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. *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.*

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.

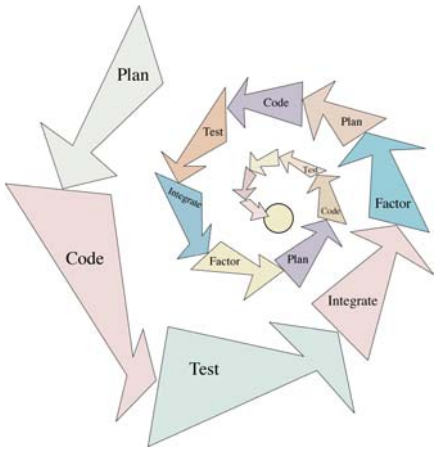

Figure 3-1: Tight Spiral Development Cycle Deployment

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

```
   Objectives:
   Demonstrate your ability to utilize the following language features:
   Arrays
   Program flow control structures
   Variables
   Constants
   Functions
   Simple iostream input and output
   Enumerated types
   Preprocessing directives
   Demonstrate your ability to create multi-file projects.

   Task:
   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.

   Hints:
   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.)

   At minimum, provide a text-based command menu with the following or similar
command choices:

                    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

IST 156
Project 1
Robot Rat

```
When the menu choice 6 is selected to print the floor, it might look something
like this, assuming you chose '*' to represent a marked area of the floor and '0'
to represent and unmarked area. You may use another pattern if desired.
.
                                        *****000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000
                                        00000000000000000000


In this example the robot rat has moved from the upper left hand corner of the
floor five spaces to the east with the pen down.
```

Table 3-3: Project Specification

Analyzing The Project Specification

Let us now step through the project approach strategy and analyze the robot rat project using each strategy area as a guide starting with the project's requirements.

Requirements

The robot rat project seems to be written clearly enough. It begins with a set of formally stated project objectives and then states the task you are to execute, namely, produce a program that lets you control the robot rat. But just what is a robot rat? That's a fair question and is one that gets asked often when I assign this project. To understand the answer requires you to think abstractly, and abstract thinking gives novice programmers the most trouble.

If I answered the question by saying, "Well, obviously, the robot rat does not really exist!", I would be insulting you. Why? Because if you are in fact wondering just what is the robot rat, then you are having difficulty abstracting the concept of the robot rat. I would be doing you a better service by saying, "The robot rat exists, but only as a collection of attributes that provide a limited description of the robot rat." I would also add that by writing a program to control the movement of the robot rat around the floor you are actually modeling the concept of a robot rat. And since a model of something usually leaves out some level of detail, or contains some simplifying assumptions, I will also tell you that the robot rat does not have legs, fur, or a cute little nose. I will talk more about the characteristics of a robot rat when I discuss the design strategy area below.

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.

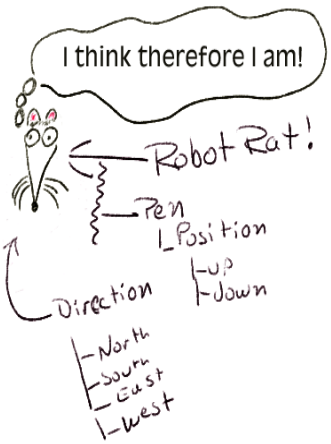

Figure 3-2: Robot Rat Viewed As Attributes

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.

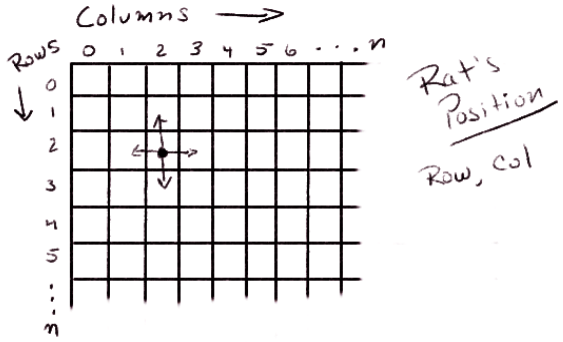

Figure 3-3: Robot Rat Floor Sketch

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.

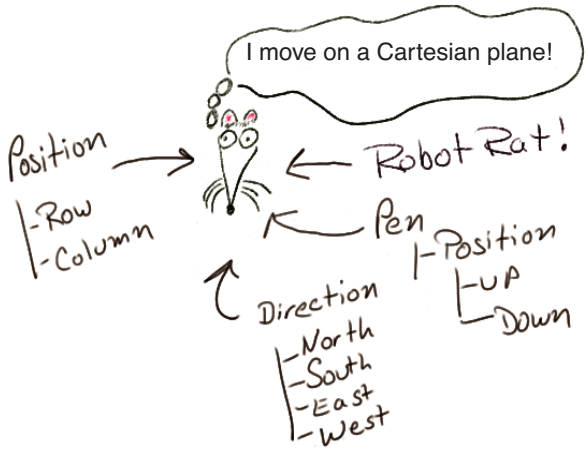

Figure 3-4: Complete Robot Rat Attributes

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.

3.1 Robot Rat Pseudocode

```
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
```

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.

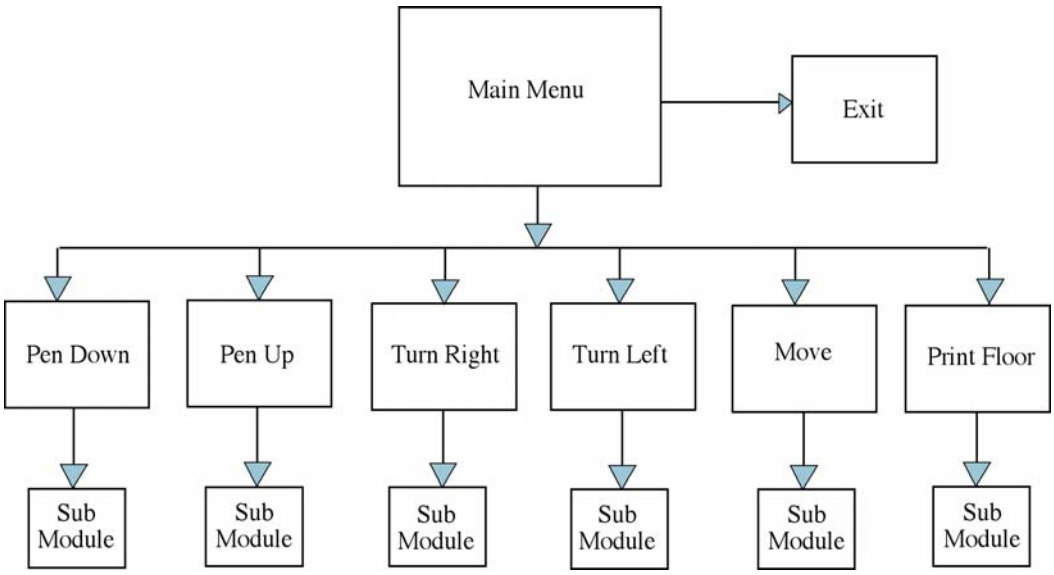

Figure 3-5: Functional Decomposition of Robot Rat Program

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™.

