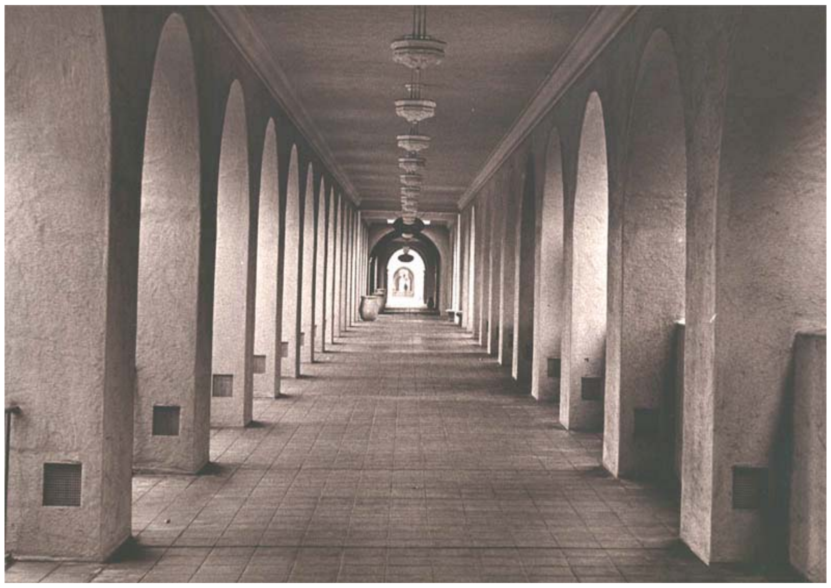
Chapter 7



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 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:

7.1 integer objects

```
int a = 1, b = 2;

cout<<"a = "<<a<<endl;
cout<<"b = "<<b<<endl;
```

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."

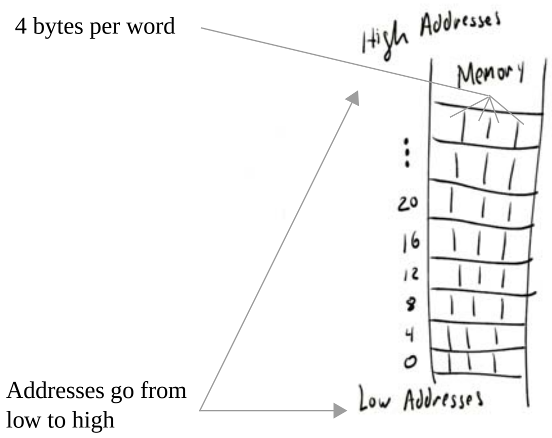


Figure 7-1: Memory

“Figure 7-2 shows another way of representing memory that reinforces the idea that it is a contiguous array of elements.

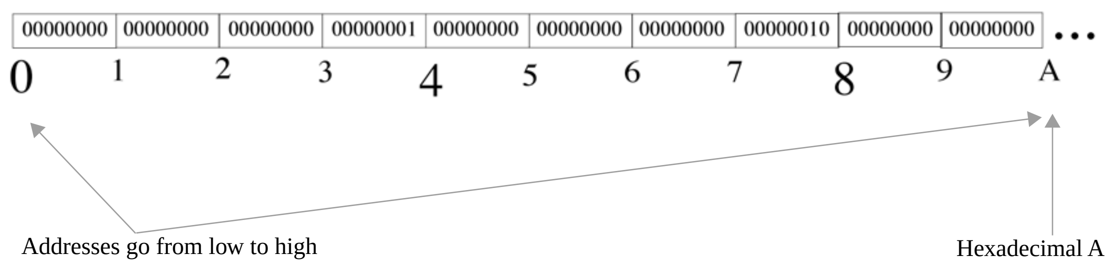


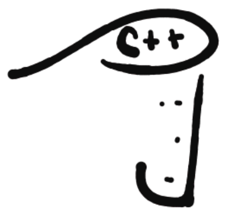
Figure 7-2: Another Way To Represent Memory

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.

