# Part III

## References

# Memory Addressing, Binary, and Hexadecimal Review

You do not have to understand the concepts in this appendix to become well-versed in C++. You can master C++, however, only if you spend some time learning about the behind-the-scenes roles played by binary numbers. The material presented here is not difficult, but many programmers do not take the time to study it; hence, there are a handful of C++ masters who learn this material and understand how C++ works "under the hood," and there are those who will never master the language as they could.

You should take the time to learn about addressing, binary numbers, and hexadecimal numbers. These fundamental principles are presented here for you to learn, and although a working knowledge of C++ is possible without knowing them, they greatly enhance your C++ skills (and your skills in every other programming language).

After reading this appendix, you will better understand why different C++ data types hold different ranges of numbers. You also will see the importance of being able to represent hexadecimal numbers in C++, and you will better understand C++ array and pointer addressing.

## Computer Memory

Each memory location inside your computer holds a single character called a *byte*. A byte is any character, whether it is a letter of the alphabet, a numeric digit, or a special character such as a period, question mark, or even a space (a *blank* character). If your computer contains 640K of memory, it can hold a total of approximately 640,000 bytes of memory. This means that as soon as you fill your computer's memory with 640K, there is no room for an additional character unless you overwrite something.

Before describing the physical layout of your computer's memory, it is best to take a detour and explain exactly what 640K means.

## Memory and Disk Measurements

K means approximately 1000 bytes and exactly 1024 bytes.

By appending the *K* (from the metric word *kilo*) to memory measurements, the manufacturers of computers do not have to attach as many zeros to the end of numbers for disk and memory storage. The *K* stands for approximately 1000 bytes. As you will see, almost everything inside your computer is based on a power of 2. Therefore, the *K* of computer memory measurements actually equals the power of 2 closest to 1000, which is 2 to the 10th power, or 1024. Because 1024 is very close to 1000, computer-users often think of *K* as meaning 1000, even though they know it only approximately equals 1000.

Think for a moment about what 640K exactly equals. Practically speaking, 640K is about 640,000 bytes. To be exact, however, 640K equals 640 times 1024, or 655,360. This explains why the PC DOS command CHKDSK returns 655,360 as your total memory (assuming that you have 640K of RAM) rather than 640,000.

M means approximately 1,000,000 bytes and exactly 1,048,576 bytes.

Because extended memory and many disk drives can hold such a large amount of data, typically several million characters, there is an additional memory measurement shortcut called *M*, which stands for *meg*, or *megabytes*. The *M* is a shortcut for approximately one million bytes. Therefore, 20M is approximately 20,000,000 characters, or bytes, of storage. As with *K*, the *M* literally stands for 1,048,576 because that is the closest power of 2 (2 to the 20th power) to one million.

How many bytes of storage is 60 megabytes? It is approximately 60 million characters, or 62,914,560 characters to be exact.

## Memory Addresses

Each memory location in your computer, just as with each house in your town, has a unique *address*. A memory address is simply a sequential number, starting at 0, that labels each memory location. Figure A.1 shows how your computer memory addresses are numbered if you have 640K of RAM.

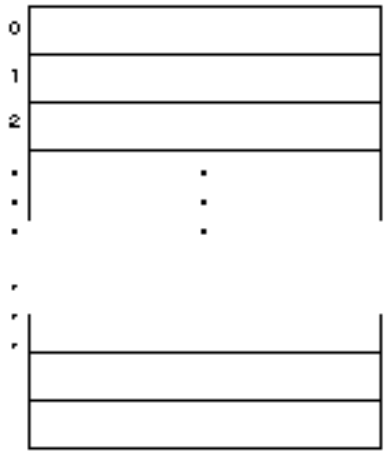



Figure A.1. Memory addresses for a 640K computer.

By using unique addresses, your computer can track memory. When the computer stores a result of a calculation in memory, it finds an empty address, or one matching the data area where the result is to go, and stores the result at that address.

Your C++ programs and data share computer memory with DOS. DOS must always reside in memory while you operate your computer. Otherwise, your programs would have no way to access disks, printers, the screen, or the keyboard. Figure A.2 shows computer memory being shared by DOS and a C++ program. The exact amount of memory taken by DOS and a C++ program is determined by the version of DOS you use, how many DOS extras (such as device drivers and buffers) your computer uses, and the size and needs of your C++ programs and data.

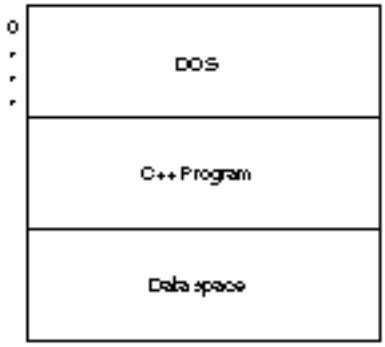



Figure A.2. DOS, your C++ program, and your program's data share the same memory.

## Bits and Bytes

You now know that a single address of memory might contain any character, called a byte. You know that your computer holds many bytes of information, but it does not store those characters in the same way that humans think of characters. For example, if you type a letter *W* on your keyboard while working in your C++ editor, you see the *W* on-screen, and you also know that the *W* is stored in a memory location at some unique address. Actually, your computer does not store the letter *W;* it stores electrical impulses that stand for the letter *W*.

Electricity, which runs through the components of your computer and makes it understand and execute your programs, can exist in only two states—on or off. As with a light bulb, electricity is either flowing (it is on) or it is not flowing (it is off). Even though you can dim some lights, the electricity is still either on or off.

The binary digits 1 and 0 (called *bits*) represent on and off states of electricity.

Today's modern digital computers employ this on-or-off concept. Your computer is nothing more than millions of on and off switches. You might have heard about integrated circuits, transistors, and even vacuum tubes that computers have contained over the years. These electrical components are nothing more than switches that rapidly turn electrical impulses on and off.

This two-state on and off mode of electricity is called a *binary* state of electricity. Computer people use a 1 to represent an on state (a switch in the computer that is on) and a 0 to represent an off state (a switch that is off). These numbers, 1 and 0, are called *binary digits.* The term binary digits is usually shortened to *bits.* A bit is either a 1 or a 0 representing an on or an off state of electricity. Different combinations of bits represent different characters.

Several years ago, someone listed every single character that might be represented on a computer, including all uppercase letters, all lowercase letters, the digits 0 through 9, the many other characters (such as %, *, {, and +), and some special control characters. When you add the total number of characters that a PC can represent, you get 256 of them. The 256 ASCII characters are listed in Appendix C's ASCII (pronounced *ask-ee*) table.

The order of the ASCII table's 256 characters is basically arbitrary, just as the telegraph's Morse code table is arbitrary. With Morse code, a different set of long and short beeps represent different letters of the alphabet. In the ASCII table, a different combination of bits (1s and 0s strung together) represent each of the 256 ASCII characters. The ASCII table is a standard table used by almost every PC in the world. ASCII stands for *American Standard Code for Information Interchange.* (Some minicomputers and mainframes use a similar table called the EBCDIC table.)

It turns out that if you take every different combination of eight 0s strung together, to eight 1s strung together (that is, from 00000000, 00000001, 00000010, and so on until you get to 11111110, and finally, 11111111), you have a total of 256 of them. (256 is 2 to the 8th power.)

Each memory location in your computer holds eight bits each. These bits can be any combination of eight 1s and 0s. This brings us to the following fundamental rule of computers.

> **NOTE:** Because it takes a combination of eight 1s and 0s to represent a character, and because each byte of computer memory can hold exactly one character, eight bits equals one byte.

To bring this into better perspective, consider that the bit pattern needed for the uppercase letter *A* is 01000001. No other character in the ASCII table "looks" like this to the computer because each of the 256 characters is assigned a unique bit pattern.

Suppose that you press the *A* key on your keyboard. Your keyboard does *not* send a letter *A* to the computer; rather, it looks in its ASCII table for the on and off states of electricity that represent the letter *A*. As Figure A.3 shows, when you press the *A* key, the keyboard actually sends 01000001 (as on and off impulses) to the computer. Your computer simply stores this bit pattern for *A* in a memory location. Even though you can think of the memory location as holding an *A*, it really holds the byte 01000001.

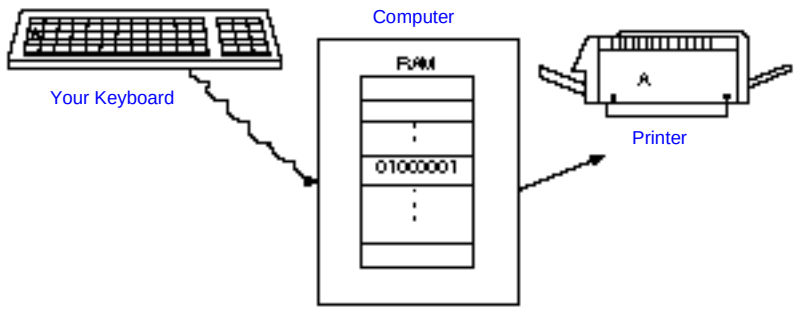



Figure A.3. Your computer keeps track of characters by their bit patterns.

If you were to print that *A*, your computer would not send an *A* to the printer; it would send the 01000001 bit pattern for an *A* to the printer. The printer receives that bit pattern, looks up the correct letter in the ASCII table, and prints an *A*.

From the time you press the *A* until the time you see it on the printer, it is not a letter *A!* It is the ASCII pattern of bits that the computer uses to represent an *A*. Because a computer is electrical, and because electricity is easily turned on and off, this is a nice way for the computer to manipulate and move characters, and it can do so very quickly. Actually, if it were up to the computer, you would enter everything by its bit pattern, and look at all results in their bit patterns. Of course, it would be much more difficult for us to learn to program and use a computer, so devices such as the keyboard, screen, and printer are created to work part of the time with letters as we know them. That is why the ASCII table is such an integral part of a computer.

There are times when your computer treats two bytes as a single value. Even though memory locations are typically eight bits wide, many CPUs access memory two bytes at a time. If this is the case, the two bytes are called a *word* of memory. On other computers (commonly mainframes), the word size might be four bytes (32 bits) or even eight bytes (64 bits).

**Summarizing Bits and Bytes**

A bit is a 1 or a 0 representing an on or an off state of electricity.

Eight bits represents a byte.

A byte, or eight bits, represents one character.

Each memory location of your computer is eight bits (a single byte) wide. Therefore, each memory location can hold one character of data. Appendix C is an ASCII table listing all possible characters.

If the CPU accesses memory two bytes at a time, those two bytes are called a word of memory.

## The Order of Bits

To further understand memory, you should understand how programmers refer to individual bits. Figure A.4 shows a byte and a two-byte word. Notice that the bit on the far right is called bit 0. From bit 0, keep counting by ones as you move left. For a byte, the bits are numbered 0 to 7, from right to left. For a double-byte (a 16-bit word), the bits are numbered from 0 to 15, from right to left.

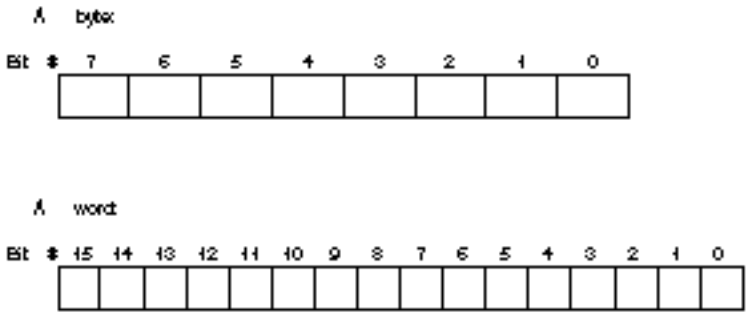



Figure A.4. The order of bits in a byte and a two-byte word.

Bit 0 is called the *least-significant bit*, or sometimes the *low-order bit*. Bit 7 (or bit 15 for a two-byte word) is called the *most-significant bit*, or sometimes the *high-order bit*.

## Binary Numbers

Because a computer works best with 1s and 0s, its internal numbering method is limited to a *base-2* (binary) numbering system. People work in a *base-10* numbering system in the "real" world. The base-10 numbering system is sometimes called the decimal numbering system. There are always as many different digits as the base in a numbering system. For example, in the base-10 system, there are ten digits, 0 through 9. As soon as you count to 9 and run out of digits, you have to combine some that you already used. The number 10 is a representation of ten values, but it combines the digits 1 and 0.

The same is true of base-2. There are only two digits, 0 and 1. As soon as you run out of digits, after the second one, you have to reuse digits. The first seven binary numbers are 0, 1, 10, 11, 100, 101, and 110.

It is okay if you do not understand how these numbers were derived; you will see how in a moment. For the time being, you should realize that no more than two digits, 0 and 1, can be used to represent any base-2 number, just as no more than ten digits, 0 through 9, can be used to represent any base-10 number in the regular numbering system.

You should know that a base-10 number, such as 2981, does not really mean anything by itself. You must assume what base it is. You get very used to working with base-10 numbers because you use them every day. However, the number 2981 actually represents a quantity based on powers of 10. For example, Figure A.5 shows what the number 2981 actually represents. Notice that each digit in the number represents a certain number of a power of 10.

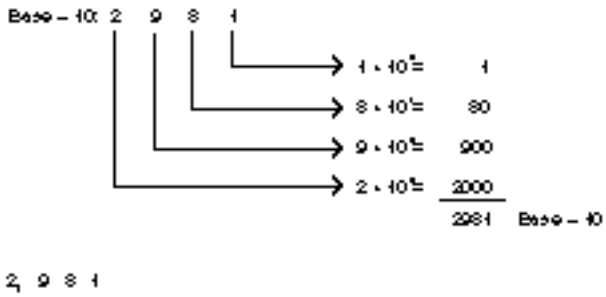



Figure A.5. The base-10 breakdown of the number 2981.

A binary number can contain only the digits 1 and 0.

This same concept applies when you work in a base-2 numbering system. Your computer does this because the power of 2 is just as common to your computer as the power of 10 is to you. The only difference is that the digits in a base-2 number represent powers of 2 and not powers of 10. Figure A.6 shows you what the binary numbers 10101 and 10011110 are in base-10. This is how you convert any binary number to its base-10 equivalent.

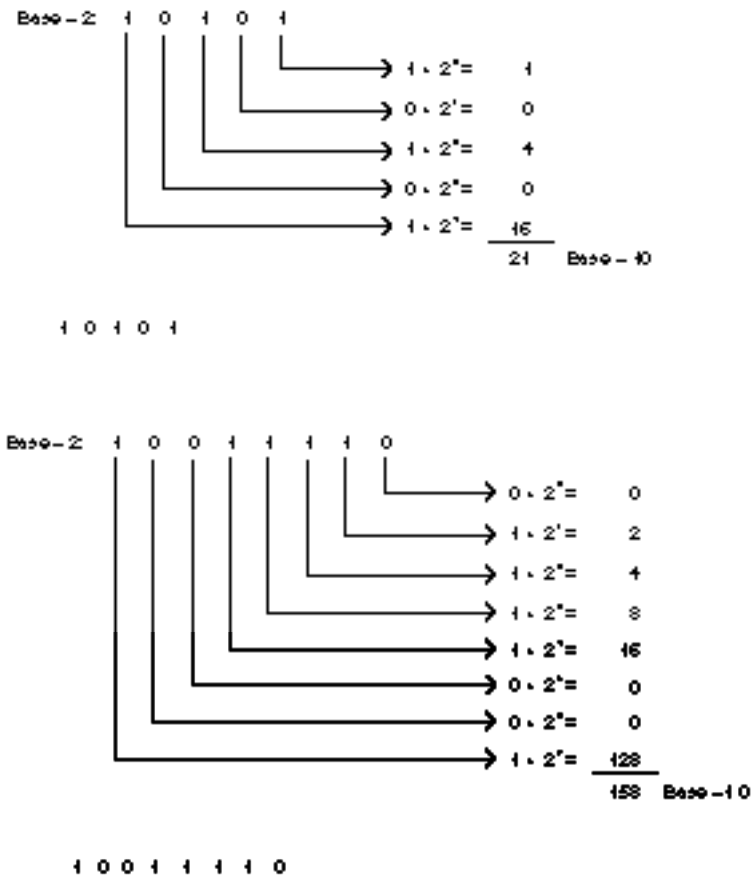



Figure A.6. The base-2 breakdown of the numbers 10101 and 10011110.

A base-2 number contains only 1s and 0s. To convert any base-2 number to base-10, add each power of 2 everywhere a 1 appears in the number. The base-2 number 101 represents the base-10 number 5. (There are two 1s in the number, one in the 2 to the 0 power, which equals 1, and one in the 2 to the second power, which equals 4.) Table A.1 shows the first 18 base-10 numbers and their matching base-2 numbers.

Table A.1. The first 17 base-10 and base-2 (binary) numbers.

| *Base-10* | *Base-2* |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 1010 |
| 11 | 1011 |
| 12 | 1100 |
| 13 | 1101 |
| 14 | 1110 |
| 15 | 1111 |
| 16 | 10000 |
| 17 | 10001 |

You do not have to memorize this table; you should be able to figure the base-10 numbers from their matching binary numbers by adding the powers of two for each 1 (on) bit. Many programmers do memorize the first several binary numbers because it comes in handy in advanced programming techniques.

What is the largest binary number a byte can hold? The answer is all 1s, or 11111111. If you add the first eight powers of 2, you get 255.

A byte holds either a number or an ASCII character, depending on how it is accessed. For example, if you were to convert the base-2 number 01000001 to a base-10 number, you would get 65. However, this also happens to be the ASCII bit pattern for the uppercase letter *A*. If you check the ASCII table, you see that the *A* is ASCII code 65. Because the ASCII table is so closely linked with the bit patterns, the computer knows whether to work with a number 65 or a letter *A*—by the context of how the patterns are used.

A binary number is not limited to a byte, as an ASCII character is. Sixteen or 32 bits at a time can represent a binary number (and usually do). There are more powers of 2 to add when converting that number to a base-10 number, but the process is the same. By now you should be able to figure out that 1010101010101010 is 43,690 in base-10 decimal numbering system (although it might take a little time to calculate).

To convert from decimal to binary takes a little more effort. Luckily, you rarely need to convert in that direction. Converting from base-10 to base-2 is not covered in this appendix.

## Binary Arithmetic

At their lowest level, computers can only add and convert binary numbers to their negative equivalents. Computers cannot truly subtract, multiply, or divide, although they simulate these operations through judicious use of the addition and negative-conversion techniques.

If a computer were to add the numbers 7 and 6, it could do so (at the binary level). The result is 13. If, however, the computer were instructed to subtract 7 from 13, it could not do so. It can, however, take the negative value of 7 and add that to 13. Because –7 plus 13 equals 6, the result is a *simulated* subtraction.

To multiply, computers perform repeated addition. To multiply 6 by 7, the computer adds seven 6s together and gets 42 as the answer. To divide 42 by 7, a computer keeps subtracting 7 from 42 repeatedly until it gets to a 0 answer (or less than 0 if there is a remainder), then counts the number of times it took to reach 0.

Because all math is done at the binary level, the following additions are possible in binary arithmetic:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 10

Because these are binary numbers, the last result is not the number 10, but the binary number 2. (Just as the binary 10 means "no ones, and carry an additional power of 2," the decimal number 10 means "no ones, and carry a power of 10.") No binary digit represents a 2, so you have to combine the 1 and the 0 to form the new number.

Because binary addition is the foundation of all other math, you should learn how to add binary numbers. You will then understand how computers do the rest of their arithmetic.

Using the binary addition rules shown previously, look at the following binary calculations:

```
  01000001  (65 decimal)
+ 00101100  (44 decimal)
  01101101  (109 decimal)
```

The first number, 01000001, is 65 decimal. This also happens to be the bit pattern for the ASCII *A*, but if you add with it, the computer interprets it as the number 65 rather than the character *A*.

The following binary addition requires a carry into bit 4 and bit 6:

```
  00101011  (43 decimal)
+ 00100111  (39 decimal)
  01010010  (82 decimal)
```

Typically, you have to ignore bits that carry past bit 7, or bit 15 for double-byte arithmetic. For example, both of the following

binary additions produce incorrect positive results:

```
 10000000  (128 decimal)
+10000000  (128 decimal)
 00000000  (0 decimal)
```

```
 1000000000000000
 (65536 decimal)
+1000000000000000
 (65536 decimal)
 0000000000000000
 (0 decimal)
```

There is no 9th or 17th bit for the carry, so both of these seem to produce incorrect results. Because the byte and 16-bit word cannot hold the answers, the magnitude of both these additions is not possible. The computer must be programmed, at the bit level, to perform *multiword arithmetic,* which is beyond the scope of this book.

## Binary Negative Numbers

Negative binary numbers are stored in their 2's complement format.

Because subtracting requires understanding binary negative numbers, you need to learn how computers represent them. The computer uses *2's complement* to represent negative numbers in binary form. To convert a binary number to its 2's complement (to its negative) you must:

1. Reverse the bits (the 1s to 0s and the 0s to 1s).
2. Add 1.

This might seem a little strange at first, but it works very well for binary numbers. To represent a binary –65, you have to take the binary 65 and convert it to its 2's complement, such as

```
 01000001  (65 decimal)
 10111110  (Reverse the bits)
+1 (Add 1)
 10111111  (–65 binary)
```

By converting the 65 to its 2's complement, you produce –65 in binary. You might wonder what makes 10111111 mean –65, but by the 2's complement definition it means –65.