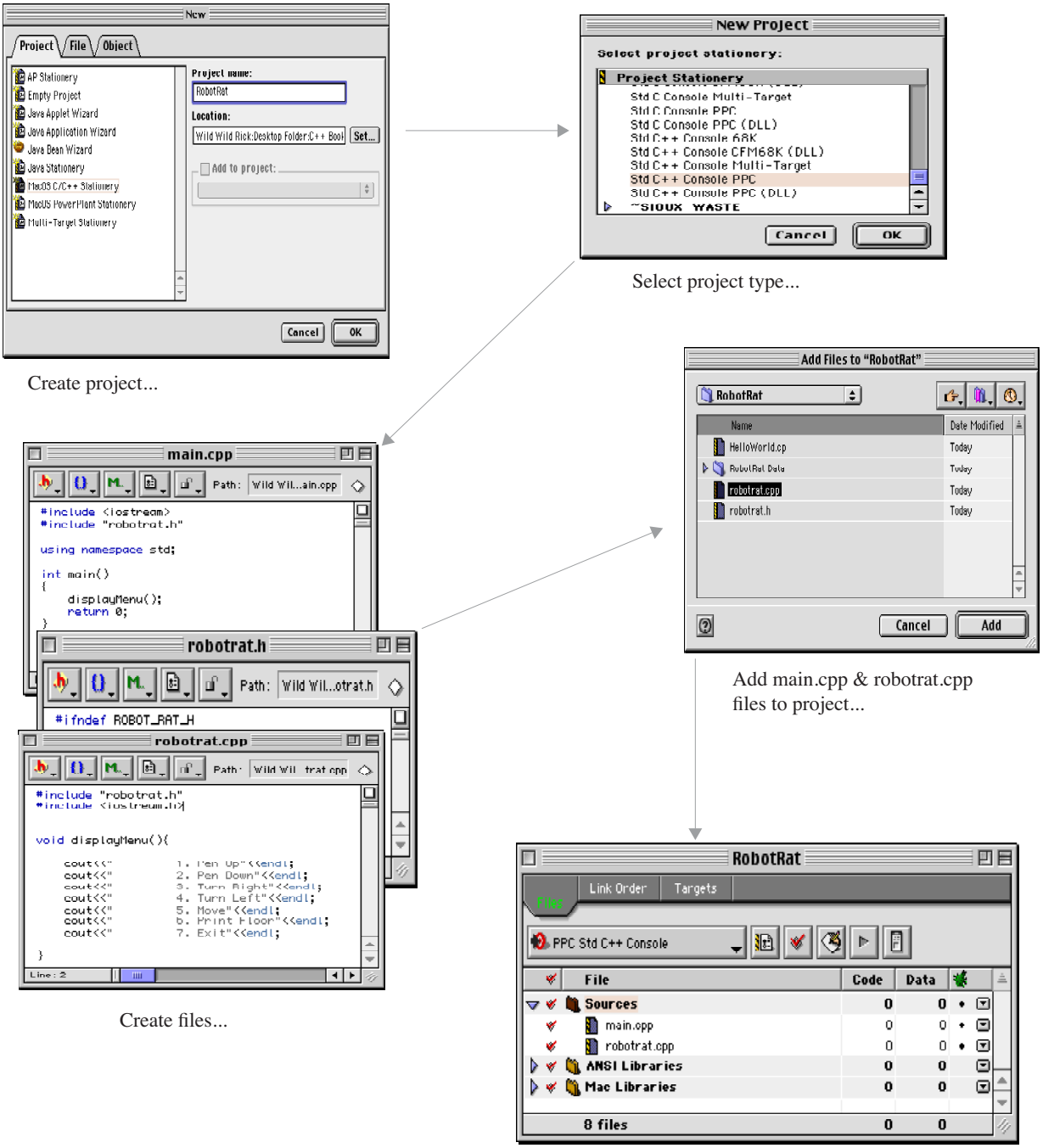

Figure 3-6: Overview of Project Creation Process

First, create the RobotRat project and select the desired project type. Since the project requires only simple iostream 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.

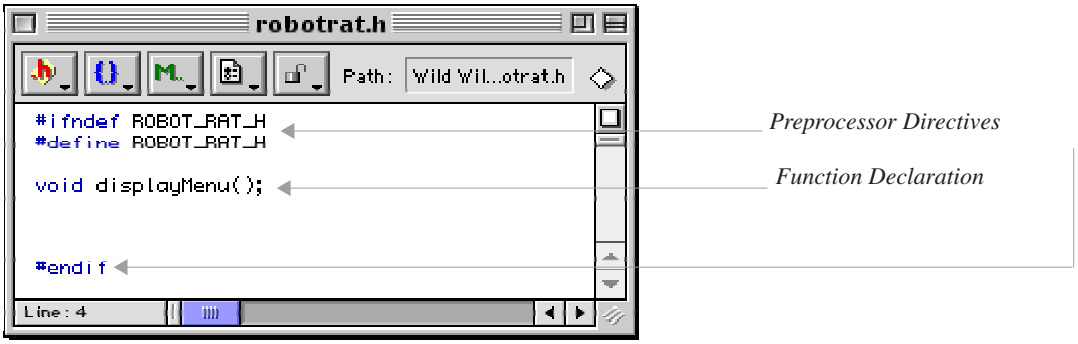

Figure 3-7: robotrat.h

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.

