requirements 6
 - strategy areas 6
 - using 44
 - project complexity
 - managing 11
 - project file format 14
 - project objectives 47
 - project requirements 6
 - project specification 49
 - project stationary 28
 - projects
 - multi-file 12
 - properties rule 497
 - protected 328, 332, 334
 - protected function 331
 - protected functions 275
 - protocol 101
 - pseudocode 69, 70
 - public 328, 332, 334
 - public access specifier 259
 - public interface functions 329
 - pure abstraction 416
 - pure virtual function 342, 420
 - pure virtual functions 328, 434
 - pure virtual member functions 416
 - purpose of a constructor 279
- Q**
- question mark 109
 - quicksort 233
- R**
- RAM 88
 - random access memory 88
 - Rational Rose™ 506
 - recursion 232
 - creating recursive functions 232
 - stopping 232
 - recursive function calls 232
 - recursive functions 232
 - Reduced Instruction Set Computer (RISC) 82
 - reference 177
 - declaration and use of 178
 - returning from function 229
 - region of storage 167
 - register usage 83
 - relational operators 120
 - reliable object-oriented software
 - creating 483
 - requirements 6, 44
 - gaining insight through pictures 49
 - requirements gathering 6
 - resolving C function library names 458
 - result code 100
 - result store 87
 - return 133
 - return keyword 226
 - reverse engineering
 - merging two systems after 528
 - using Describe™ 527
 - right shift operator 119
 - RISC 82
 - Robot Rat 507
 - robot rat
 - version two design issues 513
 - version two project specification 508
 - robot rat project specification 46
 - analyzing 47
- S**
- scope 128
 - creating scope blocks with {} 129
 - file 130
 - function 130
 - scope blocks 129
 - scope resolution operator 278
 - scoping blocks 212
 - scoping rules 212
 - Sedgewick 233
 - selection statements 142, 143
 - if 142
 - if-else 142
 - switch 142
 - semicolon
 - placement of 275
 - semicolon as sequence point 142
 - sending a message to an object 267
 - sentinel values 151
 - separating interface from implementation 293
 - sequence diagram 308
 - sequence diagrams
 - adding messages to 523
 - creating in Describe™ 522
 - sequence point 142
 - shallow copy 379
 - shift operators 118
 - short int 105
 - shorthand member access 255
 - shorthand member access operator 256
 - side 142
 - signature rule 497
 - signed char 104
 - signed int 105
 - signed long int 105
 - signed long long int 105
 - signed short int 105
 - simple aggregate

- definition 302
- simple aggregation 322
- simple escape sequences 109
- simplifying assumptions 309
- single quote 109
- sizeof 106, 114
- smallest C++ program 100
- software 83
- software development roles 5
 - analyst 5
 - architect 5
 - programmer 6
- software modules 458
- software systems 458
- source code
 - generating in Describe™ 525
- source code control systems 507
- speakers 80, 81
- special member functions 274
 - default behavior 286
 - four types 278
- special register 101
- stable application architecture 328
- stack 175
- stack and heap memory 175
- stack frame 212
- stages of execution 82
- standard C++ console application 54
- standard libraries 83
- Standard Template Library 405
- state transition diagram 65, 66
- statement 100
 - declaration statement 142
 - goto 159
 - termination with semicolon 142
- statements 103, 142
 - block statement 143
 - break 157
 - compound statement 143
 - continue 158
 - do 155
 - difference from while 155
 - nesting 155
 - examples of 142
 - for 155
 - nesting 157
 - written as a while statement 156
 - goto
 - usage advice 159
 - if 143
 - if-else 145
 - nesting 146
 - iteration 150
 - labeled 159
 - looping forever 152
 - null statement 142
 - selection statements 143
 - side effects of 142

- switch 147
 - break and the default case 150
 - case labels 147
 - default label 147
 - use of break keyword 148
- using compound statements 143
- while 150
 - exiting with break 152
 - exiting with exit() 153
 - nesting 152
 - using a switch statement 153
 - using sentinel value 151
- static function variables 213
- STL 405, 412
 - adapters 406
 - algorithms 408
 - containers 406
 - iterators 407
 - list 411
 - vector 408
- STL components 398
- store 88
- strategy
 - project approach 6
- stream operators
 - overloaded 376
- strengthening preconditions 491
- string
 - array 111
 - literals 111
 - null termination 111
- string literals 111
 - assigning to an array 111
- strings 200
- struct 254
- structs
 - accessing members via pointer 255
- structure
 - member definition format 262
 - template 398
- structures 254
 - accessing elements within 255
 - C++ Style 259
- stubbing 11, 56
- subclass 266, 328
- subtraction operator 118
- SUN 36
- superclass 266, 328
- superscalar 82, 83
- supertypes & subtypes
 - reasoning about 483
- switch statement 147
- switch statements 252
- system bus 81
- system software 80
- system unit 80, 81