If you were told that 10111111 is a negative number, how would you know which binary number it is? You perform the 2's complement on it. Whatever number you produce is the positive of that negative number. For example:

```
10111111 (–65 decimal)
01000000 (Reverse the bits)
       +1 (Add 1)
01000001 (65 decimal)
```

Something might seem wrong at this point. You just saw that 10111111 is the binary –65, but isn't 10111111 also 191 decimal (adding the powers of 2 marked by the 1s in the number, as explained earlier)? It depends whether the number is a *signed* or an *unsigned* number. If a number is signed, the computer looks at the most-significant bit (the bit on the far left), called the *sign bit*. If the most-significant bit is a 1, the number is negative. If it is 0, the number is positive.

Most numbers are 16 bits long. That is, two-byte words are used to store most integers. This is not always the case for all computers, but it is true for most PCs.

In the C++ programming language, you can designate numbers as either signed integers or unsigned integers (they are signed by default if you do not specify otherwise). If you designate a variable as a signed integer, the computer interprets the high-order bit as a sign bit. If the high-order bit is on (1), the number is negative. If the high-order bit is off (0), the number is positive. If, however, you designate a variable as an unsigned integer, the computer uses the high-order bit as just another power of 2. That is why the range of unsigned integer variables goes higher (generally from 0 to 65535, but it depends on the computer) than for signed integer variables (generally from –32768 to +32767).

After so much description, a little review is in order. Assume that the following 16-bit binary numbers are unsigned:

0011010110100101

1001100110101010

1000000000000000

These numbers are unsigned, so the bit 15 is not the sign bit, but simply another power of 2. You should practice converting these large 16-bit numbers to decimal. The decimal equivalents are

13733

39338

32768

If, on the other hand, these numbers are signed numbers, the high-order bit (bit 15) indicates the sign. If the sign bit is 0, the numbers are positive and you convert them to decimal in the usual manner. If the sign bit is 1, you must convert the numbers to their 2's complement to find what they equal. Their decimal equivalents are

+13733

–26198

–32768

To compute the last two binary numbers to their decimal equivalents, take their 2's complement and convert it to decimal. Put a minus sign in front of the result and you find what the original number represents.

**TIP:** To make sure that you convert a number to its 2's complement correctly, you can add the 2's complement to its original positive value. If the answer is 0 (ignoring the extra carry to the left), you know that the 2's complement number is correct. This is similar to the concept that decimal opposites, such as –72 + 72, add up to zero.

# Hexadecimal Numbers

Hexadecimal numbers use 16 unique digits, 0 through F.

All those 1s and 0s get confusing. If it were up to your computer, however, you would enter *everything* as 1s and 0s! This is unacceptable to people because we do not like to keep track of all those 1s and 0s. Therefore, a *hexadecimal* numbering system (sometimes called *hex*) was devised. The hexadecimal numbering system is based on base-16 numbers. As with other bases, there are 16 unique digits in the base-16 numbering system. Here are the first 19 hexadecimal numbers:

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12

Because there are only 10 unique digits (0 through 9), the letters *A* through *F* represent the remaining six digits. (Anything could have been used, but the designers of the hexadecimal numbering system decided to use the first six letters of the alphabet.)

To understand base-16 numbers, you should know how to convert them to base-10 so they represent numbers with which people are familiar. Perform the conversion to base-10 from base-16 the same way you did with base-2, but instead of using powers of 2, represent each hexadecimal digit with powers of 16. Figure A.7 shows how to convert the number 3C5 to decimal.

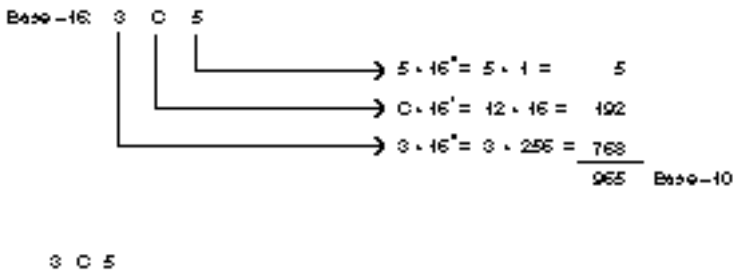



Figure A.7. Converting hexadecimal 3C5 to its decimal equivalent.

**TIP:** There are calculators available that convert numbers between base-16, base-10, and base-2, and also perform 2's complement arithmetic.

You should be able to convert 2B to its decimal 43 equivalent, and E1 to decimal 225 in the same manner. Table A.2 shows the first 20 decimal, binary, and hexadecimal numbers.

Table A.2. The first 20 base-10, base-2 (binary), and base-16 (hexadecimal) numbers.

| *Base-10* | *Base-2* | *Base-16* |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 10 | 2 |
| 3 | 11 | 3 |
| 4 | 100 | 4 |
| 5 | 101 | 5 |
| 6 | 110 | 6 |
| 7 | 111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| 10 | 1010 | A |
| 11 | 1011 | B |
| 12 | 1100 | C |
| 13 | 1101 | D |
| 14 | 1110 | E |
| 15 | 1111 | F |
| 16 | 10000 | 10 |
| 17 | 10001 | 11 |
| 18 | 10010 | 12 |
| 19 | 10011 | 13 |
| 20 | 10100 | 14 |

## Why Learn Hexadecimal?

Because of its close association to the binary numbers your computer uses, hexadecimal notation is extremely efficient for describing memory locations and values. It is much easier for you (and more importantly at this level, for your computer) to convert from base-16 to base-2 than from base-10 to base-2. Therefore, you sometimes want to represent data at the bit level, but using hexadecimal notation is easier (and requires less typing) than using binary numbers.

To convert from hexadecimal to binary, convert each hex digit to its four-bit binary number. You can use Table A.2 as a guide for this. For example, the following hexadecimal number

5B75

can be converted to binary by taking each digit and converting it to four binary numbers. If you need leading zeroes to "pad" the four digits, use them. The number becomes

0101 1011 0111 0101

It turns out that the binary number 0101101101110101 is exactly equal to the hexadecimal number 5B75. This is much easier than converting them both to decimal first.

To convert from binary to hexadecimal, reverse this process. If you were given the following binary number

11010000111101111111010

you could convert it to hexadecimal by grouping the bits into groups of four, starting with the bit on the far right. Because there is not an even number of groups of four, pad the one on the far left with zeroes. You then have the following:

0011 0100 0011 1101 1111 1010

Now you only have to convert each group of four binary digits into their hexadecimal number equivalent. You can use Table A.2 to help. You then get the following base-16 number:

343DFA

The C++ programming language also supports the base-8 *octal* representation of numbers. Because octal numbers are rarely used with today's computers, they are not covered in this appendix.

## How Binary and Addressing Relate to C++

The material presented here may seem foreign to many programmers. The binary and 2's complement arithmetic reside deep in your computer, shielded from most programmers (except assembly-language programmers). Understanding this level of your computer, however, makes everything else you learn seem more clear.

Many C++ programmers learn C++ before delving into binary and hexadecimal representation. For those programmers, much about the C++ language seems strange, but it could be explained very easily if they understood the basic concepts.

For example, a signed integer holds a different range of numbers than an unsigned integer. You now know that this is because the sign bit is used in two different ways, depending on whether the number is designated as signed or unsigned.

The ASCII table (see Appendix C) also should make more sense to you after this discussion. The ASCII table is an integral part of your computer. Characters are not actually stored in memory and variables; rather, their ASCII bit patterns are. That is why C++ can move easily between characters and integers. The following two C++ statements are allowed, whereas they probably would not be in another programming language:

```
char c = 65;    // Places the ASCII letter A in c.
int ci = 'A';   // Places the number 65 in ci.
```

The hexadecimal notation also makes much more sense if you truly understand base-16 numbers. For example, if you see the following line in a C++ program

```
char a = '\x041';
```

you could convert the hex 41 to decimal (65 decimal) if you want to know what is being assigned. Also, C++ systems programmers find that they can better interface with assembly-language programs when they understand the concepts presented in this appendix.

If you gain only a cursory knowledge of this material at this point, you will be very much ahead of the game when you program in C++!

# Answers to Review Questions

## Chapter 1

1. BCPL or Algol
2. True
3. 1980s
4. False. C++'s compact size makes it an excellent programming language for smaller computers.
5. The hard disk
6. A modem
7. b. Input. By moving the mouse, you give cursor-direction commands to the computer.
8. NumLock
9. UNIX

10. When you turn off the computer, the contents of RAM are destroyed.
11. True
12. 524,288 bytes (512 times 1,024)
13. *Mo*dulate, *dem*odulate

## Chapter 2

1. A set of detailed instructions that tells the computer what to do.
2. Buy one or write it yourself.
3. False
4. The program produces the output.
5. A program editor
6. The .CPP extension
7. You must first plan the program by deciding which steps you will take to produce the final program.
8. To get the errors out of your program
9. So your programs work with various compilers and computer equipment
10. False. You must compile a program before linking it. Most compilers link the program automatically.

## Chapter 3

1. Two comment markers (`//`)
2. A holding place for data that can be changed
3. A value that cannot be changed
4. The `+`, `-`, `*`, and `/` operators

5. The = assignment operator.
6. False. There are floating-point, double floating-point, short integers, long integers, and many more variable data types.
7. `cout`
8. `city` must be a variable name because it is not enclosed in quotation marks.
9. All C++ commands must be in lowercase.

## Chapter 4

1. `my_name` and `sales_89`
2. Characters: `'X'` and `'0'`

   Strings: `"2.0"` and `"X"`

   Integers: `0` and `-708`

   Floating-point literals: `-12.0` and `65.4`
3. Seven variables are declared: three integers, three characters, and one floating-point variable.
4. A null zero, also called a binary zero or an ASCII zero.
5. True
6. 1
7. It is stored as a series of ASCII values, representing the characters and blanks in the string, ending in an ASCII `0`.
8. It is stored as a single ASCII `0`.
9. The constant value called `age` cannot be changed.

## Chapter 5

1. `char my_name[] "This is C++";`
2. The string is 11 characters long.

3. It consumes 12 bytes.
4. All string literals end with a binary zero.
5. Two character arrays are declared, each with 25 elements.
6. False. The keyword `char` must precede the variable name.
7. True. The binary zero terminates the string.
8. False. The characters do not represent a string because there is no terminating zero.

## Chapter 6

1. False. You can define only constants with the `#define` preprocessor directive.
2. The `#include` directive
3. The `#define` directive
4. True
5. The preprocessor changes your source code before the compiler reads the source code.
6. The `const` keyword
7. Use angled brackets when the `include` files reside in the compiler's `include` subdirectory. Use quotation marks when the `include` file resides in the same subdirectory as the source program.
8. Defined literals are easier to change because you have to change only the line with `#define` and not several other lines in the program.
9. iostream.h
10. False. You cannot define constants enclosed in quotation marks (as `"MESSAGE"` is in the `cout` statement).
11. `Amount is 4`

# Chapter 7

1. `cout` sends output to the screen, and `cin` gets input from the keyboard.
2. The prompt tells the user what is expected.
3. The user enters four values.
4. `cin` assigns values to variables when the user types them, whereas the programmer must assign data when using the assignment operator (=).
5. True. When printing strings, you do not need `%s`.
6. Arrays
7. `The backslash "\" character is special`
8. The following value prints, with one leading space: `123.456`

# Chapter 8

1. a. 5

   b. 6

   c. 5
2. a. 2

   b. 7
3. a. `a = (3+3) / (4+4);`

   b. `x = (a-b)*( (a-c) * (a-c));`

   c. `f = (a*a)/(b*b*b);`

   d. `d = ((8 - x*x)/(x - 9))-((4*2 - 1)/(x*x*x));`
4. The area of a circle:

```
#include stdio.h>
const float PI = 3.14159;
main()
```

```
{
   printf("%f", (PI*(4*4));
   return;
}
```

5. Assignment and `printf()` statements:

```
r = 100%4;
cout <<  r;
```

## Chapter 9

1. The `==` operator
2. a. True

   b. True

   c. True

   d. True
3. True
4. The `if` statement determines what code executes when the relational test is true. The `if-else` statement determines what happens for both the True and the False relational test.
5. No
6. a. False

   b. False

   c. False

## Chapter 10

1. The `&&`, `||`, and `!` operators are the three logical operators.
2. a. False

   b. False

c. True

d. True

3. a. True

   b. True

   c. True

4. `g is 25 and f got changed to 8`

5. a. True

   b. True

   c. False

   d. True

6. Yes

## Chapter 11

1. The `if-else` statement
2. The conditional operator is the only C++ operator with three arguments.
3. 
```
if (a == b)
   { ans = c + 2; }
else
   { ans = c + 3; }
```
4. True
5. The increment and decrement operators compile into single assembly instructions.
6. A comma operator (`,`), which forces a left-to-right execution of the statements on either side
7. The output cannot be determined reliably. Do not pass an increment operator as an argument.

8. `The size of name is 20`
9. a. True
   b. True
   c. False
   d. False

## Chapter 12

1. The `while` loop tests for a true condition at the beginning of the loop. The `do-while` tests for the condition at the end of the loop.
2. A counter variable increments by one. A total variable increments by the addition to the total you are performing.
3. The ++ operator
4. If the body of the loop is a single statement, the braces are not required. However, braces are *always* recommended.
5. There are no braces. The second `cout` always executes, regardless of the result of the `while` loop's relational test.
6. The stdlib.h header file
7. One time
8. By returning a value inside the `exit()` function's parentheses
9. 
```
This is the outer loop
This is the outer loop
This is the outer loop
This is the outer loop
```

# Chapter 13

1. A loop is a sequence of one or more instructions executed repeatedly.
2. False
3. A nested loop is a loop within a loop.
4. Because the expressions might be initialized elsewhere, such as before the loop or in the body of the loop
5. The inner loop
6. 
```
10
7

4

1
```
7. True
8. The body of the `for` loop stops repeating.
9. False, due to the semicolon after the first `for` loop
10. There is no output. The value of `start` is already less than `end` when the loop begins; therefore, the `for` loop's test is immediately False.

# Chapter 14

1. Timing loops force a program to pause.
2. Because some computers are faster than others.
3. If the `continue` and `break` statements were unconditional, there would be little use for them.
4. Because of the unconditional `continue` statement, there is no output.
5. `*****`

6. A single variable rarely can hold a large enough value for the timer's count.

## Chapter 15

1. The program does not execute sequentially, as it would without `goto`.
2. The `switch` statement
3. A `break` statement
4. False because you should place the `case` most likely to execute at the beginning of the `case` options.
5. 
```
switch (num)
{ case (1) : { cout << "Alpha";
               break; }
  case (2) : { cout << "Beta";
               break; }
  case (3) : { cout << "Gamma";
               break; }
  default :  { cout << "Other";
               break; }
}
```
6. 
```
do
  { cout << "What is your first name? ";
    cin >> name;
  } while ((name[0] < 'A') || (name[0] > 'Z'));
```

## Chapter 16

1. True
2. `main()`

3. Several smaller functions are better because each function can perform a single task.
4. Function names always end with a pair of parentheses.
5. By putting separating comments between functions.
6. The function `sq_25()` cannot be nested in `calc_it()`.
7. A function definition (a prototype).
8. True

## Chapter 17

1. True
2. Local variables are passed as arguments.
3. False
4. The variable data types
5. Static
6. You should never pass global variables—they do not need to be passed.
7. Two arguments (the string `"The rain has fallen %d inches"`, and the variable, `rainf`)

## Chapter 18

1. Arrays
2. Nonarray variables are always passed by value, unless you override the default with `&` before each variable name.
3. True
4. No
5. Yes

6. The data types of variables `x`, `y`, and `z` are not declared in the receiving parameter list.
7. c

## Chapter 19

1. By putting the return type to the left of the function name.
2. One
3. To prototype built-in functions.
4. `int`
5. False
6. Prototypes ensure that the correct number of parameters is being passed.
7. Global variables are already known across functions.
8. The return type is `float`. Three parameters are passed: a character, an integer, and a floating-point variable.

## Chapter 20

1. In the function prototypes.
2. Overloaded functions
3. Overloaded functions
4. False. You can specify multiple default arguments.
5. `void my_fun(float x, int i=7, char ch='A');`
6. False. Overloaded functions must differ in their argument lists, not only in their return values.

## Chapter 21

1. For portability between different computers
2. False. The standard output can be redirected to any device through the operating system.
3. `getch()` assumes `stdin` for the input device.
4. `get`
5. `>` and `<`
6. `getche()`
7. False. The input from `get` goes to a buffer as you type it.
8. Enter
9. True

## Chapter 22

1. The character-testing functions do not change the character passed to them.
2. `gets()` and `fgets()`
3. `floor()` rounds down and `ceil()` rounds up.
4. The function returns `0` (false) because `islower('s')` returns a `1` (true) and `isalpha(1)` is `0`.
5. `PeterParker`
6. `8 9`
7. True
8. `Prog` with a null zero at the end.
9. True

## Chapter 23

1. False
2. The array subscripts differentiate array elements.
3. C does not initialize arrays for you.
4. `0`
5. Yes. All arrays are passed by address because an array name is nothing more than an address to that array.
6. C++ initializes all types of global variables (and every other static variable in your program) to zero or null zero.

## Chapter 24

1. False
2. From the low numbers floating to the top of the array like bubbles.
3. Ascending order
4. The name of an array is an address to the starting element of that array.
5. a. `Eagles`
   b. `Rams`
   c. `les`
   d. `E`
   e. `E`
   f. The statement prints the character string, `s`.
   g. The third letter of “Eagles” (`g`) prints.

# Chapter 25

1. `int scores[5][6];`
2. `char initials[4][10][20]`
3. The first subscript represents rows and the last represents columns.
4. 30 elements
5. a. 2

   b. 1

   c. 91

   d. 8
6. Nested `for` loops step through multidimensional tables very easily.
7. a. 78

   b. 100

   c. 90

# Chapter 26

1. a. Integer pointer

   b. Character pointer

   c. Floating-point pointer
2. "Address of "
3. The `*` operator
4. `pt_sal = &salary;`
5. False
6. Yes
7. a. 2313.54

b. 2313.54

c. invalid

d. invalid

8. b

## Chapter 27

1. Array names are pointer constants, not pointer variables.
2. 8
3. *a*, *c*, and *d* are equivalent. Parentheses are needed around `iptr+4` and `iptr+1` to make *b* and *e* valid.
4. You have to move only pointers, not entire strings.
5. a and d

## Chapter 28

1. Structures hold groups of more than one value, each of which can be a different data type.
2. Members
3. At declaration time and at runtime
4. Structures pass by copy.
5. False. Memory is reserved only when structure variables are declared.
6. Globally
7. Locally
8. 4

## Chapter 29

1. True
2. Arrays are easier to manage.
3. a. `inventory[32].price = 12.33;`

   b. `inventory[11].part_no[0] = 'X';`

   c. `inventory[96] = inventory[62];`
4. a. `item` is not a structure variable.

   b. `inventory` is an array and must have a subscript.

   c. `inventory` is an array and must have a subscript.

## Chapter 30

1. Write, append, and read.
2. Disks hold more data than memory.
3. You can access sequential files only in the same order that they were originally written.
4. An error condition occurs.
5. The old file is overwritten.
6. The file is created.
7. C++ returns an end-of-file condition.

## Chapter 31

1. Records are stored in files and structures are stored in memory.
2. False
3. The file pointer continually updates to point to the next byte to read.

4. `read()` and `write()`
5. The `open()` function cannot be called without a filename.