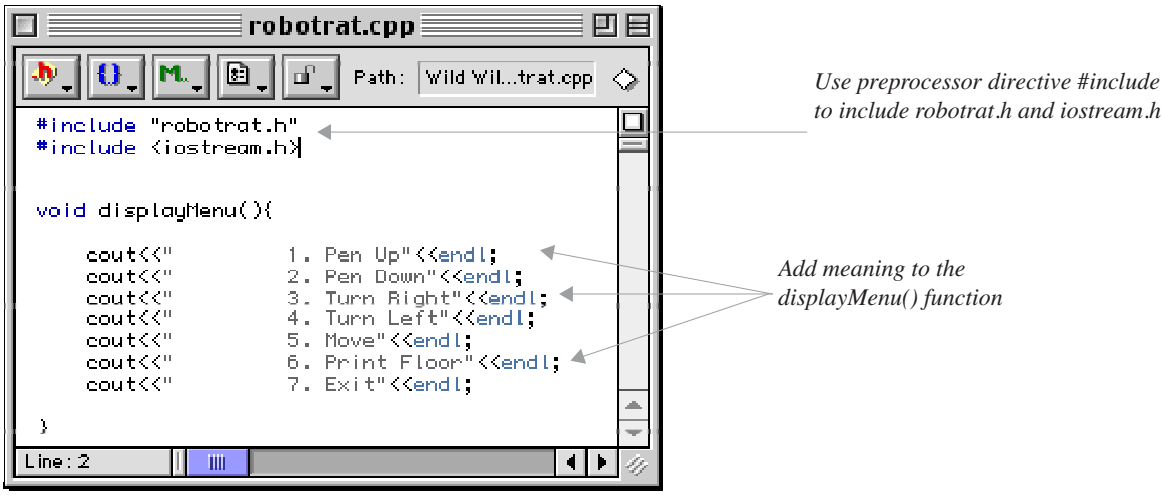

Figure 3-8: robotrat.cpp

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.

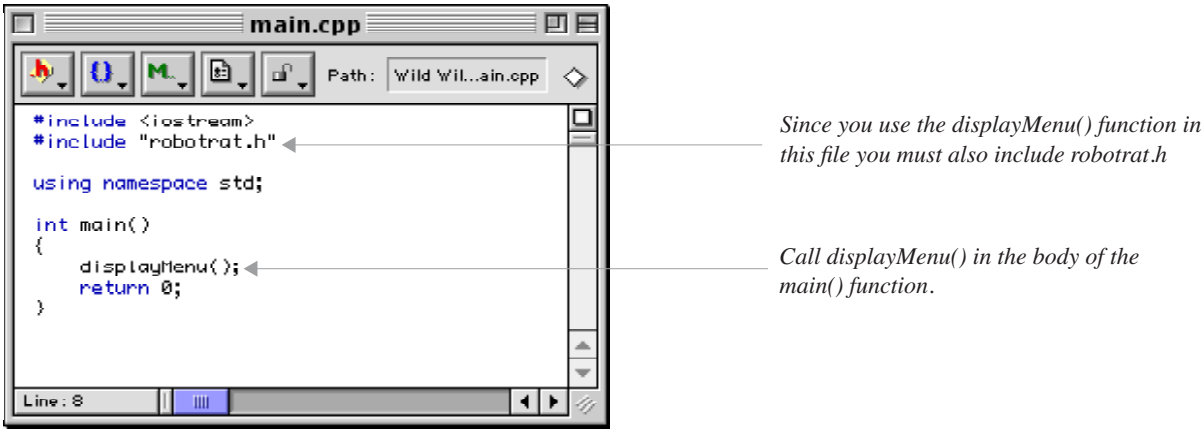

Figure 3-9: main.cpp

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.

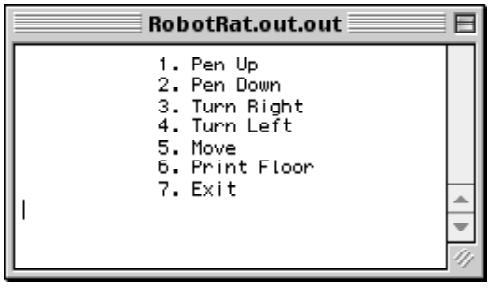

Figure 3-10: Robot Rat Menu

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

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.

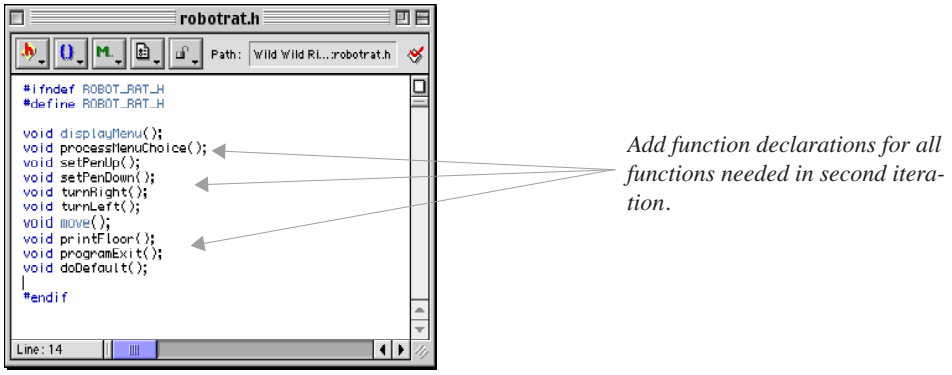

Figure 3-11: robotrat.h

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.

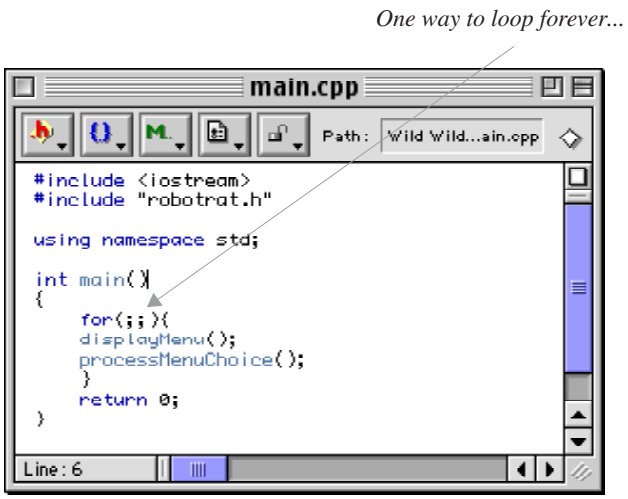

Figure 3-12: main.cpp

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.