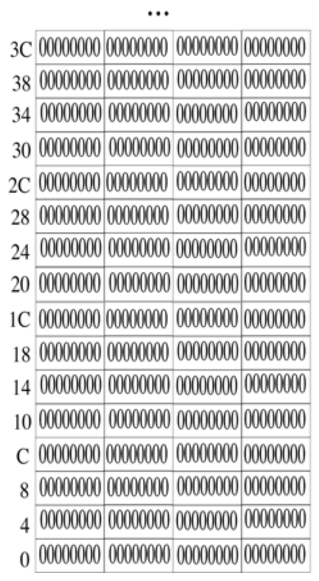


“So you see, Perplexed One, all objects in memory are accessible via an address. The addresses go from low to high and are organized according to the hardware architecture of the computer. A large object may span several addresses but have only one starting address.”

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.”

7.2 Using & operator

```
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!"

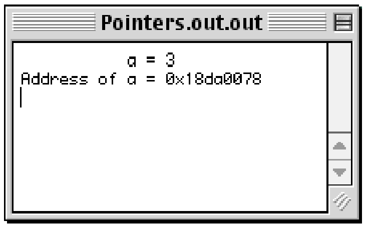
```
Hunk:Kind=HUNK_GLOBAL_CODE    Align=4  Class=PR  Name=".main"(22)  Size=136
00000000: 7C0802A6  mflr      r0
00000004: 90010008  stw       r0,8(SP)
00000008: 9421FFC0  stwu      SP,-64(SP)  <-------------  Set stack register (SP) to our stack
0000000C: 38000003  li        r0,3  <-------------------  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       r4,endl<c,Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream
                             <c,Q23std14char_traits<c>>(RTOC)
00000034: 48000001  bl        .__ls__Q23std39basic_ostream<c,Q23std14char_traits<c>>FPFRQ23std39basic_ostream
                             <c,Q23std14char_traits<c>>_RQ23std39basic_ostream<c,Q23std14char_traits<c>>
00000038: 80620000  lwz       r3,cout__3std(RTOC)
0000003C: 80820000  lwz       r4,@662(RTOC)
00000040: 48000001  bl        .__ls<Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream
                             <c,Q23std14char_traits<c>>PCc              endl, load "Address of a "
00000044: 60000000  nop                                                  ...then print
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       r4,endl<c,Q23std14char_traits<c>>__3stdFRQ23std39basic_ostream
                             <c,Q23std14char_traits<c>>(RTOC)
00000058: 48000001  bl        .__ls__Q23std39basic_ostream<c,Q23std14char_traits<c>>FPFRQ23std39basic_ostream
                             <c,Q23std14char_traits<c>>_RQ23std39basic_ostream<c,Q23std14char_traits<c>>
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  blr
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."

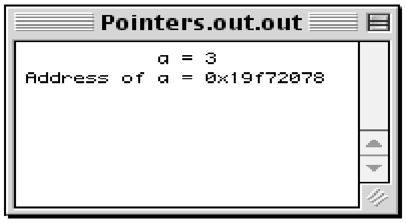
"Notice here, Perplexed One, that the address of a is shown to be 0x18da0078. Shutting down all other programs and then running example 7.2 again produced the results shown in figure 7-5."

"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."



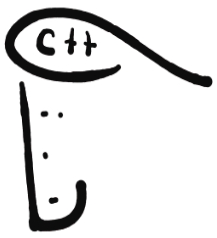
```
            a = 3
Address of a = 0x18da0078
```

Figure 7-4: Running Example 7.2 with CodeWarrior



```
            a = 3
Address of a = 0x19f72078
```

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.”

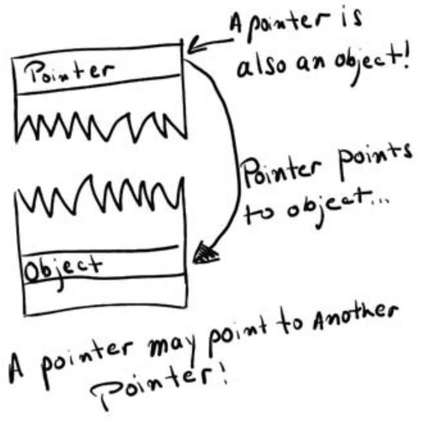

A pointer is also an object!
Pointer
Pointer points to object...
Object
A pointer may point to Another Pointer!