## Chapter 32

1. Data members and member functions
2. No
3. No
4. Private
5. Declare it with the `public` keyword.

# ASCII Table

## (Including IBM Extended Character Codes)

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 000 | 00 | 0000 0000 | null |
| 001 | 01 | 0000 0001 | ☺ |
| 002 | 02 | 0000 0010 | ☻ |
| 003 | 03 | 0000 0011 | ♥ |
| 004 | 04 | 0000 0100 | ♦ |
| 005 | 05 | 0000 0101 | ♣ |
| 006 | 06 | 0000 0110 | ♠ |
| 007 | 07 | 0000 0111 | • |
| 008 | 08 | 0000 1000 | ◘ |
| 009 | 09 | 0000 1001 | ○ |
| 010 | 0A | 0000 1010 | ◙ |
| 011 | 0B | 0000 1011 | ♂ |
| 012 | 0C | 0000 1100 | ♀ |
| 013 | 0D | 0000 1101 | ♪ |
| 014 | 0E | 0000 1110 | ♫ |
| 015 | 0F | 0000 1111 | ☼ |
| 016 | 10 | 0001 0000 | ► |

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 017 | 11 | 0001 0001 | ◄ |
| 018 | 12 | 0001 0010 | ↕ |
| 019 | 13 | 0001 0011 | !! |
| 020 | 14 | 0001 0100 | ¶ |
| 021 | 15 | 0001 0101 | § |
| 022 | 16 | 0001 0110 | – |
| 023 | 17 | 0001 0111 | ↨ |
| 024 | 18 | 0001 1000 | ↑ |
| 025 | 19 | 0001 1001 | ↓ |
| 026 | 1A | 0001 1010 | → |
| 027 | 1B | 0001 1011 | ← |
| 028 | 1C | 0001 1100 | FS |
| 029 | 1D | 0001 1101 | GS |
| 030 | 1E | 0001 1110 | RS |
| 031 | 1F | 0001 1111 | US |
| 032 | 20 | 0010 0000 | SP |
| 033 | 21 | 0010 0001 | ! |
| 034 | 22 | 0010 0010 | " |
| 035 | 23 | 0010 0011 | # |
| 036 | 24 | 0010 0100 | $ |
| 037 | 25 | 0010 0101 | % |
| 038 | 26 | 0010 0110 | & |
| 039 | 27 | 0010 0111 | ' |
| 040 | 28 | 0010 1000 | ( |
| 041 | 29 | 0010 1001 | ) |
| 042 | 2A | 0010 1010 | * |
| 043 | 2B | 0010 1011 | + |
| 044 | 2C | 0010 1100 | ’ |
| 045 | 2D | 0010 1101 | - |
| 046 | 2E | 0010 1110 | . |
| 047 | 2F | 0010 1111 | / |

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 048 | 30 | 0011 0000 | 0 |
| 049 | 31 | 0011 0001 | 1 |
| 050 | 32 | 0011 0010 | 2 |
| 051 | 33 | 0011 0011 | 3 |
| 052 | 34 | 0011 0100 | 4 |
| 053 | 35 | 0011 0101 | 5 |
| 054 | 36 | 0011 0110 | 6 |
| 055 | 37 | 0011 0111 | 7 |
| 056 | 38 | 0011 1000 | 8 |
| 057 | 39 | 0011 1001 | 9 |
| 058 | 3A | 0011 1010 | : |
| 059 | 3B | 0011 1011 | ; |
| 060 | 3C | 0011 1100 | < |
| 061 | 3D | 0011 1101 | = |
| 062 | 3E | 0011 1110 | > |
| 063 | 3F | 0011 1111 | ? |
| 064 | 40 | 0100 0000 | @ |
| 065 | 41 | 0100 0001 | A |
| 066 | 42 | 0100 0010 | B |
| 067 | 43 | 0100 0011 | C |
| 068 | 44 | 0100 0100 | D |
| 069 | 45 | 0100 0101 | E |
| 070 | 46 | 0100 0110 | F |
| 071 | 47 | 0100 0111 | G |
| 072 | 48 | 0100 1000 | H |
| 073 | 49 | 0100 1001 | I |
| 074 | 4A | 0100 1010 | J |
| 075 | 4B | 0100 1011 | K |
| 076 | 4C | 0100 1100 | L |
| 077 | 4D | 0100 1101 | M |
| 078 | 4E | 0100 1110 | N |

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 079 | 4F | 0100 1111 | O |
| 080 | 50 | 0101 0000 | P |
| 081 | 51 | 0101 0001 | Q |
| 082 | 52 | 0101 0010 | R |
| 083 | 53 | 0101 0011 | S |
| 084 | 54 | 0101 0100 | T |
| 085 | 55 | 0101 0101 | U |
| 086 | 56 | 0101 0110 | V |
| 087 | 57 | 0101 0111 | W |
| 088 | 58 | 0101 1000 | X |
| 089 | 59 | 0101 1001 | Y |
| 090 | 5A | 0101 1010 | Z |
| 091 | 5B | 0101 1011 | [ |
| 092 | 5C | 0101 1100 | \ |
| 093 | 5D | 0101 1101 | ] |
| 094 | 5E | 0101 1110 | ^ |
| 095 | 5F | 0101 1111 | – |
| 096 | 60 | 0110 0000 | ` |
| 097 | 61 | 0110 0001 | a |
| 098 | 62 | 0110 0010 | b |
| 099 | 63 | 0110 0011 | c |
| 100 | 64 | 0110 0100 | d |
| 101 | 65 | 0110 0101 | e |
| 102 | 66 | 0110 0110 | f |
| 103 | 67 | 0110 0111 | g |
| 104 | 68 | 0110 1000 | h |
| 105 | 69 | 0110 1001 | i |
| 106 | 6A | 0110 1010 | j |
| 107 | 6B | 0110 1011 | k |
| 108 | 6C | 0110 1100 | l |
| 109 | 6D | 0110 1101 | m |

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 110 | 6E | 0110 1110 | n |
| 111 | 6F | 0110 1111 | o |
| 112 | 70 | 0111 0000 | p |
| 113 | 71 | 0111 0001 | q |
| 114 | 72 | 0111 0010 | r |
| 115 | 73 | 0111 0011 | s |
| 116 | 74 | 0111 0100 | t |
| 117 | 75 | 0111 0101 | u |
| 118 | 76 | 0111 0110 | v |
| 119 | 77 | 0111 0111 | w |
| 120 | 78 | 0111 1000 | x |
| 121 | 79 | 0111 1001 | y |
| 122 | 7A | 0111 1010 | z |
| 123 | 7B | 0111 1011 | { |
| 124 | 7C | 0111 1100 | \| |
| 125 | 7D | 0111 1101 | } |
| 126 | 7E | 0111 1110 | ~ |
| 127 | 7F | 0111 1111 | DEL |
| 128 | 80 | 1000 0000 | Ç |
| 129 | 81 | 1000 0001 | ü |
| 130 | 82 | 1000 0010 | é |
| 131 | 83 | 1000 0011 | â |
| 132 | 84 | 1000 0100 | ä |
| 133 | 85 | 1000 0101 | à |
| 134 | 86 | 1000 0110 | å |
| 135 | 87 | 1000 0111 | ç |
| 136 | 88 | 1000 1000 | ê |
| 137 | 89 | 1000 1001 | ë |
| 138 | 8A | 1000 1010 | è |
| 139 | 8B | 1000 1011 | ï |
| 140 | 8C | 1000 1100 | î |

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 141 | 8D | 1000 1101 | ì |
| 142 | 8E | 1000 1110 | Ä |
| 143 | 8F | 1000 1111 | Å |
| 144 | 90 | 1001 0000 | É |
| 145 | 91 | 1001 0001 | æ |
| 146 | 92 | 1001 0010 | Æ |
| 147 | 93 | 1001 0011 | ô |
| 148 | 94 | 1001 0100 | ö |
| 149 | 95 | 1001 0101 | ò |
| 150 | 96 | 1001 0110 | û |
| 151 | 97 | 1001 0111 | ù |
| 152 | 98 | 1001 1000 | ÿ |
| 153 | 99 | 1001 1001 | Ö |
| 154 | 9A | 1001 1010 | Ü |
| 155 | 9B | 1001 1011 | ¢ |
| 156 | 9C | 1001 1100 | £ |
| 157 | 9D | 1001 1101 | ¥ |
| 158 | 9E | 1001 1110 | $P_t$ |
| 159 | 9F | 1001 1111 | *f* |
| 160 | A0 | 1010 0000 | á |
| 161 | A1 | 1010 0001 | í |
| 162 | A2 | 1010 0010 | ó |
| 163 | A3 | 1010 0011 | ú |
| 164 | A4 | 1010 0100 | ñ |
| 165 | A5 | 1010 0101 | Ñ |
| 166 | A6 | 1010 0110 | a |
| 167 | A7 | 1010 0111 | o |
| 168 | A8 | 1010 1000 | ® |
| 169 | A9 | 1010 1001 | © |
| 170 | AA | 1010 1010 | ø |
| 171 | AB | 1010 1011 | ´ |

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 172 | AC | 1010 1100 | ¨ |
| 173 | AD | 1010 1101 | ≠ |
| 174 | AE | 1010 1110 | Æ |
| 175 | AF | 1010 1111 | Ø |
| 176 | B0 | 1011 0000 | ∞ |
| 177 | B1 | 1011 0001 | ± |
| 178 | B2 | 1011 0010 | ≤ |
| 179 | B3 | 1011 0011 | ≥ |
| 180 | B4 | 1011 0100 | ¥ |
| 181 | B5 | 1011 0101 | µ |
| 182 | B6 | 1011 0110 | ∂ |
| 183 | B7 | 1011 0111 | ∑ |
| 184 | B8 | 1011 1000 | ∏ |
| 185 | B9 | 1011 1001 | π |
| 186 | BA | 1011 1010 | ∫ |
| 187 | BB | 1011 1011 | ª |
| 188 | BC | 1011 1100 | º |
| 189 | BD | 1011 1101 | Ω |
| 190 | BE | 1011 1110 | æ |
| 191 | BF | 1011 1111 | ™ |
| 192 | C0 | 1100 0000 | L |
| 193 | C1 | 1100 0001 | ¡ |
| 194 | C2 | 1100 0010 | ¬ |
| 195 | C3 | 1100 0011 | √ |
| 196 | C4 | 1100 0100 | ƒ |
| 197 | C5 | 1100 0101 | + |
| 198 | C6 | 1100 0110 | Δ |
| 199 | C7 | 1100 0111 | « |
| 200 | C8 | 1100 1000 | » |
| 201 | C9 | 1100 1001 | … |
| 202 | CA | 1100 1010 | |

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 203 | CB | 1100 1011 | ╦ |
| 204 | CC | 1100 1100 | ╠ |
| 205 | CD | 1100 1101 | = |
| 206 | CE | 1100 1110 | Œ |
| 207 | CF | 1100 1111 | œ |
| 208 | D0 | 1101 0000 | – |
| 209 | D1 | 1101 0001 | — |
| 210 | D2 | 1101 0010 | “ |
| 211 | D3 | 1101 0011 | ” |
| 212 | D4 | 1101 0100 | ‘ |
| 213 | D5 | 1101 0101 | ’ |
| 214 | D6 | 1101 0110 | ÷ |
| 215 | D7 | 1101 0111 | ◊ |
| 216 | D8 | 1101 1000 | ÿ |
| 217 | D9 | 1101 1001 | Ÿ |
| 218 | DA | 1101 1010 | ⁄ |
| 219 | DB | 1101 1011 | ¤ |
| 220 | DC | 1101 1100 | ‹ |
| 221 | DD | 1101 1101 | › |
| 222 | DE | 1101 1110 | fi |
| 223 | DF | 1101 1111 | fl |
| 224 | E0 | 1110 0000 | α |
| 225 | E1 | 1110 0001 | β |
| 226 | E2 | 1110 0010 | Γ |
| 227 | E3 | 1110 0011 | π |
| 228 | E4 | 1110 0100 | ‰ |
| 229 | E5 | 1110 0101 | Â |
| 230 | E6 | 1110 0110 | µ |
| 231 | E7 | 1110 0111 | τ |
| 232 | E8 | 1110 1000 | Φ |
| 233 | E9 | 1110 1001 | θ |

| *Dec* $X_{10}$ | *Hex* $X_{16}$ | *Binary* $X_2$ | *ASCII Character* |
|---|---|---|---|
| 234 | EA | 1110 1010 | Í |
| 235 | EB | 1110 1011 | δ |
| 236 | EC | 1110 1100 | ∞ |
| 237 | ED | 1110 1101 | ø |
| 238 | EE | 1110 1110 | Ó |
| 239 | EF | 1110 1111 | ∩ |
| 240 | F0 | 1111 0000 |  |
| 241 | F1 | 1111 0001 | Ò |
| 242 | F2 | 1111 0010 | Ú |
| 243 | F3 | 1111 0011 | Û |
| 244 | F4 | 1111 0100 | Ù |
| 245 | F5 | 1111 0101 | ı |
| 246 | F6 | 1111 0110 | ÷ |
| 247 | F7 | 1111 0111 | ˜ |
| 248 | F8 | 1111 1000 | ° |
| 249 | F9 | 1111 1001 | • |
| 250 | FA | 1111 1010 | · |
| 251 | FB | 1111 1011 | √ |
| 252 | FC | 1111 1100 | η |
| 253 | FD | 1111 1101 | 2 |
| 254 | FE | 1111 1110 | ■ |
| 255 | FF | 1111 1111 |  |

# C++ Precedence Table

| *Precedence Level* | *Symbol* | *Description* | *Associativity* |
|---|---|---|---|
| 1 Highest | ( ) | Function call | Left to right |
| | [ ] | Array subscript | |
| | → | C++ indirect component selector | |
| | : : | C++ scope access / resolution | |
| | . | C++ direct component selector | |
| 2 Unary | ! | Logical negation | Right to left |
| | ~ | Bitwise (1's) complement | |
| | + | Unary plus | |
| | - | Unary minus | |

| *Precedence Level* | *Symbol* | *Description* | *Associativity* |
|---|---|---|---|
| | `++` | Preincrement or postincrement | |
| | `––` | Predecrement or postdecrement | |
| | `&` | Address of | |
| | `*` | Indirection | |
| | `sizeof` | (Returns size of operand, in bytes.) | |
| | `new` | (Dynamically allocates C++ storage.) | |
| | `delete` | (Dynamically deallocates C++ storage.) | |
| 3 Member Access | `.*` | C++ dereference | Left to right |
| | `→*` | C++ dereference | |
| 4 Multipli-cative | `*` | Multiply | Left to right |
| | `/` | Divide | |
| | `%` | Remainder (modulus) | |
| 5 Additive | `+` | Binary plus | Left to right |
| | `-` | Binary minus | |
| 6 Shift | `<<` | Shift left | Left to right |
| | `>>` | Shift right | |

| *Precedence Level* | *Symbol* | *Description* | *Associativity* |
|---|---|---|---|
| 7 Relational | < | Less than | Left to right |
| | <= | Less than or equal to | |
| | > | Greater than | |
| | >= | Greater than or equal to | |
| 8 Equality | == | Equal to | Left to right |
| | != | Not equal to | |
| 9 | & | Bitwise AND | Left to right |
| 10 | ^ | Bitwise XOR | Left to right |
| 11 | \| | Bitwise OR | Left to right |
| 12 | && | Logical AND | Left to right |
| 13 | \|\| | Logical OR | Left to right |
| 14 Condi-tional | ?: | | Right to left |
| 15 Assignment | = | Simple assignment | Right to left |
| | *= | Assign product | |
| | /= | Assign quotient | |

| *Precedence Level* | *Symbol* | *Description* | *Associativity* |
|---|---|---|---|
| | %= | Assign remainder | Right to left |
| | += | Assign sum | |
| | -= | Assign difference | |
| | &= | Assign bitwise AND | |
| | ^= | Assign bitwise XOR | |
| | \|= | Assign bitwise OR | |
| | <<= | Assign left shift | |
| | >>= | Assign right shift | |
| 16 Comma | , | Evaluate | Left to right |

# Keyword and Function Reference

These are the 46 C++ standard keywords:

| | | | |
|---|---|---|---|
| auto | double | new* | switch |
| asm* | else | operator* | template |
| break | enum | private* | this* |
| case | extern | protected | typedef |
| catch* | float | public* | union |
| char | for | register | unsigned |
| class* | friend* | return | virtual* |
| const | goto | short | void |
| continue | if | signed | volatile |
| default | inline* | sizeof | while |
| delete* | int | static | |
| do | long | struct | |

* These keywords are specific to C++. All others exist in both C and C++.

The following are the built-in function prototypes, listed by their header files. The prototypes describe the parameter data types that each function requires.

## stdio.h

```
int fclose(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fflush(FILE *stream);
int fgetc(FILE *stream);
char *fgets(char *, int, FILE *stream);
FILE *fopen(const char *filename, const char *mode);
int fprintf(FILE *stream, const char *format, ...);
int fputc(int, FILE *stream);
int fputs(const char *, FILE *stream);
size_t fread(void *, size_t, size_t, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fseek(FILE *stream, long offset, int origin);
size_t fwrite(const void *, size_t, size_t, FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *);
void perror(const char *);
int putc(int, FILE *stream);
int putchar(int);
int puts(const char *);
int remove(const char *filename);
void rewind(FILE *stream);
int scanf(const char *format, ...);
```

## ctype.h

```
int isalnum(unsigned char);
int asalpha(unsigned char);
int iscntrl(unsigned char);
int isdigit(unsigned char);
int isgraph(unsigned char);
int islower(unsigned char);
```

```
int isprint(unsigned char);
int ispunct(unsigned char);
int isspace(unsigned char);
int isupper(unsigned char);
int isxdigit(unsigned char);
int tolower(int);
int toupper(int);
```

## string.h

```
char *strcat(char *, char *);
int strcmp(char *, char *);
int strcpy(char *, char *);
size_t strlen(char *);
```

## math.h

```
double ceil(double);
double cos(double);
double exp(double);
double fabs(double);
double floor(double);
double fmod(double, double);
double log(double);
double log10(double);
double pow(double, double);
double sin(double);
double sqrt(double);
double tan(double);
```

## stdlib.h

```
double atof(const char *);
int atoi(const char *);
long atol(const char *);
void exit(int);
int rand(void);
void srand(unsigned int);
```

# The Mailing List Application

This appendix shows a complete program that contains most the commands and functions you learned in this book. This program manages a mailing list for your personal or business needs.

When you run the program, you are presented with a menu of choices that guides you through the program's operation. Comments throughout the program offer improvements you might want to make. As your knowledge and practice of C++ improve, you might want to expand this mailing list application into a complete database of contacts and relatives.