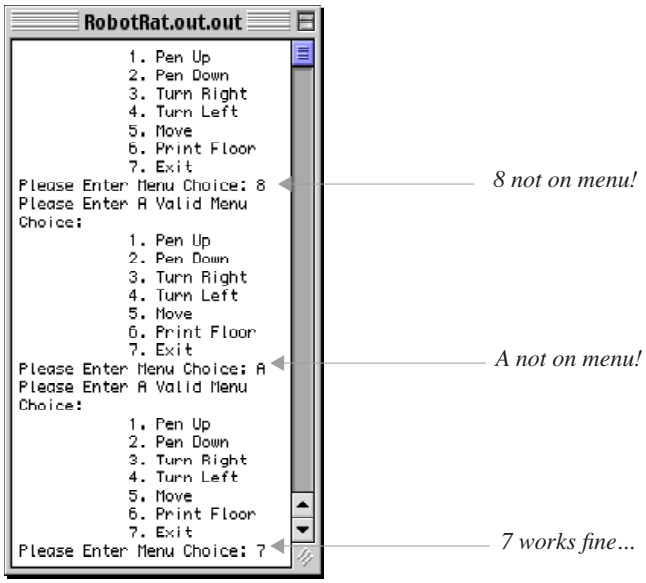

Figure 3-14: Default Case Test

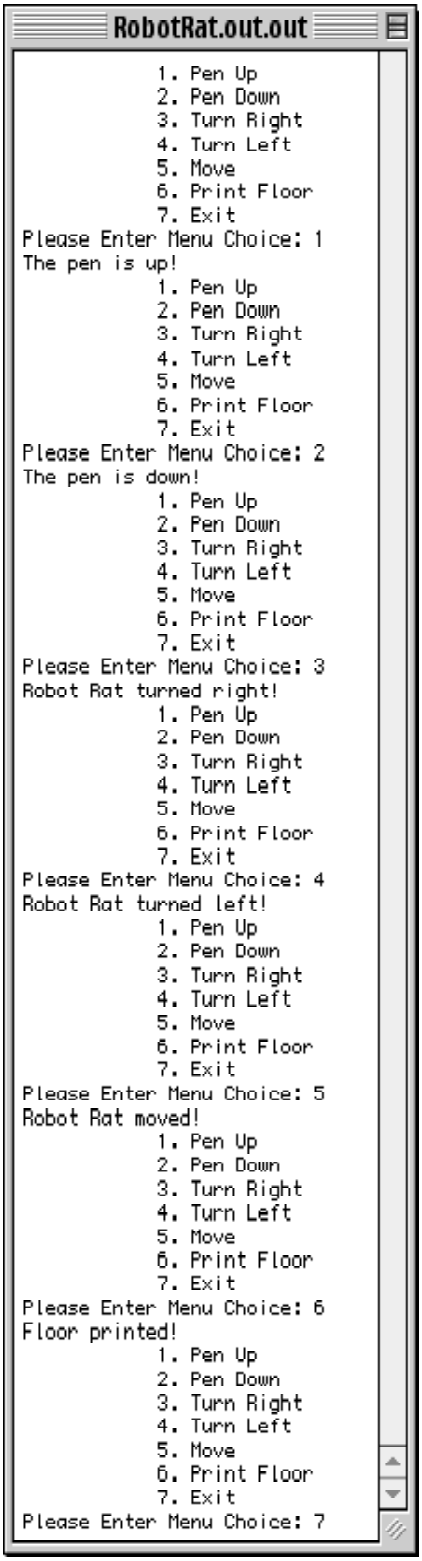

Figure 3-13: Test Results

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.

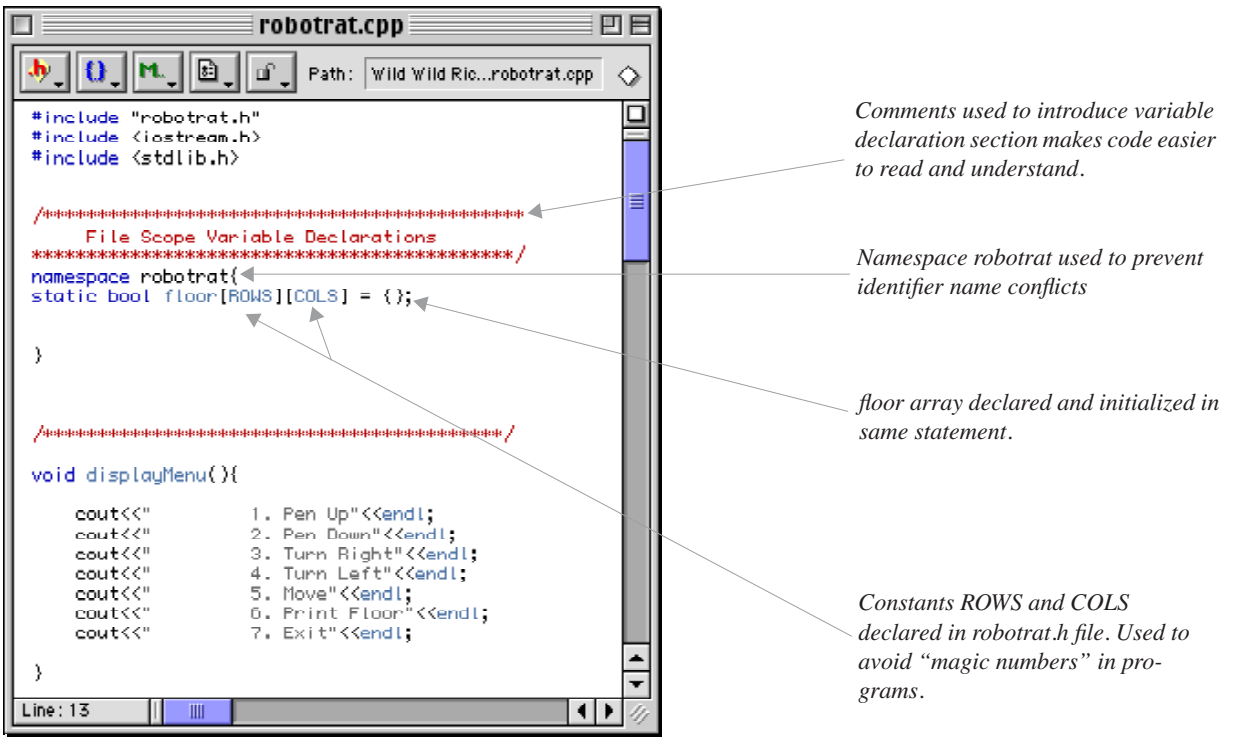

Figure 3-15: robotrat.cpp with Floor Array Declaration

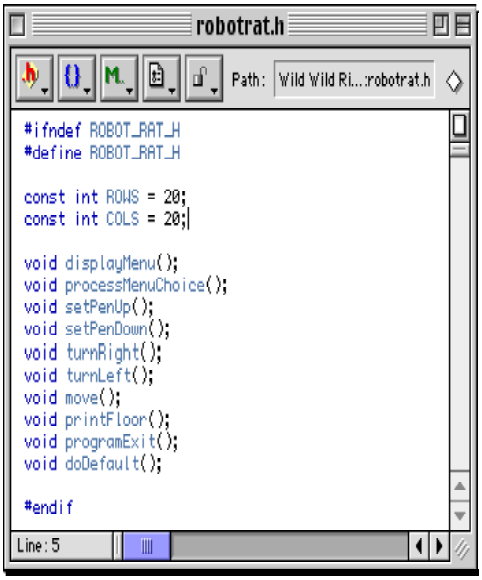

Figure 3-16: robotrat.h with ROWS & COLS Constants Declared

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.