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!”

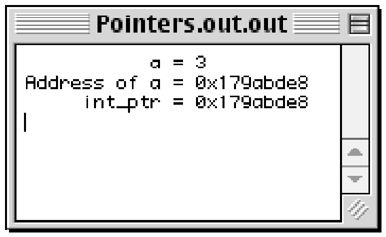
How do you declare a pointer?

“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:

*7.3 Using * operator*

```
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;
```

“Notice how the asterisk is used on line 2. The variable int_prt has the type pointer to integer or pointer to int. The address of the variable a is assigned to int_prt 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.”



```
           a = 3
Address of a = 0x179abde8
     int_ptr = 0x179abde8
```

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;
```

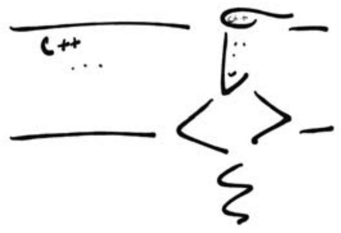
“I like the first method” Perplexed One said, busily scratching notes with his pencil. “It makes it clear to me what’s being done” he added.

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.

7.4 dereferencing pointers

```
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;
```

“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?”

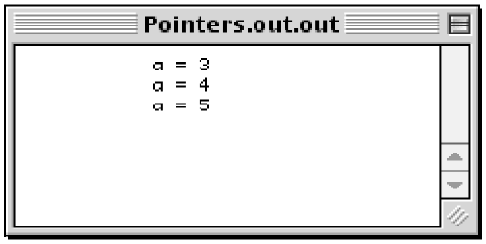
“Yep, that’s about it.” Here’s another example:

7.5 dereferencing pointers

```
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;
```

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."



```
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 though 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."