T

- table of contents 101
- template
 - class 398
 - complex class example 404
 - definition of 398
 - function 398
 - member function definition 403
 - placeholders 398
- template declaration
 - grouping declaration and implementation 400
- templates
 - declaring class templates 403
 - how they work 398
 - special template definition syntax 402
 - structure 398
 - type placeholder 399
 - using multiple type placeholders 400
- Tenon Intersystems
 - MachTen CodeBuilder 36
- testing
 - robot rat project 55
- text books, references, & quick reference guides
 - difference between 19
- text editor 27
- the art of programming 4, 8
 - inspiration 8
 - money but no time 8
 - mood setting 9
 - time but no money 8
 - where not to start 8
 - your computer 8
- this pointer 262, 278
- tight spiral development 45
- TOC 101
- top-down functional decomposition 52
- trapping bad input 134
- trigraphs 85
- true 111
- type definition 250
- type synonym 250
 - reason for creating 250
- typedef 250, 254
 - example usage 250
- typedef keyword 250

U

- UML 302, 307, 506
 - class diagram 276, 322, 328
 - diagram object
 - linking in Describe™ 529
 - diagrams 507
 - linking diagrams in Describe™ 511

- modeling tool
 - purpose of 506
 - two important roles played by 506
 - sequence diagram 308, 322
 - web reports
 - creating with Describe™ 530
 - UML modeling tools 506
 - Unified Modeling Language 274, 506
 - UNIX 36
 - creating makefile 37
 - make utility 37
 - makefile 37
 - unsigned char 105
 - unsigned int 105
 - unsigned long int 105
 - unsigned long long int 105
 - unsigned short int 105
 - usage for two dimensions 190
 - use case diagrams
 - adding documentation to 511
 - creating in Describe™ 509
 - programmer perspective 512
 - robot rat use case diagram 510
 - using directive 103
 - utility software 80
- V**
- variable 49, 100
 - scoping 100
 - variable naming 17
 - variables 103, 128
 - declaring 128
 - function scope 128
 - global 19
 - limiting variable scope 131
 - local scope 128
 - naming 128
 - naming convention 18
 - restricting global 19
 - scope 128
 - sharing file scope variables across multiple files 131
 - verb phrases 48
 - verbs 48, 49
 - vertical tab 109
 - Vessel class 343
 - virtual destructor 420
 - virtual destructors 420
 - virtual function 267, 422
 - virtual functions 340, 362
 - virtual inheritance 360
 - virtual keyword 328
 - vocabulary
 - object-oriented 266
- W**
- wchar_t 105
 - Weapon class 343
 - well-behaved object
 - definition 446
 - What are pointers good for? 166
 - What is a function? 206
 - What is a pointer? 171
 - What is an object? 167
 - while statement 150
 - whole class 302
 - Windows™ 506
 - wireless local area network card 81
 - word 90
 - wrapping up a project 72
 - writing generic code 398

Supplemental CD-ROM

The C++ For Artists supplemental CD-ROM includes the complete source code examples used throughout the book, Metrowerks CodeWarrior projects, UML design tools and demos, and C++ compilers. It also includes the eBook/PDF edition of C++ For Artists and Adobe Acrobat Reader Version 6 for Windows and Macintosh.

SOURCE CODE

The complete source code examples are provided as stand-alone source files and with Metrowerks CodeWarrior projects. The source files are grouped by chapter and project. To compile the examples you should follow the steps for creating projects with your chosen IDE given in the text. If you have Metrowerks CodeWarrior 5.0 or later for the Macintosh you can convert the project files.

C++ COMPILER TOOLS

The CD-ROM includes two C++ compiler tools. Borland's free Turbo C++ command line tool suite for Windows and the Open Watcom Project's C++ compiler tool suite for Windows. Macintosh users who are looking for a free C++ compiler can download Apple's Macintosh Programmer's Workshop (MPW) tool suite.

UML DESIGN TOOLS AND DEMOS

Two UML design tools are included, one for Windows and one for Macintosh. For Windows, a fourteen-day demo of Embarcadero Technologies' Describe. For the Macintosh, the shareware edition of ObjectPlant. Both tools will reverse engineer and generate C++ and Java code. Describe features are detailed in chapter 20.

eBook/PDF Edition of C++ FOR ARTISTS

The eBook/PDF edition of C++ For Artists is included on the CD-ROM in two versions: One large PDF file of the entire book and several smaller PDF files for each individual part of the book. (Front matter, contents, chapters, index, etc.)