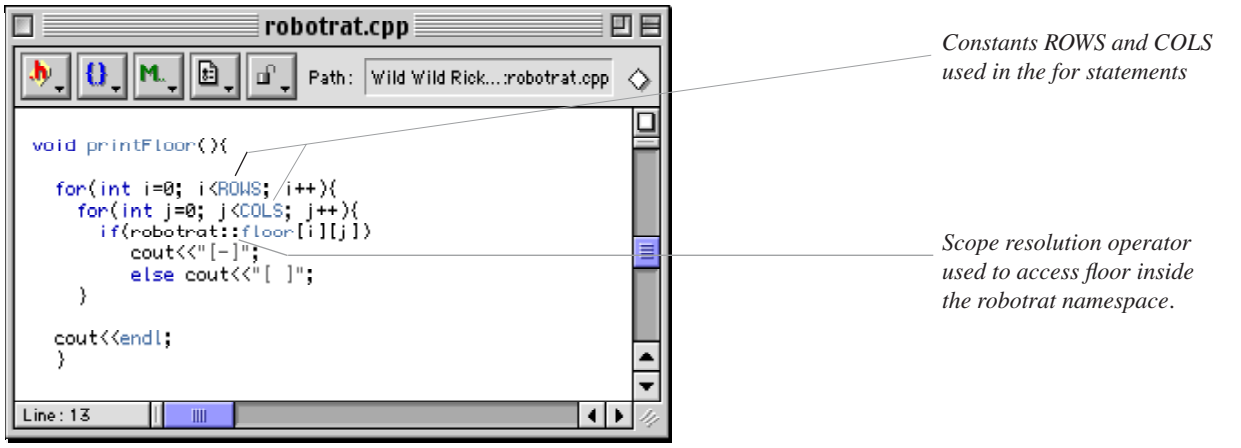

Figure 3-17: The printFloor() Function

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.

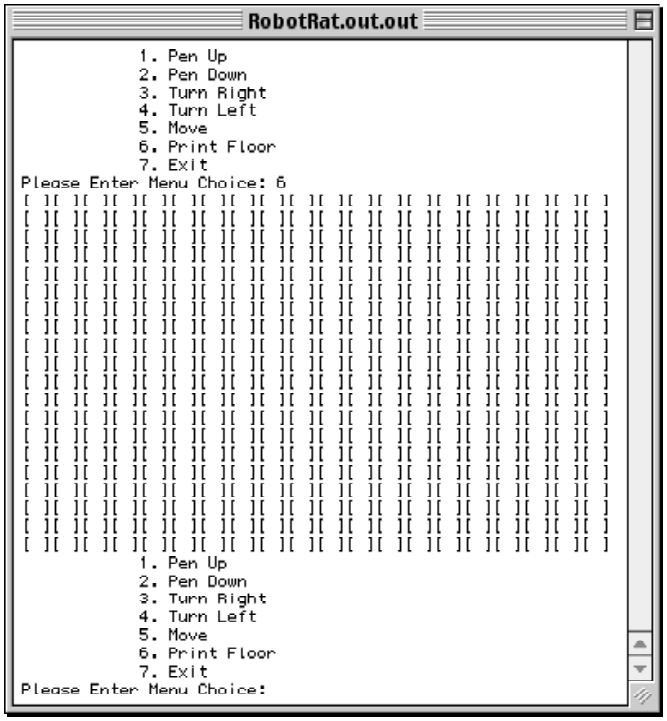

Figure 3-18: Robot Rat printFloor() Function Test

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.

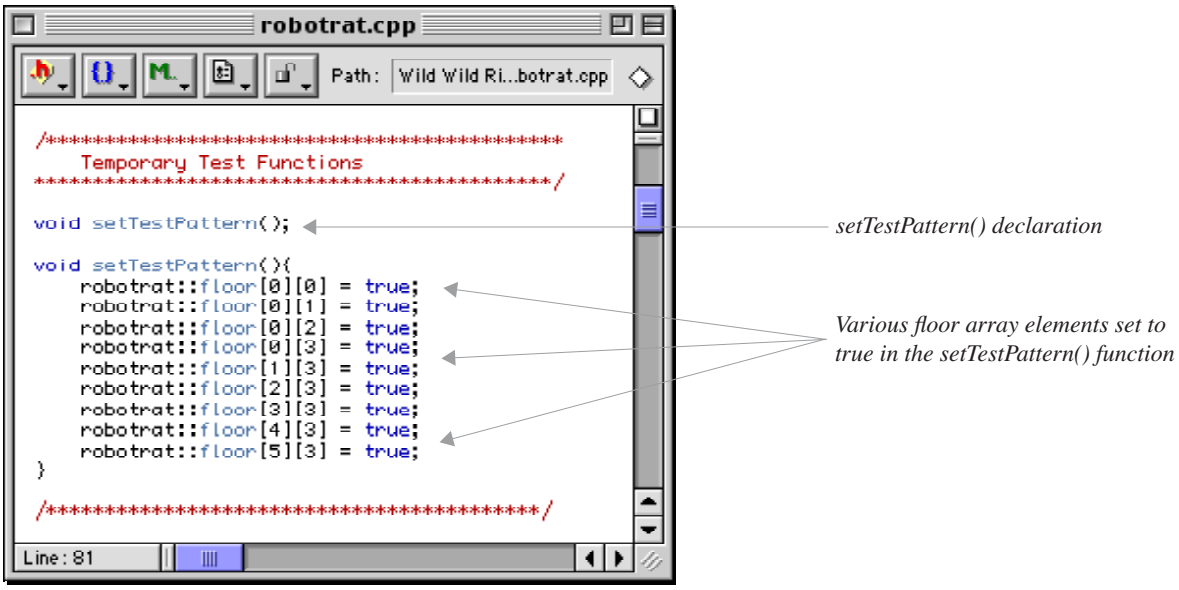

Figure 3-19: setTestPattern() Function

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().