Here is the listing of the complete program:

```
// Filename: MAILING.CPP
// * Mailing List Application *
//   ------------------------
//
//   This program enables the user to enter, edit, maintain, and
//   print a mailing list of names and addresses.
//
//   All commands and concepts included in this program are
//   explained throughout the text of C++ By Example.
```

```
//
//
//
// These are items you might want to add or change:
// 1. Find your compiler's clear screen function to
//    improve upon the screen-clearing function.
// 2. Add an entry for the 'code' member to track different
//    types of names and addresses (i.e., business codes,
//    personal codes, etc.)
// 3. Search for a partial name (i.e., typing "Sm" finds
//    "Smith" and "Smitty" and "Smythe" in the file).
// 4. When searching for name matches, ignore case (i.e.,
//    typing "smith" finds "Smith" in the file).
// 5. Print mailing labels on your printer.
// 6. Allow for sorting a listing of names and address by name
//    or ZIP code.


// Header files used by the program:

#include <conio.h>
#include <ctype.h>
#include <fstream.h>
#include <iostream.h>
#include <string.h>

const char FILENAME[] = "ADDRESS.DAT";

// Prototype all of this program's functions.

char get_answer(void);
void disp_menu (void);
void clear_sc (void);
void change_na (void);
void print_na (void);
void err_msg (char err_msg[ ]);
void pause_sc (void);

const int NAME_SIZE = 25;
const int ADDRESS_SIZE = 25;
const int CITY_SIZE = 12;
```

```
const int STATE_SIZE = 3;
const int ZIPCODE_SIZE = 6;
const int CODE_SIZE = 7;

// Class of a name and address
class     Mail
{
private:
   char name[NAME_SIZE]; // Name stored here, should
                         // be Last, First order
   char address[ADDRESS_SIZE];
   char city[CITY_SIZE];
   char state[STATE_SIZE];    // Save room for null zero.
   char zipcode[ZIPCODE_SIZE];
   char code[CODE_SIZE];   // For additional expansion.  You
                           // might want to use this member
                           // for customer codes, vendor codes,
                           // or holiday card codes.
public:
   void pr_data(Mail *item)
   {
   // Prints the name and address sent to it.
      cout << "\nName    : " << (*item).name << "\n";
      cout << "Address: " << (*item).address << "\n";
      cout << "City    : " << (*item).city << "\tState: "
           << (*item).state << "  Zipcode: " << (*item).zipcode
           << "\n";
   }

   void get_new_item(Mail *item)
   {
      Mail temp_item;   // Holds temporary changed input.

      cout << "\nEnter new name and address information below\n(Press the ";
      cout << "Enter key without typing data to retain old "
              "information)\n\n";
      cout << "What is the new name? ";
      cin.getline(temp_item.name, NAME_SIZE);
      if (strlen(temp_item.name)) // Only save new data if user
      { strcpy((*item).name, temp_item.name); } // types something.
      cout << "What is the address? ";
      cin.getline(temp_item.address, ADDRESS_SIZE);
```

```
        if (strlen(temp_item.address))
        {  strcpy((*item).address, temp_item.address); }
        cout << "What is the city? ";
        cin.getline(temp_item.city, CITY_SIZE);
        if (strlen(temp_item.city))
        {  strcpy((*item).city, temp_item.city); }
        cout << "What is the state? (2 letter abbreviation only) ";
        cin.getline(temp_item.state, STATE_SIZE);
        if (strlen(temp_item.state))
        { strcpy((*item).state, temp_item.state); }
        cout << "What is the ZIP code? ";
        cin.getline(temp_item.zipcode, ZIPCODE_SIZE);
        if (strlen(temp_item.zipcode))
        {  strcpy((*item).zipcode, temp_item.zipcode); }
        (*item).code[0] = 0;    // Null out the code member
                                // (unused here).
    }

    void add_to_file(Mail *item);
    void change_na(void);
    void enter_na(Mail *item);
    void getzip(Mail *item);
};

void Mail::change_na(void)
{
// This search function can be improved by using the
// code member to assign a unique code to each person on the
// list.  Names are difficult to search for since there are
// so many variations (such as Mc and Mac and St. and Saint).

    Mail item;
    fstream file;
    int ans;
    int s;   // Holds size of structure.
    int change_yes = 0; // Will become TRUE if user finds
    char test_name[25]; // a name to change.

    cout << "\nWhat is the name of the person you want to change? ";
    cin.getline(test_name, NAME_SIZE);
    s = sizeof(Mail); // To ensure fread() reads properly.
```

```
   file.open(FILENAME, ios::in | ios::out);
   if (!file)
   {
      err_msg("*** Read error - Ensure file exists before "
              "reading it ***");
      return;
   }
   do
   {
      file.read((unsigned char *)&item, sizeof(Mail));
      if (file.gcount() != s)
      {
         if (file.eof())
         { break; }
      }
      if (strcmp(item.name, test_name) == 0)
      {
         item.pr_data(&item);   // Print name and address.
         cout << "\nIs this the name and address to " <<
                 "change? (Y/N) ";
         ans = get_answer();
         if (toupper(ans) == 'N')
         {  break;  } // Get another name.
         get_new_item(&item);  // Enable user to type new
                               // information.
         file.seekg((long)-s, ios::cur);  // Back up a structure.
         file.write((const unsigned char *)(&item),
                    sizeof(Mail));   // Rewrite information.
         change_yes = 1;  // Changed flag.
         break; // Finished
      }
   }
   while (!file.eof());
   if (!change_yes)
   {  err_msg("*** End of file encountered before finding the name ***");}
}

void Mail::getzip(Mail *item) // Ensure that ZIP code
                              // is all digits.
{
   int ctr;
```

```
   int bad_zip;

   do
   {
      bad_zip = 0;
      cout << "What is the ZIP code? ";
      cin.getline((*item).zipcode, ZIPCODE_SIZE);
      for (ctr = 0; ctr < 5; ctr++)
      {
         if (isdigit((*item).zipcode[ ctr ]))
         { continue; }
         else
         {
            err_msg("*** The ZIP code must consist of digits only ***");
         bad_zip = 1;
         break;
         }
      }
   }
   while (bad_zip);
}

void Mail::add_to_file(Mail *item)
{
   ofstream file;

   file.open(FILENAME, ios::app);   // Open file in append mode.
   if (!file)
   {
      err_msg("*** Disk error - please check disk drive ***");
      return;
   }
   file.write((const unsigned char *)(item), sizeof(Mail));
                                    // Add structure to file.
}

void Mail::enter_na(Mail *item)
{
   char ans;
```

```
   do
   {
      cout << "\n\n\n\n\nWhat is the name? ";
      cin.getline((*item).name, NAME_SIZE);
      cout << "What is the address? ";
      cin.getline((*item).address, ADDRESS_SIZE);
      cout << "What is the city? ";
      cin.getline((*item).city, CITY_SIZE);
      cout << "What is the state? (2 letter abbreviation only)";
      cin.getline((*item).state, STATE_SIZE);
      getzip(item); // Ensure that ZIP code is all digits.
      strcpy((*item).code, " ");  // Null out the code member.
      add_to_file(item); // Write new information to disk file.
      cout << "\n\nDo you want to enter another name " <<
              "and address? (Y/N) ";
      ans = get_answer();
   }
   while (toupper(ans) == 'Y');
}

//***************************************************************

// Defined constants
// MAX is total number of names allowed in memory for
// reading mailing list.

const int MAX = 250;
const char BELL = '\x07';

//***************************************************************

int main(void)
{
   char ans;
   Mail item;

   do
   {
      disp_menu();  // Display the menu for the user.
      ans = get_answer();
      switch (ans)
      {
        case '1':
```

```
            item.enter_na(&item);
            break;
         case '2':
            item.change_na();
            break;
         case '3':
            print_na();
            break;
         case '4':
            break;
         default:
            err_msg("*** You have to enter 1 through 4 ***");
            break;
      }
   }
   while (ans != '4');
   return 0;
}

//************************************************************

void disp_menu(void) // Display the main menu of program.
{
   clear_sc(); // Clear the screen.
   cout << "\t\t*** Mailing List Manager ***\n";
   cout << "\t\t    --------------------\n\n\n\n";
   cout << "Do you want to:\n\n\n";
   cout << "\t1. Add names and addresses to the list\n\n\n";
   cout << "\t2. Change names and addresses in the list\n\n\n";
   cout << "\t3. Print names and addresses in the list\n\n\n";
   cout << "\t4. Exit this program\n\n\n";
   cout << "What is your choice? ";
}

//************************************************************

void clear_sc() // Clear the screen by sending 25 blank
                // lines to it.
{
   int ctr; // Counter for the 25 blank lines.

   for (ctr = 0; ctr < 25; ctr++)
```

```
   { cout << "\n"; }
}

//***************************************************************

void print_na(void)
{
   Mail item;
   ifstream file;
   int s;
   int linectr = 0;

   s = sizeof(Mail); // To ensure fread() reads properly.
   file.open(FILENAME);
   if (!file)
   {
      err_msg("*** Error - Ensure file exists before"
              "reading it ***");
      return;
   }
   do
   {
      file.read((signed char *)&item, s);
      if (file.gcount() != s)
      {
         if (file.eof()) // If EOF, quit reading.
         { break; }
      }
      if (linectr > 20) // Screen is full.
      {
         pause_sc();
         linectr = 0;
      }
      item.pr_data(&item); // Print the name and address.
      linectr += 4;
   }
   while (!file.eof());
   cout << "\n- End of list -";
   pause_sc(); // Give user a chance to see names
               // remaining on-screen.
}
```

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

void err_msg(char err_msg[ ])
{
   cout << "\n\n" << err_msg << BELL << "\n";
}

//************************************************************

void pause_sc()
{
   cout << "\nPress the Enter key to continue...";
   while (getch() != '\r')
   { ; }  // Wait for Enter key.
}

//************************************************************

char get_answer(void)
{
   char ans;

   ans = getch();
   while (kbhit())
   { getch(); }
   putch(ans);
   return ans;
}
```

# Glossary

**Address.** Each memory (RAM) location (each byte) has a unique address. The first address in memory is 0, the second RAM location's address is 1, and so on until the last RAM location (thousands of bytes later).

**ANSI.** *A*merican *N*ational *S*tandards *I*nstitute, the committee that approves computer standards.

**Argument.** The value sent to a function or procedure. This can be a constant or a variable and is enclosed inside parentheses.

**Array.** A list of variables, sometimes called a table of variables.

**Array of Structures.** A table of one or more structure variables.

**ASCII.** Acronym for *A*merican *S*tandard *C*ode for *I*nformation *I*nterchange.

**ASCII File.** A file containing characters that can be used by any program on most computers. Sometimes called a text file or an ASCII text file.

**AUTOEXEC.BAT.** A batch file in PCs that executes a series of commands whenever you start or reset the computer.

**Automatic Variables.** Local variables that lose their values when their block (the one in which they are defined) ends.

**Backup File.** A duplicate copy of a file that preserves your work in case you damage the original file. Files on a hard disk are commonly backed up on floppy disks or tapes.

**Binary.** A numbering system based on only two digits. The only valid digits in a binary system are 0 and 1. See also *Bit*.

**Binary zero.** Another name for null zero.

**Bit.** Binary digit, the smallest unit of storage on a computer. Each bit can have a value of 0 or 1, indicating the absence or presence of an electrical signal. See also *Binary*.

**Bit Mask.** A pattern of bits that changes other bits on and off to meet a certain logical condition.

**Bitwise Operators.** C++ operators that manipulate the binary representation of values.

**Block.** One or more statements treated as though they are a single statement. A block is always enclosed in braces, { and }.

**Boot.** To start a computer with the operating system software in place. You must boot your computer before using it.

**Bubble Sort.** A method of sorting data into ascending or descending order. See also *Quicksort, Shell Sort*.

**Bug.** An error in a program that prevents the program from running correctly. The term originated when a moth short-circuited a connection in one of the first computers, preventing the computer from working!

**Byte.** A basic unit of data storage and manipulation. A byte is equivalent to eight bits and can contain a value ranging from 0 to 255.

**Cathode Ray Tube (CRT).** The television-like screen, also called the *monitor*. It is one place to which the output of the computer can be sent.

**Central Processing Unit (CPU).** The controlling circuit responsible for operations in the computer. These operations generally include system timing, logical processing, and logical operations. It controls every operation of the computer system. On PCs, the central processing unit is called a microprocessor; it is stored on a single integrated circuit chip.

**Code.** A set of instructions written in a programming language. See *Source Code.*

**Comment.** A message in a program, ignored by the computer, that tells users what the program does.

**Compile.** Process of translating a program written in a programming language such as C++ into machine code that your computer understands.

**Class.** A C++ user-defined data type that consists of data members and member functions. Its members are private by default.

**Concatenation.** The process of attaching one string to the end of another or combining two or more strings into a longer string.

**Conditional Loop.** A series of C++ instructions that occurs a fixed number of times.

**Constant.** Data defined with the `const` keyword that do not change during a program run.

**Constructor Function.** The function executed when the program declares an instance of a class.

**CPU.** Central Processing Unit.

**CRT.** Cathode Ray Tube.

**Data.** Information stored in the computer as numbers, letters, and special symbols such as punctuation marks. This also refers to the characters you input into your program so the program can produce meaningful information.

**Data Member.** A data component of a class or structure.

**Data Processing.** What computers really do. They take data and manipulate it into meaningful output. The meaningful output is called information.

**Data Validation.** The process of testing the values entered in a program. Checking whether a number is negative or positive or simply ensuring that a number is in a certain range are two examples of data validation.

**Debug.** Process of locating an error (bug) in a program and removing it.

**Declaration.** A statement that declares the existence of a data object or function. A declaration reserves memory.

**Default.** A predefined action or command that the computer chooses unless you specify otherwise.

**Default Argument List.** A list of argument values, specified in a function's prototypes, that determine initial values of the arguments if no values are passed for those arguments.

**Definition.** A statement that defines the format of a data object or function. A definition reserves no memory.

**Demodulate.** To convert an analog signal into a digital signal for use by a computer. See also *Modulate.*

**Dereference.** The process of finding a value to which a pointer variable is pointing.

**Destructor.** The function called when a class instance goes out of scope.

**Determinate Loop.** A `for` loop that executes a fixed number of times.

**Digital Computer.** A term that comes from the fact that your computer operates on binary (*on* and *off*) digital impulses of electricity.

**Directory.** A list of files stored on a disk. Directories within existing directories are called *subdirectories.*

**Disk.** A round, flat magnetic storage medium. Floppy disks are made of flexible material and enclosed in 5 1/4-inch or 3 1/2-inch protective cases. Hard disks consist of a stack of rigid disks housed in a single unit. A disk is sometimes called *external memory.* Disk storage is nonvolatile. When you turn off your computer, the disk's contents do not go away.

**Disk Drive.** A device that reads and writes data to a floppy or hard disk.

**Diskettes.** Another name for the removable floppy disks.

**Display.** A screen or monitor.

**Display Adapter.** Located in the system unit, the display adapter determines the amount of *resolution* and the possible number of colors on-screen.

**DOS.** Disk Operating System.

**Dot-Matrix Printer.** One of the two most common PC printers. The *laser* printer is the other. A dot-matrix printer is inexpensive and fast; it uses a series of small dots to represent printed text and graphics.

**Element.** An individual variable in an array.

**Execute.** To run a program.

**Expanded Memory.** A tricky way of expanding your computer's memory capacity beyond the 640K barrier using a technique called bank switching. See also *Extended Memory*.

**Extended Memory.** RAM above 640K, usually installed directly on the motherboard of your PC. You cannot access this extra RAM without special programs. See also *Expanded Memory*.

**External Modem.** A modem that sits in a box outside your computer. See also *Internal Modem*.

**Field.** A member in a data record.

**File.** A collection of data stored as a single unit on a floppy or hard disk. Files always have a filename that identifies them.

**File Extension.** Used by PCs and consists of a period followed by up to three characters. The file extension follows the filename.

**Filename.** A unique name that identifies a file. Filenames can be up to eight characters long, and can have a period followed by an extension (normally three characters long).

**Fixed Disk.** See *Hard Disk*.

**Fixed-Length Records.** A record where each field takes the same amount of disk space, even if that field's data value does not fill the field.

**Floppy Disk.** See *Disk*.

**Format.** Process of creating a "map" on the disk that tells the operating system how the disk is structured. This process is how the operating system keeps track of where files are stored.

**Function.** A self-contained coding segment designed to do a specific task. All C++ programs must have at least one function called `main()`. Some functions are library routines that manipulate numbers, strings, and output.

**Function Keys.** The keys labeled F1 through F12 (some keyboards only have up to F10).

**Global Variables.** A variable that can be seen from (and used by) every statement in the program.

**Hard Copy.** The printout of a program (or its output). Also a safe backup copy for a program in case the disk is erased.

**Hard Disk.** Sometimes called *fixed disks*. These hold much more data and are many times faster than floppy disks. See *Disk*.

**Hardware.** The physical parts of the machine. Hardware has been defined as "anything you can kick."

**Header Files.** Files that contain prototypes of C++'s built-in functions.

**Hexadecimal.** A numbering system based on 16 elements. Digits are numbered 0 through F, as follows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

**Hierarchy of Operators.** See *Order of Operators*.

**Indeterminate Loop.** A loop that continues an indeterminate amount of times (unlike the `for` loop, which continues a known amount of times).

**Infinite Loop.** The never-ending repetition of a block of C++ statements.

**Information.** The meaningful product from a program. Data go into a program to produce meaningful output (information).

**Inline Function.** A function that compiles as inline code each time the function is called.

**Input.** The entry of data into a computer through a device such as the keyboard.

**Input-Process-Output.** This model is the foundation of everything that happens in your computer. Data are input, then processed by your program in the computer, and finally information is output.

**I/O.** Acronym for Input/Output.

**Integer Variable.** Variables that can hold integers.

**Internal Modem.** A modem that resides inside the system unit. See also *External Modem*.

**Kilobyte (K).** A unit of measurement that refers to 1,024 bytes.

**Laser Printer.** A type of printer that is faster, in general, than dot-matrix printers. Laser printer output is much sharper than that of a dot-matrix printer, because a laser beam actually burns toner ink into the paper. Laser printers are more expensive than dot-matrix printers.

**Least Significant Bit.** The rightmost bit of a byte. For example, a binary 00000111 would have a 1 as the least significant bit.

**Line Printer.** Another name for your printer.

**Link Editing.** The last step the C++ compiler performs when preparing your program for execution.

**Literal.** Data that remains the same during program execution.

**Local Variable.** A variable that can be seen from (and used by) only the block in which it is defined.

**Loop.** The repeated execution of one or more statements.

**Machine Language.** The series of binary digits that a microprocessor executes to perform individual tasks. People seldom (if ever) program in machine language. Instead, they program in assembly language, and an assembler translates their instructions into machine language.

**Main Module.** The first function of a modular program called `main()` that controls the execution of the other functions.

**Maintainability.** The computer industry's word for the ability to change and update programs written in a simple style.

**Manipulator.** A value used by a program to inform the stream to modify one of its modes.

**Math Operator.** A symbol used for addition, subtraction, multiplication, division, or other calculations.

**Megabyte (M).** In computer terminology, approximately a million bytes (1,048,576 bytes).

**Member.** A piece of a structure variable that holds a specific type of data, or a class variable that holds a specific type of data or a function acting on that data.

**Member Function.** A function defined inside a class.

**Memory.** Storage area inside the computer, used to temporarily store data. The computer's memory is erased when the power is off.

**Menu.** A list of commands or instructions displayed on-screen. These lists organize commands and make a program easier to use.

**Menu-Driven.** Describes a program that provides menus for choosing commands.

**Microchip.** A small wafer of silicon that holds computer components and occupies less space than a postage stamp.

**Microcomputer.** A small computer that can fit on a desktop, such as a PC. The microchip is the heart of the microcomputer. Microcomputers are much less expensive than their larger counterparts.

**Microprocessor.** The chip that does the calculations for PCs. Sometimes it is called the Central Processing Unit (CPU).

**Modem.** A piece of hardware that modulates and demodulates signals so your computer can communicate with other computers over telephone lines. See also *External Modems, Internal Modems.*

**Modular Programming.** The process of writing your programs in several modules rather than as one long program. By breaking a program into several smaller program-line routines, you can isolate problems better, write correct programs faster, and produce programs that are much easier to maintain.

**Modulate.** Before your computer can transmit data over a telephone line, the information to be sent must be converted (modulated) into analog signals. See also *Demodulate.*

**Modulus.** The integer remainder of division.

**Monitor.** The television-like screen that enables the computer to display information. It is an output device.

**Mouse.** A hand-held device that you move across the desktop to move an indicator, called a mouse pointer, across the screen. Used instead of the keyboard to select and move items (such as text or graphics), execute commands, and perform other tasks.

**MS-DOS.** An operating system for IBM and compatible PCs.

**Multidimensional Arrays.** Arrays with more than one dimension. As two-dimensional arrays, they are sometimes called tables or matrices, which have rows and columns.

**Nested Loop.** A loop within a loop.

**Null String.** An empty string with an initial character of null zero and with a length of 0.

**Null Zero.** The string-terminating character. All C++ string constants and strings stored in character arrays end in null zero. The ASCII value for the null zero is 0.

**Numeric Functions.** Library routines that work with numbers.

**Object.** C++ class members consisting of both data and member functions.

**Object Code.** A "halfway step" between source code and executable machine language. Object code consists mostly of machine language but is not directly executable by the computer. It must first be linked in order to resolve external references and address references. See also *Source Code, Machine Language.*

**Object-Oriented Programming.** A programming approach that treats data as objects capable of manipulating themselves.

**Operator.** An operator works on data and performs math calculations or changes data to other data types. Examples include the `+`, `-`, and `sizeof()` operators.

**Order of Operators.** Sometimes called the *hierarchy of operators* or the *precedence of operators*. It determines exactly how C++ computes formulas.

**Output Device.** Where the results of a program are output, such as the screen, the printer, or a disk file.

**Overloading.** The process of writing more than one function with the same name. The functions must differ in their argument lists so C++ can identify which one to call.

**Parallel Arrays.** Two arrays working side by side. Each element in each array corresponds to one in the other array.

**Parallel Port.** A connector used to plug a device such as a printer into the computer. Transferring data through a parallel port is much faster than transferring data through a serial port.

**Parameter.** A list of variables enclosed in parentheses that follow the name of a function or procedure. Parameters indicate the number and type of arguments that are sent to the function or procedure.

**Passing by Address.** Also called *passing by reference*. When an argument (a local variable) is passed by address, the variable's address in memory is sent to, and is assigned to, the receiving function's parameter list. (If more than one variable is passed by address, each of their addresses is sent to and assigned to the receiving function's parameters.) A change made to the parameter in the function also changes the value of the argument variable.

**Passing by Copy.** Another name for *passing by value*.

**Passing by Reference.** Another name for *passing by address*.

**Passing by Value.** By default, all C++ variable arguments are passed *by value*. When the value contained in a variable is passed to the parameter list of a receiving function, changes made to the parameter in the routine do not change the value of the argument variable. Also called *passing by copy*.

**Path.** The route the computer travels from the root directory to any subdirectories when locating a file. The path also refers to the subdirectories that MS-DOS examines when you type a command that requires it to find and access a file.

**Peripheral.** A device attached to the computer, such as a modem, disk drive, mouse, or printer.

**Personal Computer.** A microcomputer, also called a PC, which stands for *p*ersonal *c*omputer.

**Pointer.** A variable that holds the address of another variable.

**Precedence of Operators.** See *Order of Operators.*

**Preprocessor Directive.** A command, preceded by a `#`, that you place in your source code that directs the compiler to modify the source code in some fashion. The two most common preprocessor directives are `#define` and `#include`.

**Printer.** A device that prints data from the computer to paper.

**Private Class Member.** A class member inaccessible except to the class's member functions.

**Program.** A group of instructions that tells the computer what to do.

**Programming Language.** A set of rules for writing instructions for the computer. Popular programming languages include BASIC, C, Visual Basic, C++, and Pascal.

**Prototype.** The definition of a function; includes its name, return type, and parameter list.

**Public Class Member.** A class member accessible to any function.

**Quicksort.** A method of sorting data values into ascending or descending order (faster than a Bubble Sort.) See also *Bubble Sort, Shell Sort.*

**RAM.** Random-Access Memory.

**Random-Access File.** Records in a file that can be accessed in any order you want.

**Random-Access Memory.** Memory that your computer uses to temporarily store data and programs. RAM is measured in kilobytes and megabytes. Generally, the more RAM a computer has, the more powerful programs it can run.

**Read-Only Memory.** A permanent type of computer memory. It contains the BIOS (*B*asic *I*nput / *O*utput *S*ystem), a special chip used to provide instructions to the computer when you turn the computer on.

**Real Numbers.** Numbers that have decimal points and a fractional part to the right of the decimal.

**Record.** Individual rows in files.

**Relational Operators.** Operators that compare data; they tell how two variables or constants relate to each other. They tell you whether two variables are equal or not equal, or which one is less than or more than the other.

**ROM.** Read-Only Memory.

**Scientific Notation.** A shortcut method of representing numbers of extreme values.

**Sectors.** A pattern of pie-shaped wedges on a disk. Formatting creates a pattern of *tracks* and sectors where your data and programs are stored.

**Sequence Point/Comma Operator.** This operator ensures that statements are performed in a left-to-right sequence.

**Sequential File.** A file that has to be accessed one record at a time beginning with the first record.

**Serial Port.** A connector used to plug in serial devices, such as a modem or a mouse.

**Shell Sort.** A method of sorting values into ascending or descending order. Named after the inventor of this method. See also *Bubble Sort*, *Quicksort*.

**Single-Dimensional Arrays.** Arrays that have only one subscript. Single-dimensional arrays represent a list of values.

**Software.** The data and programs that interact with your hardware. The C++ language is an example of software.

**Sorting.** A method of putting data in a specific order (such as alphabetical or numerical order), even if that order is not the same order in which the elements were entered.

**Source Code.** The C++ language instructions, written by programmers, that the C++ compiler translates into object code. See also *Object Code.*

**Spaghetti Code.** Term used when there are too many `goto`s in a program. If a program branches all over the place, it is difficult to follow and trying to follow the logic resembles a "bowl of spaghetti."

**Static Variables.** Variables that do not lose their values when the block in which they are defined ends. See also *Automatic Variables.*

**Standard Input Device.** The target of each `cout` and output function. Normally the screen unless rerouted to another device at the operating system's prompt.

**Standard Output Device.** The target of each `cin` and input function. Normally the keyboard unless rerouted to another device at the operating system's prompt.

**Stream.** Literally, a stream of characters, one following another, flowing among devices in your computer.

**String.** One or more characters terminated with a null zero.

**String Constant.** One or more groups of characters that end in a null zero.

**String Delimiter.** See *Null Zero.*

**String Literal.** Another name for a *String Constant.*

**Structure.** A unit of related information containing one or more members, such as an employee number, employee name, employee address, employee pay rate, and so on.

**Subscript.** A number inside brackets that differentiates one *element* of an array from another.

**Syntax Error.** The most common error a programmer makes. Often a misspelled word.

**System Unit.** The large box component of the computer. The system unit houses the PC's microchip (the *CPU*).

**Timing Loop.** A loop used to delay the computer for a specific amount of time.

**Tracks.** A pattern of paths on a disk. Formatting creates a pattern of tracks and *sectors* where your data and programs go.

**Truncation.** The fractional part of a number (the part of the number to the right of the decimal point) is taken off the number. No rounding is done.

**Two's Complement.** A method your computer uses to take the negative of a number. This method, plus addition, allows the computer to simulate subtraction.

**Unary Operator.** The addition or subtraction operator used before a single variable or constant.

**User-Friendliness.** A program is user-friendly if it makes the user comfortable and simulates an atmosphere that the user is already familiar with.

**Variable.** Data that can change as the program runs.

**Variable-Length Records.** A record that takes up no wasted space on the disk. As soon as a field's data value is saved to the file, the next field's data value is stored after it. There is usually a special separating character between the fields so your programs know where the fields begin and end.

**Variable Scope.** Sometimes called the *visibility of variables*, this describes how variables are "seen" by your program. See also *Global Variables* and *Local Variables.*

**Volatile.** Temporary state of memory. For example, when you turn the computer off, all the RAM is erased.

**Word.** In PC usage, two consecutive bytes (16 bits) of data.

# Index

## Symbols

## A

## B

## C

## D

## E

## F

## I

## J–K



## L

## M

## N

## O

## P

## Q–R

## S

## T

## U



## V

## W



## X–Y–Z

# Order Your Disk Program Today!

You can save yourself hours of tedious, error-prone typing by ordering the companion disk to *C++ By Example*. The disk contains the source code for all the complete programs and many of the shorter samples in the book. Appendix F's complete mailing-list application is also included on the disk, as well as the answers to many of the review exercises.

You will get code that shows you how to use most the features of C++. Samples include code for keyboard control, screen control, file I/O, control statements, structures, pointers, and more.

Disks are available in 3 1/2-inch format (high density). The cost is $10 per disk. (When ordering outside the US, please add $5 for shipping and handling.)

Just make a copy of this page, fill in the blanks, and mail it with your check or money order to:

**C++ Disk**
Greg Perry
P.O. Box 35752
Tulsa, OK 74135-0752

Please **print** the following information:

Payment method: *Check*_____ *Money Order*_____

Number of Disks:_____@ $10.00 =_____

Name:_____

Street Address:_____

City:_____State:_____

ZIP:_____

(On foreign orders, please use a separate page to give your mailing address in the format required by your post office.)

Checks and money orders should be made payable to:

**Greg Perry**

*(This offer is made by Greg Perry, not by Que Corporation.)*

# Array Processing

C++ provides many ways to access arrays. If you have programmed in other computer languages, you will find that some of C++'s array indexing techniques are unique. Arrays in the C++ language are closely linked with *pointers*. Chapter 26, "Pointers," describes the many ways pointers and arrays interact. Because pointers are so powerful, and because learning about arrays provides a good foundation for learning about pointers, this chapter attempts to describe in detail how to reference arrays.

This chapter discusses the different types of array processing. You learn how to search an array for one or more values, find the highest and lowest values in an array, and sort an array into numerical or alphabetical order.

This chapter introduces the following concepts:

- Searching arrays
- Finding the highest and lowest values in arrays
- Sorting arrays
- Advanced subscripting with arrays

Many programmers see arrays as a turning point. Gaining an understanding of array processing makes your programs more accurate and allows for more powerful programming.

# Searching Arrays

Arrays are one of the primary means by which data is stored in C++ programs. Many types of programs lend themselves to processing lists (arrays) of data, such as an employee payroll program, a scientific research of several chemicals, or customer account processing. As mentioned in the previous chapter, array data usually is read from a disk file. Later chapters describe disk file processing. For now, you should understand how to manipulate arrays so you see the data exactly the way you want to see it.

Array elements do not always appear in the order in which they are needed.

Chapter 23, "Introducing Arrays," showed how to print arrays in the same order that you entered the data. This is sometimes done, but it is not always the most appropriate method of looking at data.

For instance, suppose a high school used C++ programs for its grade reports. Suppose also that the school wanted to see a list of the top 10 grade-point averages. You could not print the first 10 grade-point averages in the list of student averages because the top 10 GPAs might not (and probably will not) appear as the first 10 array elements. Because the GPAs would not be in any sequence, the program would have to sort the array into numeric order, from high GPAs to low, or else search the array for the 10 highest GPAs.

You need a method for putting arrays in a specific order. This is called *sorting* an array. When you sort an array, you put that array in a specific order, such as in alphabetical or numerical order. A dictionary is in sorted order, and so is a phone book.

When you reverse the order of a sort, it is called a *descending sort*. For instance, if you wanted to look at a list of all employees in descending salary order, the highest-paid employees would be printed first.

Figure 24.1 shows a list of eight numbers in an array called `unsorted`. The middle list of numbers is an ascending sorted version of `unsorted`. The third list of numbers is a descending version of `unsorted`.

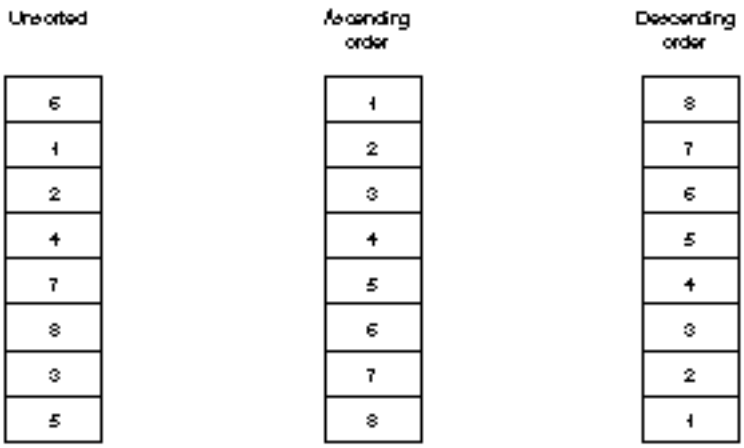



Figure 24.1. A list of unsorted numbers sorted into an ascending and a descending order.

Before you learn to sort, it would be helpful to learn how to search an array for a value. This is a preliminary step in learning to sort. What if one of those students received a grade change? The computer must be able to access that specific student's grade to change it (without affecting the others). As the next section shows, programs can search for specific array elements.

**NOTE:** C++ provides a method for sorting and searching lists of strings, but you will not understand how to do this until you learn about pointers, starting in Chapter 26, "Pointers." The sorting and searching examples and algorithms presented in this chapter demonstrate sorting and searching arrays of numbers. The same concepts will apply (and will actually be much more usable for "real-world" applications) when you learn how to store lists of names in C++.

# Searching for Values

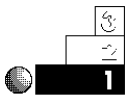
You do not have to know any new commands to search an array for a value. Basically, the `if` and `for` loop statements are all you need. To search an array for a specific value, look at each element in that array, and compare it to the `if` statement to see whether they match. If they do not, you keep searching down the array. If you run out of array elements before finding the value, it is not in the array.

You do not have to sort an array to find its extreme values.

You can perform several different kinds of searches. You might have to find the highest or the lowest value in a list of numbers. This is informative when you have much data and want to know the extremes of the data (such as the highest and lowest sales region in your division). You also can search an array to see whether it contains a matching value. For example, you can see whether an item is already in an inventory by searching a part number array for a match.

The following programs illustrate some of these array-searching techniques.

## Examples



1. To find the highest number in an array, compare each element with the first one. If you find a higher value, it becomes the basis for the rest of the array. Continue until you reach the end of the array and you will have the highest value, as the following program shows.

*Identify the program and include the I/O header file. You want to find the highest value in an array, so define the array size as a constant, then initialize the array.*

*Loop through the array, comparing each element to the highest value. If an element is higher than the highest value saved, store the element as the new high value. Print the highest value found in the array.*