ORDER FORM

website: pulpfreepress.com

email: orders@pulfreepress.com

snail mail: Pulp Free Press
 7423 Brad Street
 Falls Church, VA 22042-3642

Please send the following books:

<i>Title/ISBN</i>	<i>Quantity</i>	<i>Price</i>
<i>Sub-Total</i>		
<i>Tax (If Applicable)</i>		
<i>Total (Does not include shipping)</i>		

Please send more information on the following: (Check all that apply)
 __ *Future Titles* __ *Speaking/Seminars* __ *Mailing Lists* __ *Consulting*

Name: _____

Address: _____

City: _____ State: _____ Zip _____

Telephone: _____

email: _____

Payment: __ Check __ Credit Card: __ Visa __ MasterCard __ American Express

Card number: _____

Name on card: _____ Expiration date: _____

MASTER THE COMPLEXITIES OF OBJECT-ORIENTED PROGRAMMING AND C++

C++ FOR ARTISTS: THE ART, PHILOSOPHY, AND SCIENCE OF OBJECT-ORIENTED PROGRAMMING TAKES A REFRESHING, AND SOMETIMES CONTROVERSIAL, APPROACH TO THE COMPLEX TOPIC OF OBJECT-ORIENTED PROGRAMMING AND THE C++ LANGUAGE. INTENDED AS BOTH A CLASSROOM AND REFERENCE TEXT, C++ FOR ARTISTS BREAKS ALL MOLDS BY BEING THE FIRST BOOK OF ITS KIND SPECIFICALLY DESIGNED TO HELP READERS TAP THEIR CREATIVE ENERGY TO UNDERSTAND AND APPLY DIFFICULT PROGRAMMING CONCEPTS. C++ FOR ARTISTS WILL HELP YOU SMASH THROUGH THE BARRIERS PREVENTING YOU FROM MASTERING THE COMPLEXITIES OF OBJECT ORIENTED PROGRAMMING IN C++. START TAMING COMPLEXITY NOW! READ C++ FOR ARTISTS TODAY – ENERGIZE YOUR PROGRAMMING SKILLS FOR LIFE!

SUPERCHARGE YOUR CREATIVE ENERGY BY RECOGNIZING AND UTILIZING THE POWER OF THE “FLOW”, QUICKLY MASTER MULTI-FILE PROGRAMMING TECHNIQUES TO HELP TAME PROJECT COMPLEXITY, UTILIZE THE PROJECT APPROACH STRATEGY TO MAINTAIN PROGRAMMING PROJECT MOMENTUM, USE THE STUDENT SURVIVAL GUIDE TO HELP TACKLE ANY PROJECT THROWN AT YOU, LEARN A DEVELOPMENT CYCLE YOU CAN ACTUALLY USE AT WORK, APPLY REAL-WORLD PROGRAMMING TECHNIQUES TO PRODUCE PROFESSIONAL CODE, MASTER THE COMPLEXITIES OF AD-HOC, STATIC, AND DYNAMIC POLYMORPHISM, LEARN HOW TO CALL C AND C++ ROUTINES FROM JAVA PROGRAMS USING THE JAVA NATIVE INTERFACE (JNI), LEARN HOW TO INCORPORATE ASSEMBLY LANGUAGE ROUTINES IN YOUR C++ CODE, MASTER THREE OBJECT-ORIENTED DESIGN PRINCIPLES THAT WILL GREATLY IMPROVE YOUR SOFTWARE ARCHITECTURES, PAINLESSLY CONQUER POINTERS AND REFERENCES WITH THE HELP OF C++ MAN®, PACKED WITH OVER 43 TABLES, 216 ILLUSTRATIONS, AND 415 CODE EXAMPLES, REINFORCE YOUR LEARNING WITH THE HELP OF CHAPTER LEARNING OBJECTIVES, SKILL-BUILDING EXERCISES, SUGGESTED PROJECTS, AND SELF-TEST QUESTIONS. ALL CODE EXAMPLES WERE COMPILED AND EXECUTED BEFORE BEING USED IN THE BOOK TO GUARANTEE QUALITY, ACCOMPANYING CD CONTAINS EBOOK/PDF EDITION OF THE TEXT, SOURCE CODE FILES AND Codewarrior™ PROJECTS, PLUS UML MODELING TOOLS FOR WINDOWS™ AND MACINTOSH™, AND MUCH, MUCH, MORE...

“Finally, a writer who understands the complexities associated with the programming language learning process!”

“I wish I had this book when I was in school!” It’s clear, concise, and packed full of great information that’s hard to find conveniently located between two covers.”

“C++ FOR ARTISTS IS AWESOME! IT’S A MUST-READ FOR ANYONE TIRED OF GETTING THE SAME OLD s#!@ IN A PROGRAMMING TEXT.”

Rick Miller is a SENIOR SOFTWARE SYSTEMS ENGINEER AND Web Applications Architect for Science Applications International Corporation (SAIC), and Assistant Professor at Northern Virginia Community College, Annandale Campus, where he teaches C++ and Java programming classes.

“Read C++ For Artists Today –
ENERGIZE YOUR PROGRAMMING skills
for life!”

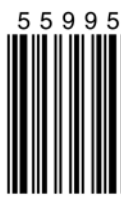


Pulp FREE PRESS

ISBN 1-932504-02-8



9 781932 504026



\$59.95