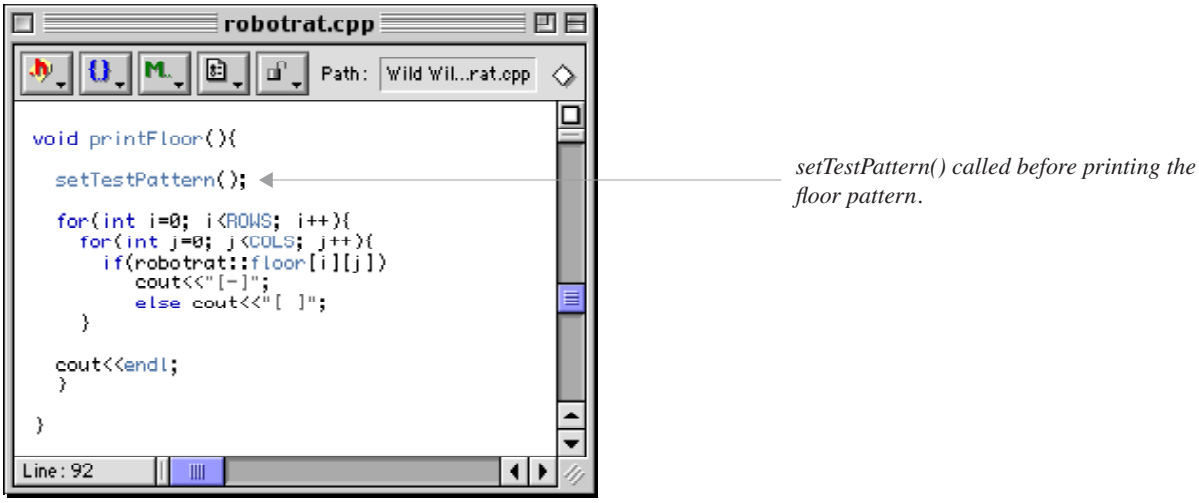

Figure 3-20: setTestPattern() Function Being Used for Testing in the printFloor() Function.

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.

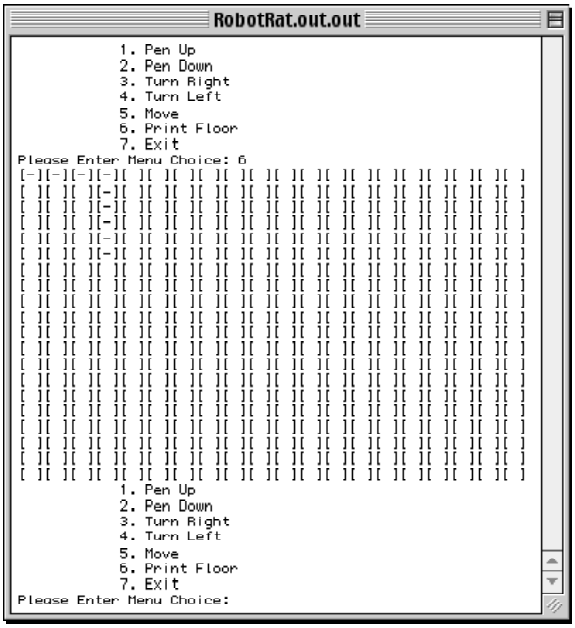

Figure 3-21: Robot Rat printFloor() Test with Test Pattern

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.

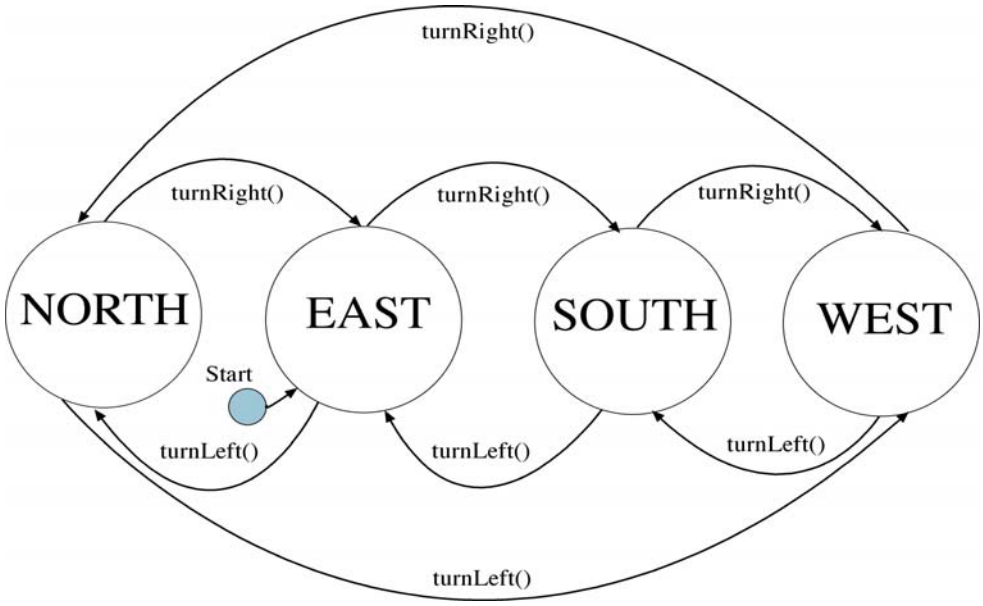

Figure 3-22: State Transition Diagram for rats_direction Variable.

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.

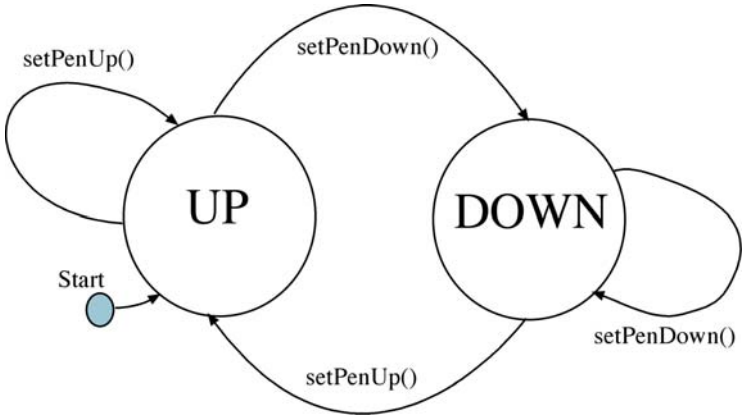

Figure 3-23: State Transition Diagram for pen_position

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.