```
// Filename: C24HIGH.CPP
// Finds the highest value in the array.
#include <iostream.h>
const int SIZE = 15;
void main()
```

```
{
   // Puts some numbers in the array.
   int ara[SIZE]={5,2,7,8,36,4,2,86,11,43,22,12,45,6,85};
   int high_val, ctr;

   high_val = ara[0];           // Initializes with first
                                        // array element.
   for (ctr=1; ctr<SIZE; ctr++)
     {                  // Stores current value if it is
                        // the higher than the highest.
       if (ara[ctr] > high_val)
         { high_val = ara[ctr]; }
     }

   cout << "The highest number in the list is "
        << high_val << "\n";
   return;
}
```

The output of the program is the following:

```
The highest number in the list is 86.
```

You have to save the element if and only if it is higher than the one you are comparing. Finding the smallest number in an array is just as easy, except that you determine whether each succeeding array element is less than the lowest value found so far.

2. The following example expands on the previous one by finding the highest and the lowest value. First, store the first array element in *both* the highest and the lowest variable to begin the search. This ensures that each element after that one is tested to see whether it is higher or lower than the first.

   This example also uses the `rand()` function from Chapter 22, "Character, String, and Numeric Functions," to fill the array with random values from 0 to 99 by applying the modulus operator (`%`) and 100 against whatever value `rand()` produces. The program prints the entire array before starting the search for the highest and the lowest.


```
// Filename: C24HILO.CPP
// Finds the highest and the lowest value in the array.
#include <iostream.h>
#include <stdlib.h>
const int SIZE = 15;
void main()
{
   int ara[SIZE];
   int high_val, low_val, ctr;

   // Fills array with random numbers from 0 to 99.
   for (ctr=0; ctr<SIZE; ctr++)
     { ara[ctr] = rand() % 100; }

   // Prints the array to the screen.
   cout << "Here are the " << SIZE << " random numbers:\n";
   for (ctr=0; ctr<SIZE; ctr++)
     {  cout << ara[ctr] << "\n"; }

   cout << "\n\n";       // Prints a blank line.
   high_val = ara[0];    // Initializes first element to
                         // both high and low.
   low_val  = ara[0];

   for (ctr=1; ctr<SIZE; ctr++)
     {                 // Stores current value if it is
                       // higher than the highest.
       if (ara[ctr] > high_val)
         { high_val = ara[ctr]; }
       if (ara[ctr] < low_val)
         { low_val = ara[ctr]; }
     }

   cout << "The highest number in the list is " <<
           high_val << "\n";
   cout << "The lowest number in the list is " <<
           low_val << "\n";
   return;
}
```

Here is the output from this program:

```
Here are the 15 random numbers:
46
30
82
90
56
17
95
15
48
26
4
58
71
79
92
The highest number in the list is 95
The lowest number in the list is 4
```

3. The next program fills an array with part numbers from an inventory. You must use your imagination, because the inventory array normally would fill more of the array, be initialized from a disk file, and be part of a larger set of arrays that hold descriptions, quantities, costs, selling prices, and so on. For this example, assignment statements initialize the array. The important idea from this program is not the array initialization, but the method for searching the array.

**NOTE:** If the newly entered part number is already on file, the program tells the user. Otherwise, the part number is added to the end of the array.

```
// Filename: C24SERCH.CPP
// Searches a part number array for the input value. If
```

```
// the entered part number is not in the array, it is
// added. If the part number is in the array, a message
// is printed.
#include <iostream.h>
const int MAX = 100;
void fill_parts(long int parts[MAX]);

void main()
{
   long int search_part;            // Holds user request.
   long int parts[MAX];
   int ctr;
   int num_parts=5;          // Beginning inventory count.

   fill_parts(parts);    // Fills the first five elements.
   do
   {
     cout << "\n\nPlease type a part number...";
     cout << "(-9999 ends program) ";
     cin >> search_part;
     if (search_part == -9999)
       { break; }             // Exits loop if user wants.
     // Scans array to see whether part is in inventory.
     for (ctr=0; ctr<num_parts; ctr++) // Checks each item.
     { if (search_part == parts[ctr])       // If it is in
                                            // inventory...
          { cout << "\nPart " << search_part <<
                    " is already in inventory";
            break;
          }
       else
        { if (ctr == (num_parts-1) )      // If not there,
                                          // adds it.
          { parts[num_parts] = search_part;   // Adds to
                                               // end of array.
            num_parts++;
            cout << search_part <<
                 " was added to inventory\n";
```

```
            break;
             }
         }
         }
   } while (search_part != -9999);      // Loops until user
                                        // signals end.
   return;
}

void fill_parts(long int parts[MAX])
{
   // Assigns five part numbers to array for testing.
   parts[0] = 12345;
   parts[1] = 24724;
   parts[2] = 54154;
   parts[3] = 73496;
   parts[4] = 83925;
   return;
}
```

Here is the output from this program:

```
Please type a part number...(-9999 ends program) 34234
34234 was added to inventory

Please type a part number...(-9999 ends program) 83925

Part 83925 is already in inventory

Please type a part number...(-9999 ends program) 52786
52786 was added to inventory

Please type a part number...(-9999 ends program) -9999
```

## Sorting Arrays

There are many times when you must sort one or more arrays. Suppose you were to take a list of numbers, write each number on a separate piece of paper, and throw all the pieces of paper into the air. The steps you take—shuffling and changing the order of the

pieces of paper and trying to put them in order—are similar to what your computer goes through to sort numbers or character data.

Because sorting arrays requires exchanging values of elements back and forth, it helps if you first learn the technique for swapping variables. Suppose you had two variables named `score1` and `score2`. What if you wanted to reverse their values (putting `score2` into the `score1` variable, and vice versa)? You could not do this:

```
score1 = score2;   // Does not swap the two values.
score2 = score1;
```

Why doesn't this work? In the first line, the value of `score1` is replaced with `score2`'s value. When the first line finishes, both `score1` and `score2` contain the same value. Therefore, the second line cannot work as desired.

To swap two variables, you have to use a third variable to hold the intermediate result. (This is the only function of this third variable.) For instance, to swap `score1` and `score2`, use a third variable (called `hold_score` in this code), as in

```
hold_score = score1;   // These three lines properly
score1 = score2;       // swap score1 and score2.
score2 = hold_score;
```

This exchanges the values in the two variables.

There are several different ways to sort arrays. These methods include the *bubble sort*, the *quicksort*, and the *shell sort*. The basic goal of each method is to compare each array element to another array element and swap them if the higher value is less than the other.

The lowest values in a list "float" to the top with the bubble sort algorithm.

The theory behind these sorts is beyond the scope of this book, however, the bubble sort is one of the easiest to understand. Values in the array are compared to each other, a pair at a time, and swapped if they are not in back-to-back order. The lowest value eventually "floats" to the top of the array, like a bubble in a glass of soda.

Figure 24.2 shows a list of numbers before, during, and after a bubble sort. The bubble sort steps through the array and compares pairs of numbers to determine whether they have to be swapped. Several passes might have to be made through the array before it is

finally sorted (no more passes are needed). Other types of sorts improve on the bubble sort. The bubble sort procedure is easy to program, but it is slower compared to many of the other methods.

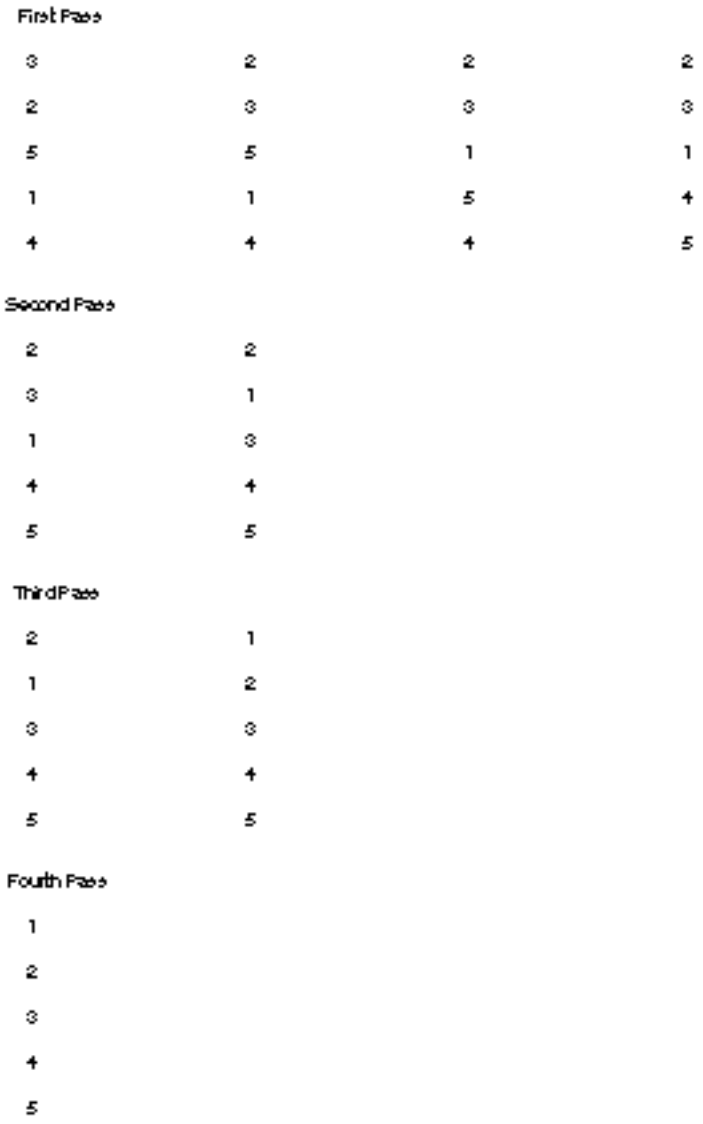



Figure 24.2. Sorting a list of numbers using the bubble sort.

The following programs show the bubble sort in action.

## Examples

1. The following program assigns 10 random numbers between 0 and 99 to an array, then sorts the array. A nested `for` loop is perfect for sorting numbers in the array (as shown in the `sort_array()` function). Nested `for` loops provide a nice mechanism for working on pairs of values, swapping them if needed. As the outside loop counts down the list, referencing each element, the inside loop compares each of the remaining values to those array elements.

```
// Filename: C24SORT1.CPP
// Sorts and prints a list of numbers.
const int MAX = 10;
#include <iostream.h>
#include <stdlib.h>
void fill_array(int ara[MAX]);
void print_array(int ara[MAX]);
void sort_array(int ara[MAX]);

void main()
{
   int ara[MAX];

   fill_array(ara);    // Puts random numbers in the array.

   cout << "Here are the unsorted numbers:\n";
   print_array(ara);   // Prints the unsorted array.

   sort_array(ara);    // Sorts the array.

   cout << "\n\nHere are the sorted numbers:\n";
   print_array(ara);   // Prints the newly sorted array.
   return;
}

void fill_array(int ara[MAX])
{
```

```
   // Puts random numbers in the array.
   int ctr;
   for (ctr=0; ctr<MAX; ctr++)
      { ara[ctr] = (rand() % 100); }   // Forces number to
                                       // 0-99 range.
   return;
}

void print_array(int ara[MAX])
{
   // Prints the array.
   int ctr;
   for (ctr=0; ctr<MAX; ctr++)
      { cout << ara[ctr] << "\n"; }
   return;
}

void sort_array(int ara[MAX])
{
   // Sorts the array.
   int temp;            // Temporary variable to swap with
   int ctr1, ctr2;            // Need two loop counters to
                              //   swap pairs of numbers.
   for (ctr1=0; ctr1<(MAX-1); ctr1++)
      { for (ctr2=(ctr1+1); ctr2<MAX; ctr2++) // Test pairs.
           { if (ara[ctr1] > ara[ctr2])    // Swap if this
               { temp = ara[ctr1]; // pair is not in order.
                 ara[ctr1] = ara[ctr2];
                 ara[ctr2] = temp;   // "Float" the lowest
                                     // to the highest.
               }
           }
      }
   return;
}
```

The output from this program appears next. If any two randomly generated numbers were the same, the bubble sort would work properly, placing them next to each other in the list.