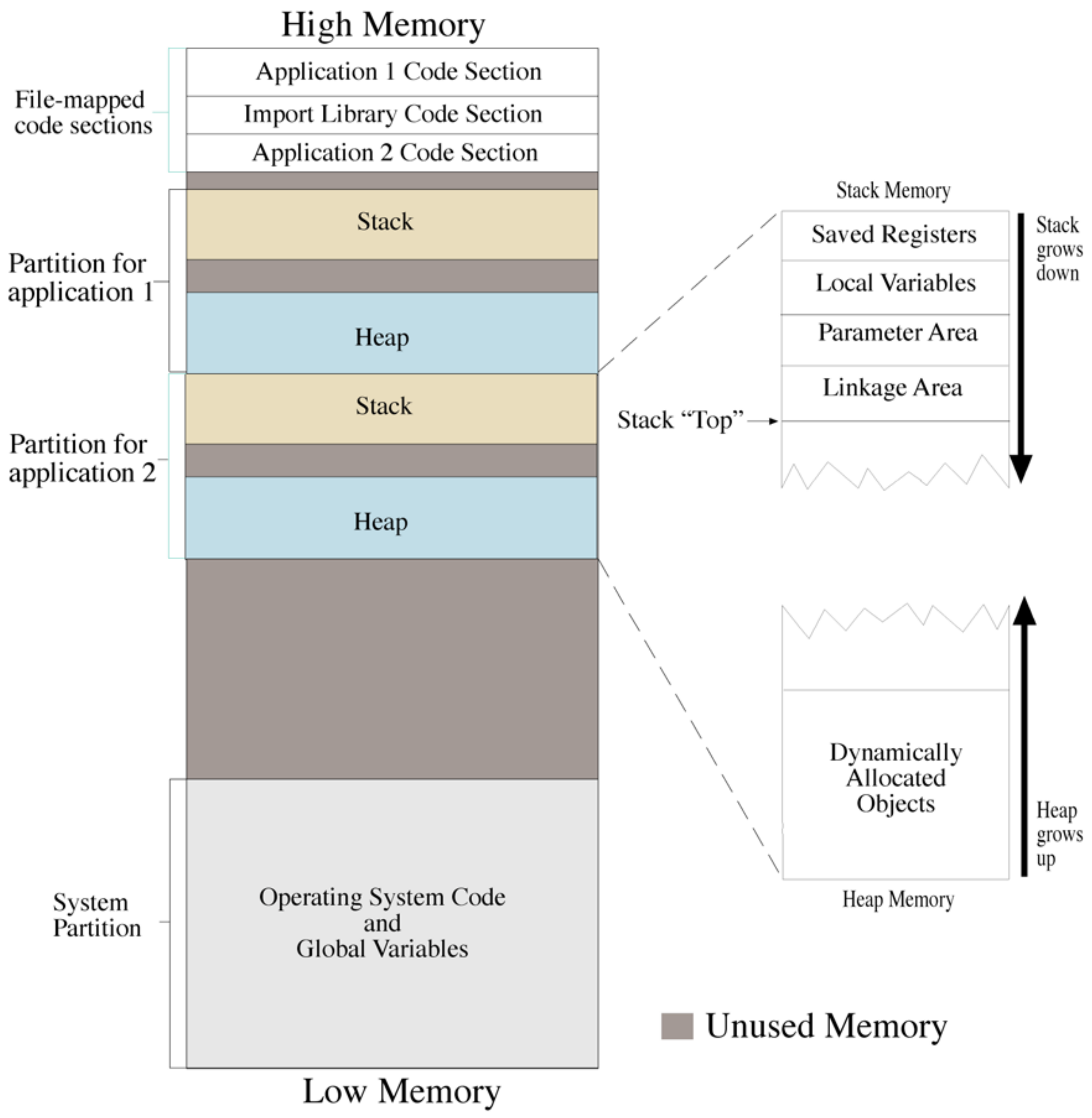

High Memory
File-mapped code sections
Application 1 Code Section
Import Library Code Section
Application 2 Code Section
Partition for application 1
Stack
Heap
Partition for application 2
Stack
Heap
System Partition
Operating System Code and Global Variables
Low Memory
Stack Memory
Saved Registers
Local Variables
Parameter Area
Linkage Area
Stack "Top"
Stack grows down
Dynamically Allocated Objects
Heap grows up
Heap Memory
Unused Memory

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

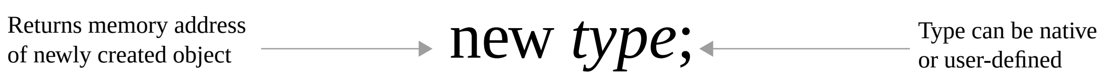
"The new operator is used to dynamically allocate objects during program runtime. The new operator makes a request for heap memory space to the operating system and, if memory is available, will return the newly created object's memory address. In the case of complex objects like structures and classes, 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:

7.6 dynamic memory allocation

```
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;
```

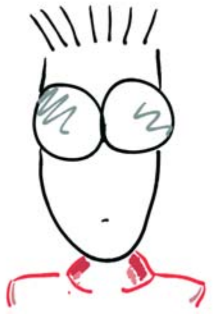
Perplexed One studied the code intensely and quickly copied it into his notebook. C++ Man began his explanation. "On line 1 I declared a pointer variable named int_ptr to hold the address returned by the new operator. On line 2 the new operator is used to create an object of type int. The address returned 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:

7.7 calling object constructor

```
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;
```

“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 towards the board.

7.8 calling object constructor

```
int* int_ptr = new int(7);
char* char_ptr = new char('e');

cout<<" The int value is: "<<*int_ptr<<endl;
cout<<"The char value is: "<<*char_ptr<<endl;

delete int_ptr;
delete char_ptr;
```

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 on 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, waiving his hand at the board. “Take a look at this code:”

7.9 using a reference

```
1 int a = 3;
2 int& ref_a = a;
3
4 cout<<"    a = "<<a<<endl;
5 cout<<"ref_a = "<<ref_a<<endl;
```

“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

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 Clifs, New Jersey, 1992. ISBN: 0-13-118233-1

Notes