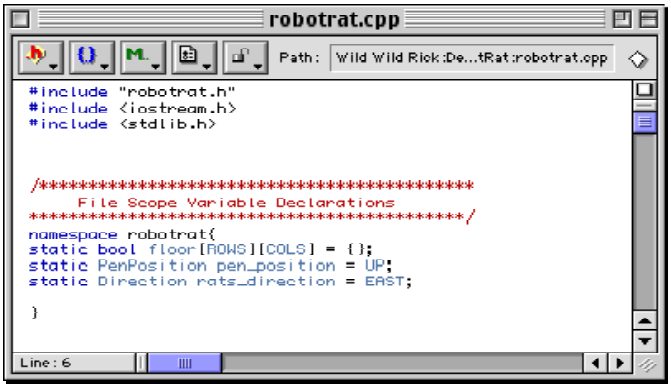

Figure 3-25: Declaration of pen_position & rats_position

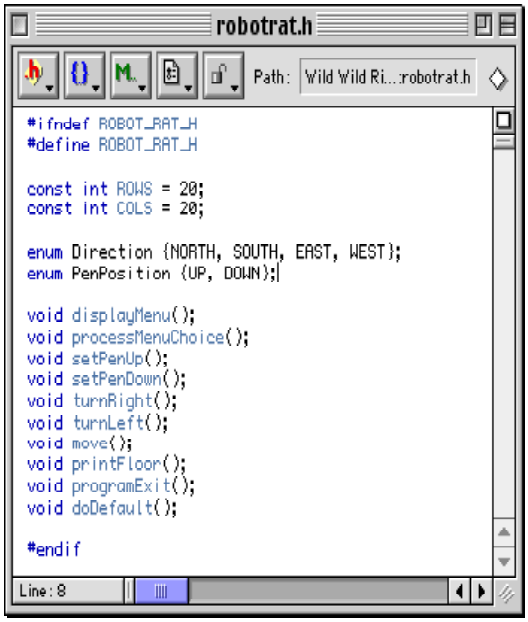

Figure 3-24: Direction and PenPosition Enum Types Added to robotrat.h

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.

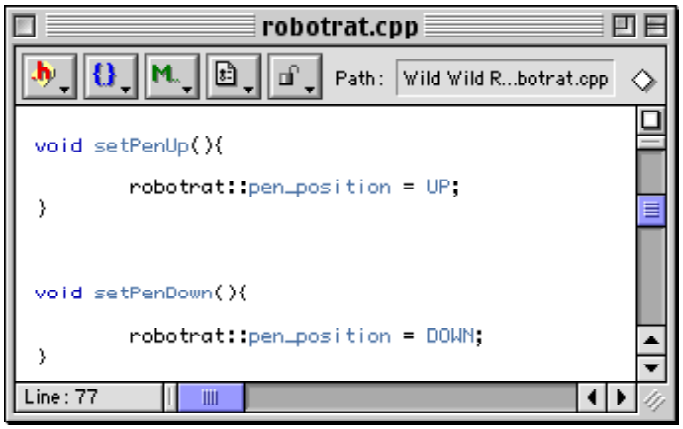

Figure 3-26: setPenUp() & setPenDown() Functions

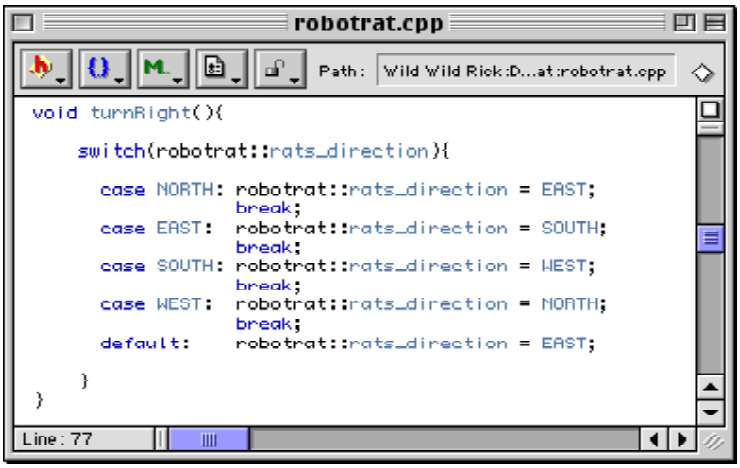

Figure 3-27: turnRight() Function

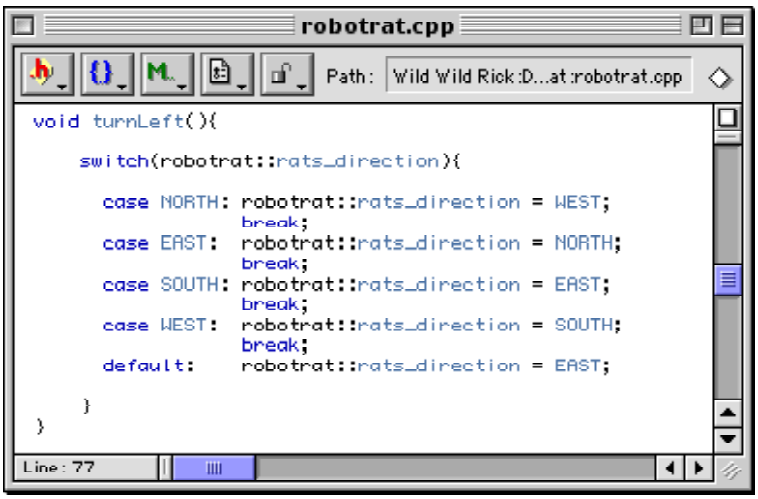

Figure 3-28: turnLeft() Function

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.

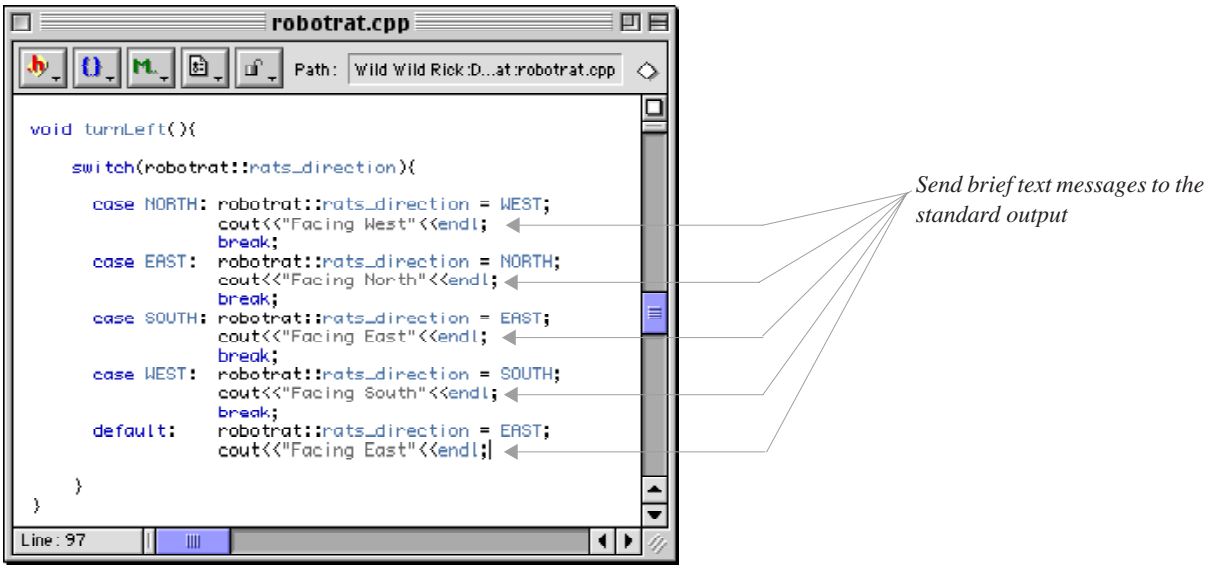

Figure 3-29: turnLeft() Function with cout Statements

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.