```
Here are the unsorted numbers:
46
30
82
90
56
17
95
15
48
26


Here are the sorted numbers:
15
17
26
30
46
48
56
82
90
95
```

To produce a descending sort, use the less-than (<) logical operator when swapping array elements.

2. The following program is just like the previous one, except it prints the list of numbers in descending order.

   A descending sort is as easy to write as an ascending sort. With the ascending sort (from low to high values), you compare pairs of values, testing to see whether the first is greater than the second. With a descending sort, you test to see whether the first is less than the second one.

```
// Filename: C24SORT2.CPP
// Sorts and prints a list of numbers in reverse
// and descending order.
const int MAX = 10;
#include <iostream.h>
#include <stdlib.h>
void fill_array(int ara[MAX]);
```

```
void print_array(int ara[MAX]);
void sort_array(int ara[MAX]);

void main()
{
   int ara[MAX];

   fill_array(ara);    // Puts random numbers in the array.

   cout << "Here are the unsorted numbers:\n";
   print_array(ara);          // Prints the unsorted array.

   sort_array(ara);                    // Sorts the array.

   cout << "\n\nHere are the sorted numbers:\n";
   print_array(ara);      // Prints the newly sorted array.
   return;
}

void fill_array(int ara[MAX])
{
   // Puts random numbers in the array.
   int ctr;
   for (ctr=0; ctr<MAX; ctr++)
      { ara[ctr] = (rand() % 100); }      // Forces number
                                          // to 0-99 range.
   return;
}

void print_array(int ara[MAX])
{
   // Prints the array
   int ctr;
   for (ctr=0; ctr<MAX; ctr++)
      { cout << ara[ctr] << "\n"; }
   return;
}

void sort_array(int ara[MAX])
{
   // Sorts the array.
   int temp;           // Temporary variable to swap with.
```

```
   int ctr1, ctr2;           // Need two loop counters
                             //   to swap pairs of numbers.
   for (ctr1=0; ctr1<(MAX-1); ctr1++)
      { for (ctr2=(ctr1+1); ctr2<MAX; ctr2++) // Test pairs
           // Notice the difference in descending (here)
           // and ascending.
           { if (ara[ctr1] < ara[ctr2]) // Swap if this
               { temp = ara[ctr1]; // pair is not in order.
                 ara[ctr1] = ara[ctr2];
                 ara[ctr2] = temp;  // "Float" the lowest
                                    // to the highest.
               }
            }
      }
   return;
}
```

**TIP:** You can save the previous programs' sort functions in two separate files named `sort_ascend` and `sort_descend`. When you must sort two different arrays, `#include` these files inside your own programs. Even better, compile each of these routines separately and link the one you need to your program. (You must check your compiler's manual to learn how to do this.)

You can sort character arrays just as easily as you sort numeric arrays. C++ uses the ASCII character set for its sorting comparisons. If you look at the ASCII table in Appendix C, you will see that numbers sort before letters and that uppercase letters sort before lowercase letters.

## Advanced Referencing of Arrays

The array notation you have seen so far is common in computer programming languages. Most languages use subscripts inside brackets (or parentheses) to refer to individual array elements. For instance, you know the following array references describe the first

and fifth element of the array called sales (remember that the starting subscript is always 0):

```
sales[0]
sales[4]
```

C++ provides another approach to referencing arrays. Even though the title of this section includes the word "advanced," this array-referencing method is not difficult. It is very different, however, especially if you are familiar with another programming language's approach.

There is nothing wrong with referring to array elements in the manner you have seen so far, however, the second approach, unique to C and C++, will be helpful when you learn about pointers in upcoming chapters. Actually, C++ programmers who have programmed for several years rarely use the subscript notation you have seen.

An array name is the address of the starting element of the array.

In C++, an array's name is not just a label for you to use in programs. To C++, the array name is the actual address where the first element begins in memory. Suppose you define an array called amounts with the following statement:

```
int amounts[6] = {4, 1, 3, 7, 9, 2};
```

Figure 24.3 shows how this array is stored in memory. The figure shows the array beginning at address 405,332. (The actual addresses of variables are determined by the computer when you load and run your compiled program.) Notice that the name of the array, amounts, is located somewhere in memory and contains the address of amounts[0], or 405,332.

You can refer to an array by its regular subscript notation, or by modifying the address of the array. The following refer to the third element of amounts:

```
amounts[3] and (amounts + 3)[0]
```

Because C++ considers the array name to be an address in memory that contains the location of the first array element, nothing keeps you from using a different address as the starting address and referencing from there. Taking this one step further, each of the following also refers to the third element of amounts:

```
(amounts+0)[3] and (amounts+2)[1] and (amounts-2)[5]
(1+amounts)[2] and (3+amounts)[0] and (amounts+1)[2]
```

You can print any of these array elements with `cout`.

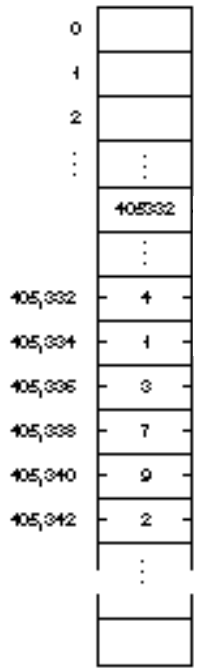



Figure 24.3. The array name `amounts` holds the address of `amounts[0]`.

When you print strings inside character arrays, referencing the arrays by their modified addresses is more useful than with integers. Suppose you stored three strings in a single character array. You could initialize this array with the following statement:

```
char names[]={'T','e','d','\0','E','v','a','\0','S','a','m','\0'};
```

Figure 24.4 shows how this array might look in memory. The array name, `names`, contains the address of the first element, `names[0]` (the letter *T*).

**CAUTION:** The hierarchy table in Appendix D, "C++ Precedence Table," shows that array subscripts have precedence over addition and subtraction. Therefore, you must enclose array names in parentheses if you want to modify the name as shown in these examples. The following are not equivalent:

`(2+amounts)[1]` and `2+amounts[1]`

The first example refers to `amounts[3]` (which is 7). The second example takes the value of `amounts[1]` (which is 1 in this example array) and adds 2 to it (resulting in a value of 3).

This second method of array referencing might seem like more trouble than it is worth, but learning to reference arrays in this fashion will make your transition to pointers much easier. An array name is actually a pointer, because the array contains the address of the first array element (it "points" to the start of the array).

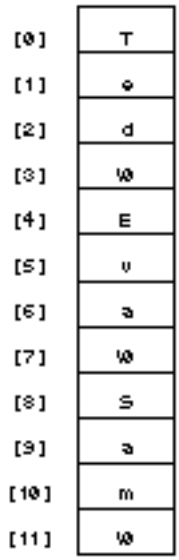



Figure 24.4. Storing more than one string in a single character array.

You have yet to see a character array that holds more than one string, but C++ allows it. The problem with such an array is how you reference, and especially how you print, the second and third strings. If you were to print this array using cout:

```
cout << names;
```

C++ would print the following:

```
Ted
```

cout prints strings in arrays starting at the array's address and continuing until it reaches the null zero.

Because cout requires a starting address, you can print the three strings with the following couts:

```
cout << names;       // Prints Ted
cout << (names+4);   // Prints Eva
cout << (names+8);   // Prints Sam
```

To test your understanding, what do the following couts print?

```
cout << (names+1);
cout << (names+6);
```

The first cout prints ed. The characters ed begin at (names+1) and the cout stops printing when it reaches the null zero. The second cout prints a. Adding six to the address at names produces the address where the a is located. The "string" is only one character long because the null zero appears in the array immediately after the a.

To sum up character arrays, the following refer to individual array elements (single characters):

```
names[2] and (names+1)[1]
```

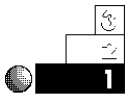
The following refer to addresses only, and as such, you can print the full strings with cout:

```
names and (names+4)
```

**CAUTION:** Never use the printf()'s %c control code to print an address reference, even if that address contains a character. Print strings by specifying an address with %s, and single characters by specifying the character element with %c.

The following examples are a little different from most you have seen. They do not perform "real-world" work, but were designed as study examples for you to familiarize yourself with this new method of array referencing. The next few chapters expand on these methods.

## Examples



1. The following program stores the numbers from 100 to 600 in an array, then prints elements using the new method of array subscripting.

```
// Filename: C24REF1.CPP
// Print elements of an integer array in different ways.
#include <iostream.h>
void main()
{
   int num[6] = {100, 200, 300, 400, 500, 600};

   cout << "num[0] is \t" << num[0] << "\n";
   cout << "(num+0)[0] is \t" << (num+0)[0] << "\n";
   cout << "(num-2)[2] is \t" << (num-2)[2] << "\n\n";

   cout << "num[1] is \t" << num[1] << "\n";
   cout << "(num+1)[0] is \t" << (num+1)[0] << "\n\n";

   cout << "num[5] is \t" << num[5] << "\n";
   cout << "(num+5)[0] is \t" << (num+5)[0] << "\n";
   cout << "(num+2)[3] is \t" << (num+2)[3] << "\n\n";

   cout << "(3+num)[1] is \t" << (3+num)[1] << "\n";
   cout << "3+num[1] is \t" << 3+num[1] << "\n";
   return;
}
```

Here is the output of this program:

```
num[0] is       100
(num+0)[0] is   100
(num-2)[2] is   100
```

```
num[1] is       200
(num+1)[0] is   200

num[5] is       600
(num+5)[0] is   600
(num+2)[3] is   600

(3+num)[1] is   500
3+num[1] is     203
```

2. The following program prints strings and characters from a character array. The `cout`s all print properly.

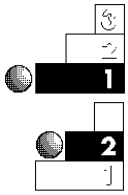
```
// Filename: C24REF2.CPP
// Prints elements and strings from an array.
#include <iostream.h>
void main()
{
   char names[]={'T','e','d','\0','E','v','a','\0',
                 'S', 'a','m','\0'};

   // Must use extra percent (%) to print %s and %c.
   cout << "names " << names << "\n";
   cout << "names+0 " << names+0 << "\n";
   cout << "names+1 " << names+1 << "\n";
   cout << "names+2 " << names+2 << "\n";
   cout << "names+3 " << names+3 << "\n";
   cout << "names+5 " << names+5 << "\n";
   cout << "names+8 " << names+8 << "\n\n";

   cout << "(names+0)[0] " << (names+0)[0] << "\n";
   cout << "(names+0)[1] " << (names+0)[1] << "\n";
   cout << "(names+0)[2] " << (names+0)[2] << "\n";
   cout << "(names+0)[3] " << (names+0)[3] << "\n";
   cout << "(names+0)[4] " << (names+0)[4] << "\n";
   cout << "(names+0)[5] " << (names+0)[5] << "\n\n";

   cout << "(names+2)[0] " << (names+2)[0] << "\n";
   cout << "(names+2)[1] " << (names+2)[1] << "\n";
   cout << "(names+1)[4] " << (names+1)[4] << "\n\n";

   return;
}
```

Study the output shown below by comparing it to the program. You will learn more about strings, characters, and character array referencing from studying this one example than from 20 pages of textual description.

```
names Ted
names+0 Ted
names+1 ed
names+2 d
names+3
names+5 va
names+8 Sam

(names+0)[0] T
(names+0)[1] e
(names+0)[2] d
(names+0)[3]
(names+0)[4] E
(names+0)[5] v

(names+2)[0] d
(names+2)[1]
(names+1)[4] v
```

## Review Questions

The answers to the review questions are in Appendix B.



1. True or false: You must access an array in the same order you initialized it.
2. Where did the bubble sort get its name?
3. Are the following values sorted in ascending or descending order?

   33 55 78 78 90 102 435 859 976 4092

4. How does C++ use the name of an array?

5. Given the following array definition:

```
char teams[] = {'E','a','g','l','e','s','\0',
                'R', 'a','m','s','\0'};
```

What is printed with each of these statements? (Answer "invalid" if the `cout` is illegal.)

a. `cout << teams;`

b. `cout << teams+7;`

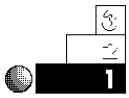
c. `cout << (teams+3);`

d. `cout << teams[0];`

e. `cout << (teams+0)[0];`

f. `cout << (teams+5);`

## Review Exercises



1. Write a program to store six of your friends' ages in a single array. Assign the ages in random order. Print the ages, from low to high, on-screen.

2. Modify the program in Exercise 1 to print the ages in descending order.

3. Using the new approach of subscripting arrays, rewrite the programs in Exercises 1 and 2. Always put a `0` in the subscript brackets, modifying the address instead (use `(ages+3)[0]` rather than `ages[3]`).

4. Sometimes *parallel arrays* are used in programs that must track more than one list of values that are related. For instance, suppose you had to maintain an inventory, tracking the integer part numbers, prices, and quantities of each item. This would require three arrays: an integer part number array, a floating-point price array, and an integer quantity array. Each array would have the same number of elements (the total number of parts in the inventory). Write a program to maintain such an inventory, and reserve enough elements

for 100 parts in the inventory. Present the user with an input screen. When the user enters a part number, search the part number array. When you locate the position of the part, print the corresponding price and quantity. If the part does not exist, enable the user to add it to the inventory, along with the matching price and quantity.

## Summary

You are beginning to see the true power of programming languages. Arrays give you the ability to search and sort lists of values. Sorting and searching are what computers do best; computers can quickly scan through hundreds and even thousands of values, looking for a match. Scanning through files of paper by hand, looking for just the right number, takes much more time. By stepping through arrays, your program can quickly scan, print, sort, and calculate a list of values. You now have the tools to sort lists of numbers, as well as search for values in a list.

You will use the concepts learned here for sorting and searching lists of character string data as well, when you learn a little more about the way C++ manipulates strings and pointers. To help build a solid foundation for this and more advanced material, you now know how to reference array elements without using conventional subscripts.

Now that you have mastered this chapter, the next one will be easy. Chapter 25, “Multidimensional Arrays,” shows you how you can keep track of arrays in a different format called a *matrix*. Not all lists of data lend themselves to matrices, but you should be prepared for when you need them.

# Multidimensional Arrays

Some data fits in lists, such as the data discussed in the previous two chapters, and other data is better suited for tables of information. This chapter takes arrays one step further. The previous chapters introduced single-dimensional arrays; arrays that have only one subscript and represent lists of values.

This chapter introduces arrays of more than one dimension, called *multidimensional arrays*. Multidimensional arrays, sometimes called *tables* or *matrices*, have at least two dimensions (rows and columns). Many times they have more than two.

This chapter introduces the following concepts:

- Multidimensional arrays
- Reserving storage for multidimensional arrays
- Putting data in multidimensional arrays
- Using nested `for` loops to process multidimensional arrays

If you understand single-dimensional arrays, you should have no trouble understanding arrays that have more than one dimension.

# Multidimensional Array Basics

A multidimensional array has more than one subscript.

A multidimensional array is an array with more than one subscript. Whereas a single-dimensional array is a list of values, a multidimensional array simulates a table of values, or multiple tables of values. The most commonly used table is a two-dimensional table (an array with two subscripts).

Suppose a softball team wanted to keep track of its players' batting records. The team played 10 games, and there are 15 players on the team. Table 25.1 shows the team's batting record.

Table 25.1. A softball team's batting record.

| *Player Name* | *Game 1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|---|
| Adams | 2 | 1 | 0 | 0 | 2 | 3 | 3 | 1 | 1 | 2 |
| Berryhill | 1 | 0 | 3 | 2 | 5 | 1 | 2 | 2 | 1 | 0 |
| Downing | 1 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 2 | 0 |
| Edwards | 0 | 3 | 6 | 4 | 6 | 4 | 5 | 3 | 6 | 3 |
| Franks | 2 | 2 | 3 | 2 | 1 | 0 | 2 | 3 | 1 | 0 |
| Grady | 1 | 3 | 2 | 0 | 1 | 5 | 2 | 1 | 2 | 1 |
| Howard | 3 | 1 | 1 | 1 | 2 | 0 | 1 | 0 | 4 | 3 |
| Jones | 2 | 2 | 1 | 2 | 4 | 1 | 0 | 7 | 1 | 0 |
| Martin | 5 | 4 | 5 | 1 | 1 | 0 | 2 | 4 | 1 | 5 |
| Powers | 2 | 2 | 3 | 1 | 0 | 2 | 1 | 3 | 1 | 2 |
| Smith | 1 | 1 | 2 | 1 | 3 | 4 | 1 | 0 | 3 | 2 |
| Smithtown | 1 | 0 | 1 | 2 | 1 | 0 | 3 | 4 | 1 | 2 |
| Townsend | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Ulmer | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 3 | 1 | 3 |
| Williams | 2 | 3 | 1 | 0 | 1 | 2 | 1 | 2 | 0 | 3 |

Do you see that the softball table is a two-dimensional table? It has rows (the first dimension) and columns (the second dimension). Therefore, this is called a two-dimensional table with 15 rows and 10 columns. (Generally, the number of rows is specified first.)

Each row has a player's name, and each column has a game number associated with it, but these are not part of the actual data. The data consists of only 150 values (15 rows by 10 columns). The data in a two-dimensional table always is the same type of data; in this case, every value is an integer. If it were a table of salaries, every element would be a floating-point decimal.

The number of dimensions, in this case two, corresponds to the dimensions in the physical world. The single-dimensioned array is a line, or list of values. Two dimensions represent both length and width. You write on a piece of paper in two dimensions; two dimensions represent a flat surface. Three dimensions represent width, length, and depth. You have seen 3-D movies. Not only do the images have width and height, but they also seem to have depth. Figure 25.1 shows what a three-dimensional array looks like if it has a depth of four, six rows, and three columns. Notice that a three-dimensional table resembles a cube.

A three-dimensional table has three dimensions: depth, rows, and columns.

It is difficult to visualize more than three dimensions. However, you can think of each dimension after three as another occurrence. In other words, a list of one player's season batting record can be stored in an array. The team's batting record (as shown in Table 25.1) is two-dimensional. The league, made of up several teams' batting records, represents a three-dimensional table. Each team (the depth of the table) has rows and columns of batting data. If there is more than one league, it is another dimension (another set of data).

C++ enables you to store several dimensions, although "real-world" data rarely requires more than two or three.

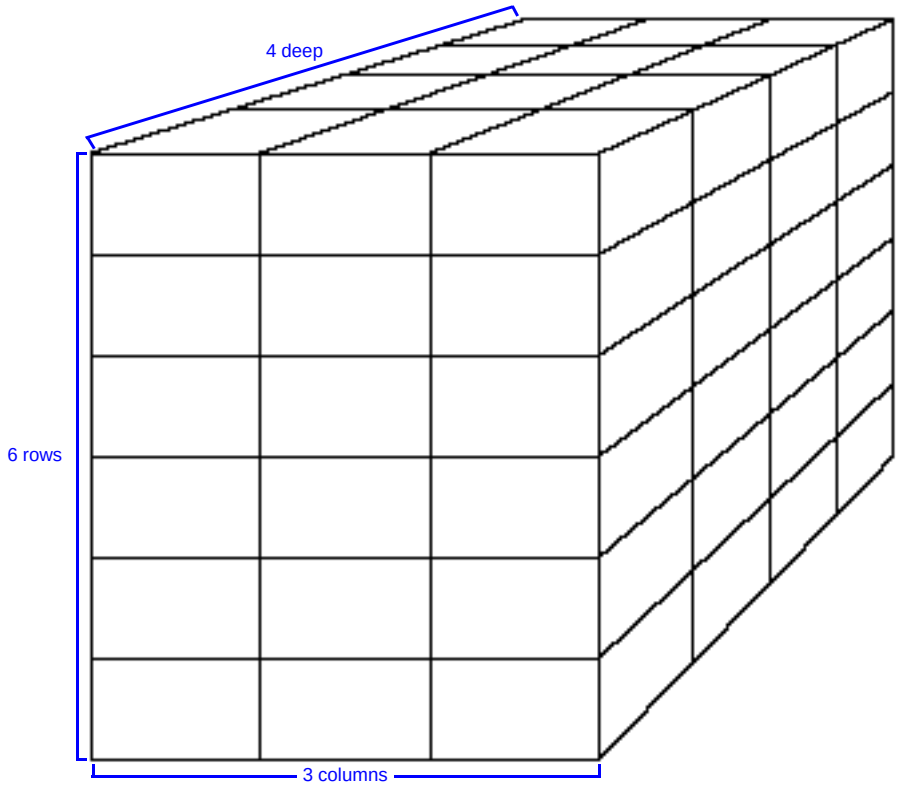



Figure 25.1. Representing a three-dimensional table (a cube).

## Reserving Multidimensional Arrays

When you reserve a multidimensional array, you must inform C++ that the array has more than one dimension by putting more than one subscript in brackets after the array name. You must put a separate number, in brackets, for each dimension in the table. For example, to reserve the team data from Table 25.1, you use the following multidimensional array declaration.

*Declare an integer array called* `teams` *with 15 rows and 10 columns.*

```
int teams[15][10];  // Reserves a two-dimensional table.
```

**CAUTION:** Unlike other programming languages, C++ requires you to enclose each dimension in brackets. Do not reserve multidimensional array storage like this:

```
int teams[15,10];   // Invalid table declaration.
```

Properly reserving the `teams` table produces a table with 150 elements. Figure 25.2 shows what each element's subscript looks like.

columns

rows

| [0][0] | [0][1] | [0][2] | [0][3] | [0][4] | [0][5] | [0][6] | [0][7] | [0][8] | [0][9] |
|---|---|---|---|---|---|---|---|---|---|
| [1][0] | [1][1] | [1][2] | [1][3] | [1][4] | [1][5] | [1][6] | [1][7] | [1][8] | [1][9] |
| [2][0] | [2][1] | [2][2] | [2][3] | [2][4] | [2][5] | [2][6] | [2][7] | [2][8] | [2][9] |
| [3][0] | [3][1] | [3][2] | [3][3] | [3][4] | [3][5] | [3][6] | [3][7] | [3][8] | [3][9] |
| [4][0] | [4][1] | [4][2] | [4][3] | [4][4] | [4][5] | [4][6] | [4][7] | [4][8] | [4][9] |
| [5][0] | [5][1] | [5][2] | [5][3] | [5][4] | [5][5] | [5][6] | [5][7] | [5][8] | [5][9] |
| [6][0] | [6][1] | [6][2] | [6][3] | [6][4] | [6][5] | [6][6] | [6][7] | [6][8] | [6][9] |
| [7][0] | [7][1] | [7][2] | [7][3] | [7][4] | [7][5] | [7][6] | [7][7] | [7][8] | [7][9] |
| [8][0] | [8][1] | [8][2] | [8][3] | [8][4] | [8][5] | [8][6] | [8][7] | [8][8] | [8][9] |
| [9][0] | [9][1] | [9][2] | [9][3] | [9][4] | [9][5] | [9][6] | [9][7] | [9][8] | [9][9] |
| [10][0] | [10][1] | [10][2] | [10][3] | [10][4] | [10][5] | [10][6] | [10][7] | [10][8] | [10][9] |
| [11][0] | [11][1] | [11][2] | [11][3] | [11][4] | [11][5] | [11][6] | [11][7] | [11][8] | [11][9] |
| [12][0] | [12][1] | [12][2] | [12][3] | [12][4] | [12][5] | [12][6] | [12][7] | [12][8] | [12][9] |
| [13][0] | [13][1] | [13][2] | [13][3] | [13][4] | [13][5] | [13][6] | [13][7] | [13][8] | [13][9] |
| [14][0] | [14][1] | [14][2] | [14][3] | [14][4] | [14][5] | [14][6] | [14][7] | [14][8] | [14][9] |

Figure 25.2. Subscripts for the softball team table.

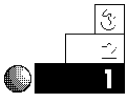
If you had to track three teams, each with 15 players and 10 games, the three-dimensional table would be created as follows:

```
int teams[3][15][10]; // Reserves a three-dimensional table.
```

The far-right dimension always represents columns, the next represents rows, and so on.

When creating a two-dimensional table, always put the maximum number of rows first, and the maximum number of columns second. C++ always uses `0` as the starting subscript of each dimension. The last element, the lower-right element of the `teams` table, is `teams[2][14][9]`.

### Examples



1. Suppose you wanted to keep track of utility bills for the year. You can store 12 months of four utilities in a two-dimensional table of floating-point amounts, as the following array declaration demonstrates:

   ```
   float utilities[12][4];          // Reserves 48 elements.
   ```

   You can compute the total number of elements in a multidimensional array by multiplying the subscripts. Because 12 times 4 is 48, there are 48 elements in this array (12 rows, 4 columns). Each of these elements is a floating-point data type.

2. If you were keeping track of five years' worth of utilities, you have to add an extra dimension. The first dimension is the years, the second is the months, and the last is the individual utilities. Here is how you reserve storage:

   ```
   float utilities[5][12][4];      // Reserves 240 elements.
   ```

## Mapping Arrays to Memory

C++ approaches multidimensional arrays a little differently than most programming languages do. When you use subscripts, you do not have to understand the internal representation of multidimensional arrays. However, most C++ programmers think a deeper understanding of these arrays is important, especially when programming advanced applications.

A two-dimensional array is actually an *array of arrays*. You program multidimensional arrays as though they were tables with rows and columns. A two-dimensional array is actually a single-dimensional array, but each of its elements is not an integer, floating-point, or character, but another array.

Knowing that a multidimensional array is an array of other arrays is critical when passing and receiving such arrays. C++ passes all arrays, including multidimensional arrays, by address. Suppose you were using an integer array called `scores`, reserved as a 5-by-6 table. You can pass `scores` to a function called `print_it()`, as follows:

```
print_it(scores);          // Passes table to a function.
```

The function `print_it()` has to identify the type of parameter being passed to it. The `print_it()` function also must recognize that the parameter is an array. If `scores` were one-dimensional, you could receive it as

```
print_it(int scores[])          // Works only if scores
                                // is one-dimensional.
```

or

```
print_it(int scores[10])        // Assuming scores
                                // has 10 elements.
```

If `scores` were a multidimensional table, you would have to designate each pair of brackets and put the maximum number of subscripts in its brackets, as in

```
print_it(int scores[5][6])       // Inform print_it() of
                                 // the array's dimensions.
```

or

```
print_it(int scores[][6])        // Inform print_it() of
                                 // the array's dimensions.
```

Notice you do not have to explicitly state the maximum subscript on the first dimension when receiving multidimensional

arrays, but you must designate the second. If `scores` were a three-dimensional table, dimensioned as 10 by 5 by 6, you would receive it with `print_it()` as

```
print_it(int scores[][5][6])      // Only first dimension
                                  // is optional.
```

or

```
print_it(int scores[10][5][6])     // Inform print_it() of
                                   // array's dimensions.
```

You should not have to worry too much about the way tables are physically stored. Even though a two-dimensional table is actually an array of arrays (and each of those arrays contains another array if it is a three-dimensional table), you can use subscripts to program multidimensional arrays as if they were stored in row-and-column order.

C++ stores multidimensional arrays in row order.

Multidimensional arrays are stored in *row order*. Suppose you want to keep track of a 3-by-4 table. The top of Figure 25.3 shows how that table (and its subscripts) are visualized. Despite the two-dimensional table organization, your memory is still sequential storage. C++ has to map multidimensional arrays to single-dimensional memory, and it does so in row order.

Each row fills memory before the next row is stored. Figure 25.3 shows how a 3-by-4 table is mapped to memory.

The entire first row (`table[0][0]` through `table[0][3]`) is stored first in memory before any of the second row. A table is actually an array of arrays, and, as you learned in previous chapters, array elements are always stored sequentially in memory. Therefore, the first row (array) completely fills memory before the second row. Figure 25.3 shows how two-dimensional arrays map to memory.

## Defining Multidimensional Arrays

C++ is not picky about the way you define a multidimensional array when you initialize it at declaration time. As with single-dimensional arrays, you initialize multidimensional arrays with

braces that designate dimensions. Because a multidimensional array is an array of arrays, you can nest braces when you initialize them.

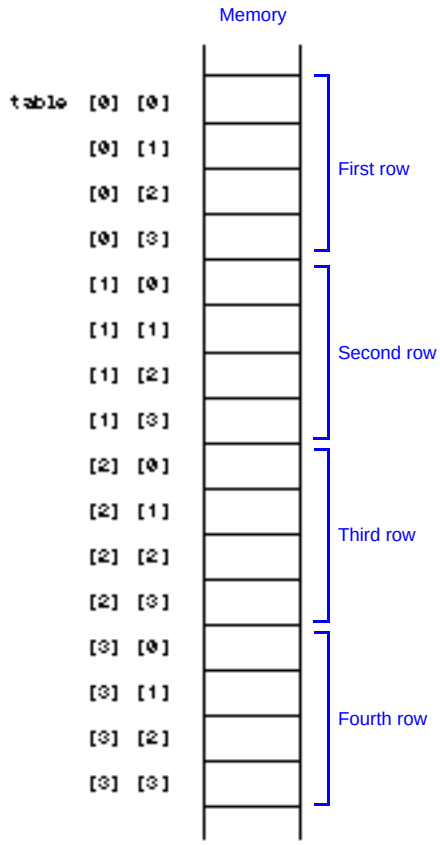



Figure 25.3. Mapping a two-dimensional table to memory.

The following three array definitions fill the three arrays `ara1`, `ara2`, and `ara3`, as shown in Figure 25.4:

```
int ara1[5] = {8, 5, 3, 25, 41};  // One-dimensional array.
int ara2[2][4]={{4, 3, 2, 1},{1, 2, 3, 4}};
int ara3[3][4]={{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11, 12}};
```

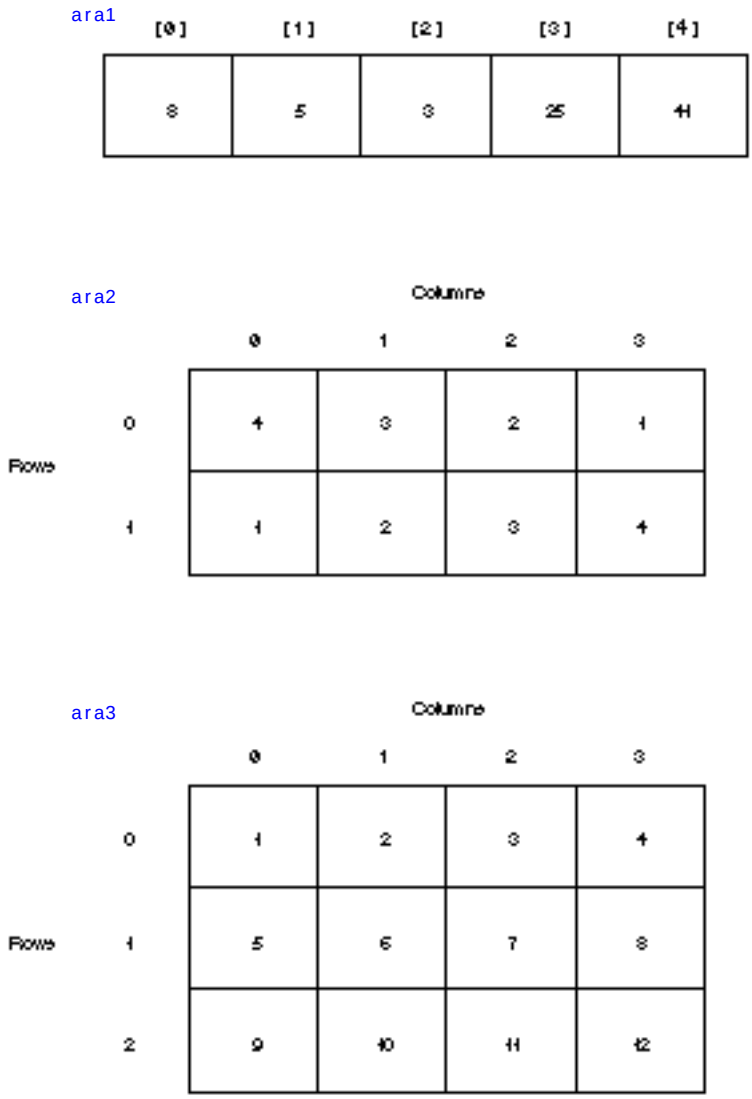



Figure 25.4. After initializing a table.

Notice that the multidimensional arrays are stored in row order. In `ara3`, the first row receives the first four elements of the definition (1, 2, 3, and 4).

**TIP:** To make a multidimensional array initialization match the array's subscripts, some programmers like to show how arrays are filled. Because C++ programs are free-form, you can initialize ara2 and ara3 as

```
int ara2[2][4]={{4, 3, 2, 1},  // Does exactly the same
                {1, 2, 3, 4}}; // thing as before.

int ara3[3][4]={{1, 2, 3, 4},
                {5, 6, 7, 8},
                {9, 10, 11, 12}; // Visually more
                                 // obvious.
```

You can initialize a multidimensional array as if it were single-dimensional in C++. You must keep track of the row order if you do this. For instance, the following two definitions also reserve storage for and initialize ara2 and ara3:

```
int ara2[2][4]={4, 3, 2, 1, 1, 2, 3, 4};
int ara3[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

There is no difference between initializing ara2 and ara3 with or without the nested braces. The nested braces seem to show the dimensions and how C++ fills them a little better, but the choice of using nested braces is yours.

**TIP:** Multidimensional arrays (unless they are global) are not initialized to specific values unless you assign them values at declaration time or in the program. As with single-dimensional arrays, if you initialize one or more of the elements, but not all of them, C++ fills the rest with zeros. If you want to fill an entire multidimensional array with zeros, you can do so with the following:

```
float sales[3][4][7][2] = {0};  // Fills all sales
                                // with zeros.
```

One last point to consider is how multidimensional arrays are viewed by your compiler. Many people program in C++ for years, but never understand how tables are stored internally. As long as you use subscripts, a table's internal representation should not matter. When you learn about pointer variables, however, you might want to know how C++ stores your tables in case you want to reference them with pointers (as shown in the next few chapters).

Figure 25.5 shows the way C++ stores a 3-by-4 table in memory. Unlike single-dimensional arrays, each element is stored contiguously, but notice how C++ views the data. Because a table is an array of arrays, the array name contains the address of the start of the primary array. Each of those elements points to the arrays it contains (the data in each row). This coverage of table storage is for your information only, at this point. As you become more proficient in C++, and write more powerful programs that manipulate internal memory, you might want to review this table storage method.

## Tables and **for** Loops

As the following examples show, nested `for` loops are useful when you want to loop through every element of a multidimensional table.

For instance, the section of code,

```
for (row=0; row<2; row++)
   { for (col=0; col<3; col++)
        { cout << row << " " << col "\n"; }
   }
```

produces the following output:

```
0    0
0    1
0    2
1    0
1    1
1    2
```

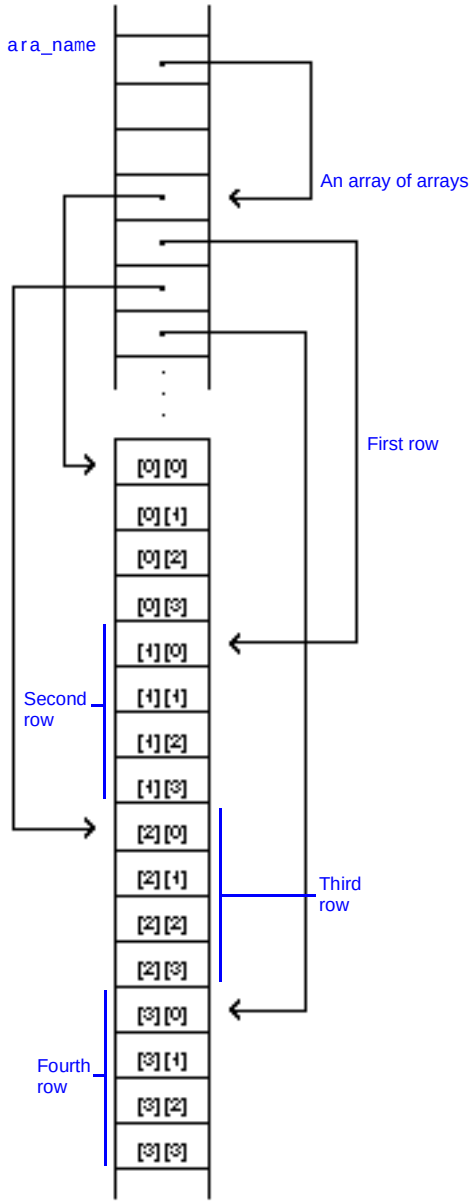



Figure 25.5. Internal representation of a two-dimensional table.

These numbers are the subscripts, in row order, for a two-row by three-column table dimensioned with

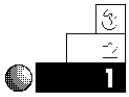
```
int table[2][3];
```

Nested loops work well with multi-dimensional arrays.

Notice there are as many `for` loops as there are subscripts in the array (two). The outside loop represents the first subscript (the rows), and the inside loop represents the second subscript (the columns). The nested `for` loop steps through each element of the table.

You can use `cin`, `gets()`, `get`, and other input functions to fill a table, and you also can assign values to the elements when declaring the table. More often, the data comes from data files on the disk. Regardless of what method stores the values in multidimensional arrays, nested `for` loops are excellent control statements to step through the subscripts. The following examples demonstrate how nested `for` loops work with multidimensional arrays.

## Examples



1. The following statements reserve enough memory elements for a television station's ratings (*A* through *D*) for one week:

   ```
   char ratings[7][48];
   ```

   These statements reserve enough elements to hold seven days (the rows) of ratings for each 30-minute time slot (48 of them in a day).

   Every element in a table is always the same type. In this case, each element is a character variable. Some are initialized with the following assignment statements:

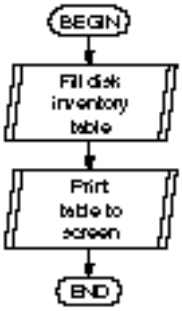
   ```
   shows[3][12] = 'B';      // Stores B in 4th row, 13th column.
   shows[1][5] = 'A' ;      // Stores C in 2nd row, 6th column.
   shows[6][20] = getch();  // Stores the letter the user types.
   ```

2. A computer company sells two sizes of disks: 3 1/2-inch and 5 1/4-inch. Each disk comes in one of four capacities: single-sided double-density, double-sided double-density, single-sided high-density, and double-sided high-density.

The disk inventory is well-suited for a two-dimensional table. The company determined that the disks have the following retail prices:

| | ***Double Density*** | | ***High Density*** | |
|---|---|---|---|---|
| | *Single* | *Double* | *Single* | *Double* |
| 3 1/2-*inch* | 2.30 | 2.75 | 3.20 | 3.50 |
| 5 1/4-*inch* | 1.75 | 2.10 | 2.60 | 2.95 |

The company wants to store the price of each disk in a table for easy access. The following program stores the prices with assignment statements.




```
// Filename: C25DISK1.CPP
// Assigns disk prices to a table.
#include <iostream.h>
#include <iomanip.h>
void main()
{
   float disks[2][4];  // Table of disk prices.
   int row, col;       // Subscript variables.

   disks[0][0] = 2.39;      // Row 1, column 1
   disks[0][1] = 2.75;      // Row 1, column 2
   disks[0][2] = 3.29;      // Row 1, column 3
   disks[0][3] = 3.59;      // Row 1, column 4
   disks[1][0] = 1.75;      // Row 2, column 1
   disks[1][1] = 2.19;      // Row 2, column 2
   disks[1][2] = 2.69;      // Row 2, column 3
   disks[1][3] = 2.95;      // Row 2, column 4

   // Print the prices.
   for (row=0; row<2; row++)
      { for (col=0; col<4; col++)
        { cout << "$" << setprecision(2) <<
                  disks[row][col] << "\n"; }
      }

   return;
}
```

This program displays the prices as follows:

```
$2.39
$2.75
$3.29
$3.59
$1.75
$2.19
$2.69
$2.95
```

It prints them one line at a time, without any descriptive titles. Although the output is not labeled, it illustrates how you can use assignment statements to initialize a table, and how nested `for` loops can print the elements.

3. The preceding disk inventory would be displayed better if the output had descriptive titles. Before you add titles, it is helpful for you to see how to print a table in its native row and column format.

   Typically, you use a nested `for` loop, such as the one in the previous example, to print rows and columns. You should not output a newline character with every `cout`, however. If you do, you see one value per line, as in the previous program's output, which is not the row and column format of the table.

   You do not want to see every disk price on one line, but you want each row of the table printed on a separate line. You must insert a `cout << "\n";` to send the cursor to the next line each time the row number changes. Printing newlines after each row prints the table in its row and column format, as this program shows:

```
// Filename: C25DISK2.CPP
// Assigns disk prices to a table
// and prints them in a table format.
#include <iostream.h>
#include <iomanip.h>
void main()
{
```

```
   float disks[2][4];  // Table of disk prices.
   int row, col;

   disks[0][0] = 2.39;      // Row 1, column 1
   disks[0][1] = 2.75;      // Row 1, column 2
   disks[0][2] = 3.29;      // Row 1, column 3
   disks[0][3] = 3.59;      // Row 1, column 4
   disks[1][0] = 1.75;      // Row 2, column 1
   disks[1][1] = 2.19;      // Row 2, column 2
   disks[1][2] = 2.69;      // Row 2, column 3
   disks[1][3] = 2.95;      // Row 2, column 4

   // Print the prices
   for (row=0; row<2; row++)
      { for (col=0; col<4; col++)
        { cout << "$" << setprecision(2) <<
                  disks[row][col] << "\t";
        }

        cout << "\n";  // Prints a new line after each row.
      }

   return;
}
```

Here is the output of the disk prices in their native table order:

```
$2.39    $2.75    $3.29    $3.59
$1.75    $2.19    $2.69    $2.95
```

4. To add the titles, simply print a row of titles before the first row of values, then print a new column title before each column, as shown in the following program:

```
// Filename: C25DISK3.CPP
// Assigns disk prices to a table
// and prints them in a table format with titles.
#include <iostream.h>
#include <iomanip.h>
```

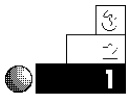
```
void main()
{
   float disks[2][4];  // Table of disk prices.
   int row, col;

   disks[0][0] = 2.39;      // Row 1, column 1
   disks[0][1] = 2.75;      // Row 1, column 2
   disks[0][2] = 3.29;      // Row 1, column 3
   disks[0][3] = 3.59;      // Row 1, column 4
   disks[1][0] = 1.75;      // Row 2, column 1
   disks[1][1] = 2.19;      // Row 2, column 2
   disks[1][2] = 2.69;      // Row 2, column 3
   disks[1][3] = 2.95;      // Row 2, column 4

   // Print the column titles.
   cout << "\tSingle-sided\tDouble-sided\tSingle-sided\t" <<
           "Double-sided\n";
   cout << "\tDouble-density\tDouble-density\tHigh-density" <<
           "\tHigh-density\n";

   // Print the prices
   for (row=0; row<2; row++)
      { if (row == 0)
       { cout << "3-1/2\"\t"; }         // Need \" to
                                       // print quotation.
     else
       { cout << "5-1/4\"\t"; }
     for (col=0; col<4; col++)    // Print the current row.
     { cout << setprecision(2) << "$" << disks[row][col]
            << "\t\t";
        }
        cout << "\n";    // Print a newline after each row.
      }

   return;
}
```

Here is the output from this program:

```
          Single-sided    Double-sided    Single-sided    Double-sided
          Double-density  Double-density  High-density    High-density
3-1/2"    $2.39           $2.75           $3.29           $3.59
5-1/4"    $1.75           $2.19           $2.69           $2.95
```

## Review Questions

The answers to the review questions are in Appendix B.