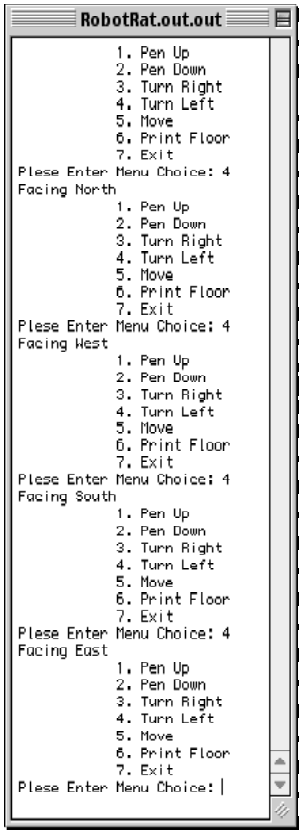

Figure 3-30: turnLeft() Test

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 me 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.

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.

3.4 move() function pseudocode

```
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)
```

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 be 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.

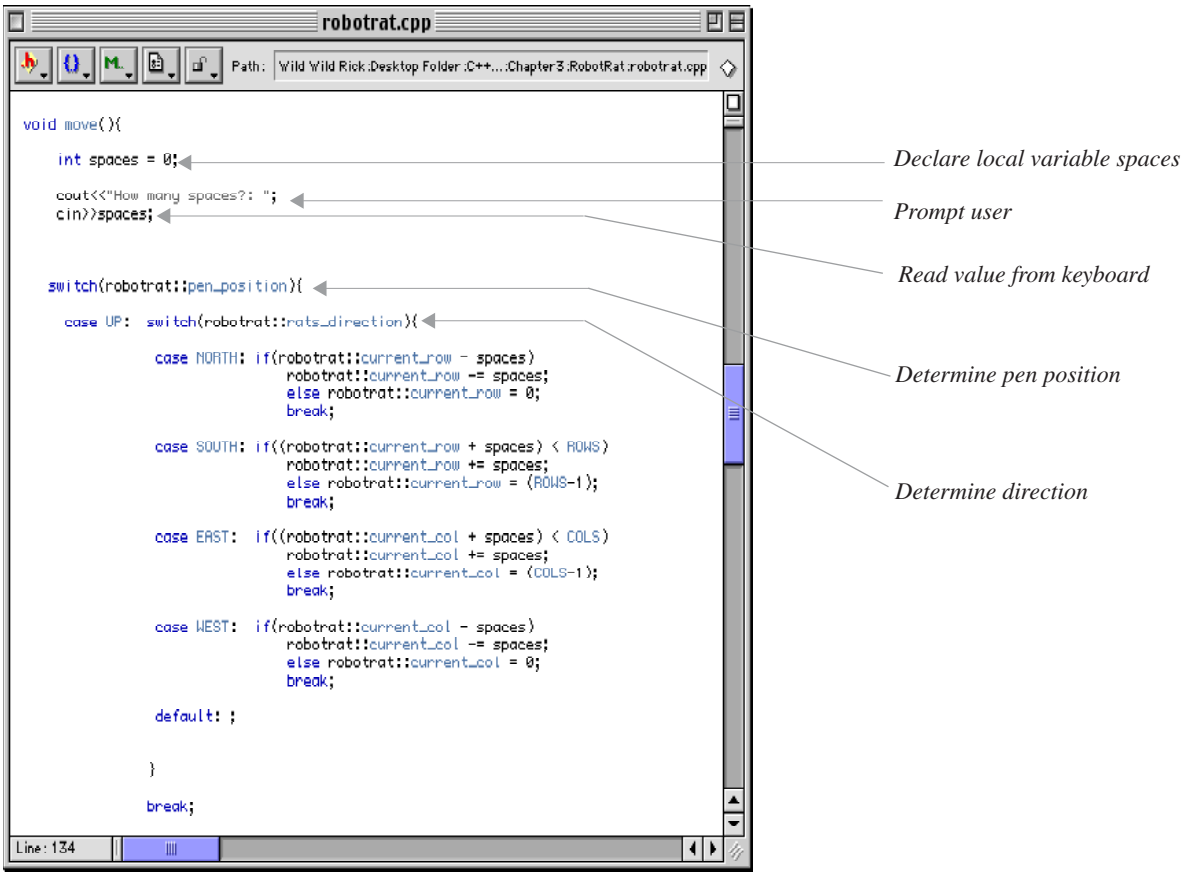

Figure 3-31: move() Function, Top Half

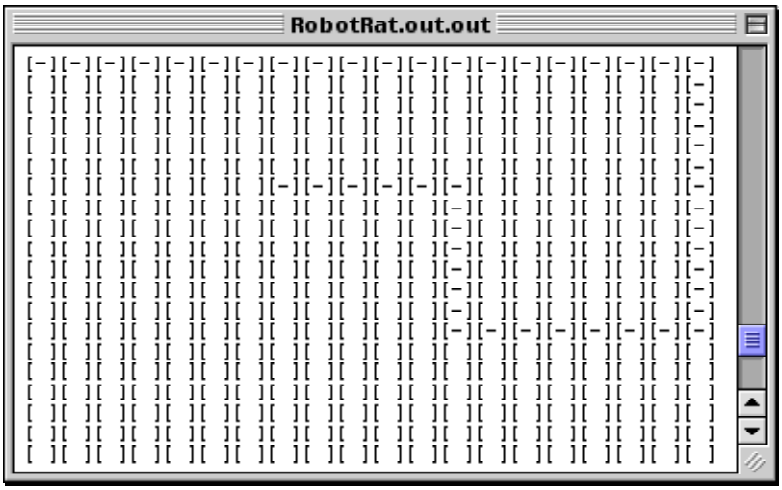

Figure 3-32: move() Function Test

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

3.7 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
```

```
/*******************************************
     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