1. What statement reserves a two-dimensional table of integers called `scores` with five rows and six columns?
2. What statement reserves a three-dimensional table of four character arrays called `initials` with 10 rows and 20 columns?
3. In the following statement, which subscript (first or second) represents rows and which represents columns?

```
int weights[5][10];
```

4. How many elements are reserved with the following statement?

```
int ara[5][6];
```

5. The following table of integers is called `ara`:

| | | | | |
|---|---|---|---|---|
| 4 | 1 | 3 | 5 | 9 |
| 10 | 2 | 12 | 1 | 6 |
| 25 | 42 | 2 | 91 | 8 |

What values do the following elements contain?

a. `ara[2][2]`

b. `ara[0][1]`

c. `ara[2][3]`

d. `ara[2][4]`

6. What control statement is best for stepping through multidimensional arrays?
7. Notice the following section of a program:

```
int grades[3][5] = {80,90,96,73,65,67,90,68,92,84,70,
                    55,95,78,100};
```

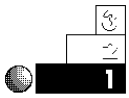
What are the values of the following:

a. `grades[2][3]`

b. `grades[2][4]`

c. `grades[0][1]`

## Review Exercises



1. Write a program that stores and prints the numbers from 1 to 21 in a 3-by-7 table. (*Hint:* Remember C++ begins subscripts at 0.)
2. Write a program that reserves storage for three years' worth of sales data for five salespeople. Use assignment statements to fill the table with data, then print it, one value per line.

3. Instead of using assignment statements, use the `cin` function to fill the salespeople data from Exercise 2.
4. Write a program that tracks the grades for five classes, each having 10 students. Input the data using the `cin` function. Print the table in its native row and column format.

## Summary

You now know how to create, initialize, and process multidimensional arrays. Although not all data fits in the compact format of tables, much does. Using nested `for` loops makes stepping through a multidimensional array straightforward.

## EXAMPLE

One of the limitations of a multidimensional array is that each element must be the same data type. This keeps you from being able to store several kinds of data in tables. Chapter 28, “Structures,” shows you how to store data in different ways to overcome this limitation.

# Pointers

C++ reveals its true power through pointer variables. Pointer variables (or *pointers*, as they generally are called) are variables that contain addresses of other variables. All variables you have seen so far have held data values. You understand that variables hold various data types: character, integer, floating-point, and so on. Pointer variables contain the location of regular data variables; they in effect point to the data because they hold the address of the data.

When first learning C++, students of the language tend to shy away from pointers, thinking that pointers will be difficult. Pointers do not have to be difficult. In fact, after you work with them for a while, you will find they are easier to use than arrays (and much more flexible).

This chapter introduces the following concepts:

- Pointers
- Pointers of different data types
- The "address of" (&) operator
- The dereferencing (*) operator
- Arrays of pointers

Pointers offer a highly efficient means of accessing and changing data. Because pointers contain the actual address of your data, your compiler has less work to do when finding that data in memory. Pointers do not have to link data to specific variable names. A pointer can point to an unnamed data value. With pointers, you gain a "different view" of your data.

## Introduction to Pointer Variables

Pointers contain addresses of other variables.

Pointers are variables. They follow all the normal naming rules of regular, nonpointer variables. As with regular variables, you must declare pointer variables before using them. There is a type of pointer for every data type in C++; there are integer pointers, character pointers, floating-point pointers, and so on. You can declare global pointers or local pointers, depending on where you declare them.

About the only difference between pointer variables and regular variables is the data they hold. Pointers do not contain data in the usual sense of the word. Pointers contain addresses of data. If you need a quick review of addresses and memory, see Appendix A, "Memory Addressing, Binary, and Hexadecimal Review."

There are two pointer operators in C++:

- `&` The "address of" operator
- `*` The dereferencing operator

Don't let these operators throw you; you might have seen them before! The `&` is the bitwise AND operator (from Chapter 11, "Additional C++ Operators") and the `*` means, of course, multiplication. These are called *overloaded* operators. They perform more than one function, depending on how you use them in your programs. C++ does not confuse `*` for multiplication when you use it as a dereferencing operator with pointers.

Any time you see the & used with pointers, think of the words "address of." The & operator always produces the memory address of whatever it precedes. The * operator, when used with pointers, either declares a pointer or dereferences the pointer's value. The next section explains each of these operators.

## Declaring Pointers

Because you must declare all pointers before using them, the best way to begin learning about pointers is to understand how to declare and define them. Actually, declaring pointers is almost as easy as declaring regular variables. After all, pointers are variables.

If you must declare a variable that holds your age, you could do so with the following variable declaration:

```
int age=30;        // Declare a variable to hold my age.
```

Declaring age like this does several things. It enables C++ to identify a variable called age, and to reserve storage for that variable. Using this format also enables C++ to recognize that you will store only integers in age, not floating-point or double floating-point data. The declaration also requests that C++ store the value of 30 in age after it reserves storage for age.

Where did C++ store age in memory? As the programmer, you should not really care where C++ stores age. You do not have to know the variable's address because you will never refer to age by its address. If you want to calculate with or print age, you call it by its name, age.

**TIP:** Make your pointer variable names meaningful. The name file_ptr makes more sense than x13 for a file-pointing variable, although either name is allowed.

Suppose you want to declare a pointer variable. This pointer variable will not hold your age, but it will point to age, the variable that holds your age. (Why you would want to do this is explained in this and the next few chapters.) p_age might be a good name for the pointer variable. Figure 26.1 illustrates what you want to do. The

figure assumes C++ stored age at the address 350,606. Your C++ compiler, however, arbitrarily determines the address of age, so it could be anything.


Figure 26.1. p_age contains the address of age; p_age points to the age variable.

The name p_age has nothing to do with pointers, except that it is the name you made up for the pointer to age. Just as you can name variables anything (as long as the name follows the legal naming rules of variables), p_age could just as easily have been named house, x43344, space_trek, or whatever else you wanted to call it. This reinforces the idea that a pointer is just a variable you reserve in your program. Create meaningful variable names, even for pointer variables. p_age is a good name for a variable that points to age (as would be ptr_age and ptr_to_age).

To declare the p_age pointer variable, you must program the following:

```
int * p_age;              // Declares an integer pointer.
```

Similar to the declaration for age, this declaration reserves a variable called p_age. The p_age variable is not a normal integer variable, however. Because of the dereferencing operator, *, C++ knows this is to be a pointer variable. Some C++ programmers prefer to declare such a variable without a space after the *, as follows:

```
int *p_age;               // Declares an integer pointer.
```

Either method is okay, but you must remember the * is *not* part of the name. When you later use p_age, you will not prefix the name with the *, unless you are dereferencing it at the time (as later examples show).

**TIP:** Whenever the dereferencing operator, *, appears in a variable definition, the variable being declared is *always* a pointer variable.

Consider the declaration for p_age if the asterisk were not there: C++ would think you were declaring a regular integer variable. The * is important, because it tells C++ to interpret p_age as a pointer variable, not as a normal, data variable.

## Assigning Values to Pointers

Pointers can point only to data of their own type.

p_age is an integer pointer. This is very important. p_age can point only to integer values, never to floating-point, double floating-point, or even character variables. If you needed to point to a floating-point variable, you might do so with a pointer declared as

```
float *point;      // Declares a floating-point pointer.
```

As with any automatic variable, C++ does not initialize pointers when you declare them. If you declared p_age as previously described, and you wanted p_age to point to age, you would have to explicitly assign p_age to the address of age. The following statement does this:

```
p_age = &age;     // Assign the address of age to p_age.
```

What value is now in p_age? You do not know exactly, but you know it is the address of age, wherever that is. Rather than assign the address of age to p_age with an assignment operator, you can declare and initialize pointers at the same time. These lines declare and initialize both age and p_age:

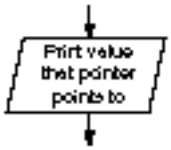
```
int age=30;       // Declares a regular integer
                  // variable, putting 30 in it.
```

```
int *p_age=&age;    // Declares an integer pointer,
                    // initializing it with the address
                    // of p_age.
```

These two lines produce the variables described in Figure 26.1.

If you wanted to print the value of age, you could do so with the following cout:

```
cout << age;            // Prints the value of age.
```




You also can print the value of age like this:

```
cout << *p_age;         // Dereferences p_age.
```

The dereference operator produces a value that tells the pointer where to point. Without the *, the last cout would print an address (the address of age). With the *, the cout prints the value at that address.

You can assign a different value to age with the following statement:

```
age=41;                 // Assigns a new value to age.
```

You also can assign a value to age like this:

```
*p_age=41;
```

This declaration assigns 41 to the value to which p_age points.

**TIP:** The * appears before a pointer variable in only two places—when you declare a pointer variable, and when you dereference a pointer variable (to find the data it points to).

## Pointers and Parameters

Now that you understand the pointer's * and & operators, you can finally see why scanf()'s requirements were not as strict as they first seemed. While passing a regular variable to scanf(), you had to prefix the variable with the & operator. For instance, the following scanf() gets three integer values from the user:

```
scanf(" %d %d %d", &num1, &num2, &num3);
```

This `scanf()` does not pass the three variables, but passes the addresses of the three variables. Because `scanf()` knows the exact locations of these parameters in memory (because their addresses were passed), it goes to those addresses and puts the keyboard input values into those addresses.

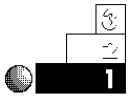
This is the only way `scanf()` could work. If you passed these variables by copy, without putting the "address of" operator (`&`) before them, `scanf()` would get the keyboard input and fill a *copy* of the variables, but not the actual variables `num1`, `num2`, and `num3`. When `scanf()` then returned control to your program, you would not have the input values. Of course, the `cin` operator does not have the ampersand (`&`) requirement and is easier to use for most C++ programs.

You might recall from Chapter 18, "Passing Values," that you can override C++'s normal default of passing by copy (or "by value"). To pass by address, receive the variable preceded by an `&` in the receiving function. The following function receives `tries` by address:

```
pr_it(int &tries);    // Receive integer tries in pr_it() by
                      // address (pr_it would normally receive
                      // tries by copy).
```

Now that you understand the `&` and `*` operators, you can understand completely the passing of nonarray parameters by address to functions. (Arrays default to passing by address without requiring that you use `&`.)

## Examples



1. The following section of code declares three regular variables of three different data types, and three corresponding pointer variables:

```
char initial= 'Q';     // Declares three regular variables
int num=40;            // of three different types.
float sales=2321.59;
```

```
char *p_initial=&initial;   // Declares three pointers.
int * ptr_num=&num;         // Pointer names and spacing
float * sales_add = &sales; // after * are not critical.
```

2. Just like regular variables, you can initialize pointers with assignment statements. You do not have to initialize them when you declare them. The next few lines of code are equivalent to the code in Example 1:

```
char initial;       // Declares three regular variables
int num;            // of three different types.
float sales;

char *p_initial;    // Declares three pointers but does
int * ptr_num;      // not initialize them yet.
float * sales_add;

initial='Q';        // Initializes the regular variables
num=40;             // with values.
sales=2321.59;

p_initial=&initial;     // Initializes the pointers with
ptr_num=&num;           // the addresses of their
sales_add=&sales;       // corresponding variables.
```

Notice that you do not put the * operator before the pointer variable names when assigning them values. You would prefix a pointer variable with the * only if you were dereferencing it.

**NOTE:** In this example, the pointer variables could have been assigned the addresses of the regular variables before the regular variables were assigned values. There would be no difference in the operation. The pointers are assigned the addresses of the regular variables no matter what the data in the regular variables are.

Keep the data type of each pointer consistent with its corresponding variable. Do not assign a floating-point variable to an integer's address. For instance, you cannot make the following assignment statement:

```
p_initial = &sales;        // Invalid pointer assignment.
```

because `p_initial` can point only to character data, not to floating-point data.

3. The following program is an example you should study closely. It shows more about pointers and the pointer operators, `&` and `*`, than several pages of text can do.

```
// Filename: C26POINT.CPP
// Demonstrates the use of pointer declarations
// and operators.
#include <iostream.h>

void main()
{
   int num=123;            // A regular integer variable.
   int *p_num;             // Declares an integer pointer.

   cout << "num is " << num << "\n";  // Prints value of num.
   cout << "The address of num is " << &num << "\n";
                           // Prints num's location.
   p_num = &num;           // Puts address of num in p_num,
                           // in effect making p_num point
                           // to num.
                           // No * in front of p_num.
   cout << "*p_num is " << *p_num << "\n"; // Prints value
                                          // of num.
   cout << "p_num is " << p_num << "\n";   // Prints location
                                          // of num.

   return;
}
```

Here is the output from this program:

```
num is 123
The address of num is 0x8fbd0ffe
*p_num is 123
p_num is 0x8fbd0ffe
```

If you run this program, you probably will get different results for the value of `p_num` because your compiler will place `num` at a different location, depending on your memory setup. The value of `p_num` prints in hexadecimal because it is an address of memory. The actual address does not matter, however. Because the pointer `p_num` always contains the address of `num`, and because you can dereference `p_num` to get `num`'s value, the actual address is not critical.

4. The following program includes a function that swaps the values of any two integers passed to it. You might recall that a function can return only a single value. Therefore, before now, you could not write a function that changed two different values and returned both values to the calling function.

   To swap two variables (reversing their values for sorting, as you saw in Chapter 24, "Array Processing"), you need the ability to pass both variables by address. Then, when the function reverses the variables, the calling function's variables also are swapped.

   Notice the function's use of dereferencing operators before each occurrence of `num1` and `num2`. It does not matter at which address `num1` and `num2` are stored, but you must make sure that you dereference whatever addresses were passed to the function.

   Be sure to receive arguments with the prefix `&` in functions that receive by address, as done here.

*Identify the program and include the I/O header file. This program swaps two integers, so initialize two integer variables in* `main()`. *Pass the variables to the swapping function, called* `swap_them`, *then switch their values. Print the results of the swap in* `main()`.


```
// Filename: C26SWAP.CPP
// Program that includes a function that swaps
// any two integers passed to it
#include <iostream.h>
void swap_them(int &num1, int &num2);

void main()
{
     int i=10, j=20;
     cout << "\n\nBefore swap, i is " << i <<
             " and j is " << j << "\n\n";
     swap_them(i, j);
     cout << "\n\nAfter swap, i is " << i <<
              " and j is " << j << "\n\n";
     return;
}
void swap_them(int &num1, int &num2)
{
   int temp;           // Variable that holds
                       // in-between swapped value.
   temp = num1;        // The calling function's variables
   num1 = num2;        // (and not copies of them) are
   num2 = temp;        // changed in this function.
   return;
}
```

## Arrays of Pointers

If you have to reserve many pointers for many different values, you might want to declare an array of pointers. You know that you can reserve an array of characters, integers, long integers, and floating-point values, as well as an array of every other data type available. You also can reserve an array of pointers, with each pointer being a pointer to a specific data type.

The following reserves an array of 10 integer pointer variables:

```
int *iptr[10]; // Reserves an array of 10 integer pointers
```

Figure 26.2 shows how C++ views this array. Each element holds an address (after being assigned values) that points to other values in memory. Each value pointed to must be an integer. You can assign an element from `iptr` an address just as you would for nonarray pointer variables. You can make `iptr[4]` point to the address of an integer variable named `age` by assigning it like this:

```
iptr[4] = &age;     // Make iptr[4] point to address of age.
```

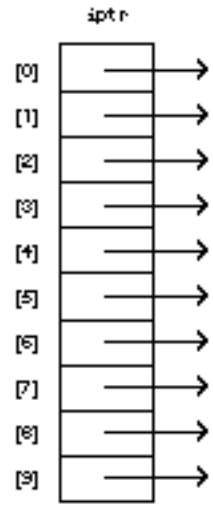



Figure 26.2. An array of 10 integer pointers.

The following reserves an array of 20 character pointer variables:

```
char *cpoint[20];          // Array of 20 character pointers.
```

Again, the asterisk is not part of the array name. The asterisk lets C++ know that this is an array of integer pointers and not just an array of integers.

Some beginning C++ students get confused when they see such a declaration. Pointers are one thing, but reserving storage for arrays of pointers tends to bog novices down. However, reserving storage for arrays of pointers is easy to understand. Remove the asterisk from the previous declaration as follows,

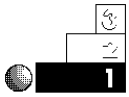
```
char cpoint[20];
```

and what do you have? You have just reserved a simple array of 20 characters. Adding the asterisk tells C++ to go one step further: rather than an array of character variables, you want an array of character pointing variables. Rather than having each element be a character variable, you have each element hold an address that points to characters.

Reserving arrays of pointers will be much more meaningful after you learn about structures in the next few chapters. As with regular, nonpointing variables, an array makes processing several pointer variables much easier. You can use a subscript to reference each variable (element) without having to use a different variable name for each value.

## Review Questions

Answers to review questions are in Appendix B.



1. What type of variable is reserved in each of the following?

   a. `int *a;`

   b. `char * cp;`

   c. `float * dp;`

2. What words should come to mind when you see the `&` operator?

3. What is the dereferencing operator?

4. How would you assign the address of the floating-point variable `salary` to a pointer called `pt_sal`?

5. True or false: You must define a pointer with an initial value when declaring it.

6. In both of the following sections of code:

```
int i;
int * pti;
i=56;
pti = &i;
```

and

```
int i;
int * pti;
pti = &i;          // These two lines are reversed
i=56;              // from the preceding example.
```

is the value of `pti` the same after the fourth line of each section?

7. In the following section of code:

```
float pay;
float *ptr_pay;
pay=2313.54;
ptr_pay = &pay;
```

What is the value of each of the following (answer "invalid" if it cannot be determined):

a. `pay`

b. `*ptr_pay`

c. `*pay`

d. `&pay`

8. What does the following declare?

```
double *ara[4][6];
```

a. An array of double floating-point values

b. An array of double floating-point pointer variables

c. An invalid declaration statement

**NOTE:** Because this is a theory-oriented chapter, review exercises are saved until you master Chapter 27, "Pointers and Arrays."

## Summary

Declaring and using pointers might seem troublesome at this point. Why assign `*p_num` a value when it is easier (and clearer) to assign a value directly to `num`? If you are asking yourself that question, you probably understand everything you should from this chapter and are ready to begin learning the true power of pointers: combining pointers and array processing.

# Pointers and Arrays

Arrays and pointers are closely related in the C++ programming language. You can address arrays as if they were pointers and address pointers as if they were arrays. Being able to store and access pointers and arrays gives you the ability to store strings of data in array elements. Without pointers, you could not store strings of data in arrays because there is no fundamental string data type in C++ (no string variables, only string literals).

This chapter introduces the following concepts:

- Array names and pointers
- Character pointers
- Pointer arithmetic
- Ragged-edge arrays of string data

This chapter introduces concepts you will use for much of your future programming in C++. Pointer manipulation is important to the C++ programming language.

# Array Names as Pointers

An array name is just a pointer, nothing more. To prove this, suppose you have the following array declaration:

```
int ara[5] = {10, 20, 30, 40, 50};
```

If you printed `ara[0]`, you would see `10`. Because you now fully understand arrays, this is the value you would expect.

An array name is a pointer.

But what if you were to print `*ara`? Would `*ara` print anything? If so, what? If you thought an error message would print because `ara` is not a pointer but an array, you would be wrong. An array name is a pointer. If you print `*ara`, you also would see `10`.

Recall how arrays are stored in memory. Figure 27.1 shows how `ara` would be mapped in memory. The array name, `ara`, is nothing more than a pointer pointing to the first element of the array. Therefore, if you dereference that pointer, you dereference the value stored in the first element of the array, which is `10`. Dereferencing `ara` is exactly the same thing as referencing to `ara[0]`, because they both produce the same value.

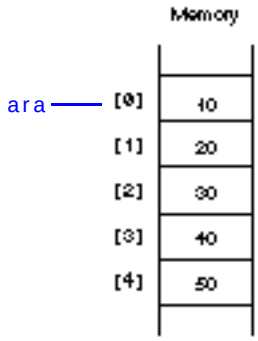



Figure 27.1. Storing the array called `ara` in memory.

You now see that you can reference an array with subscripts or with pointer dereferencing. Can you use pointer notation to print the third element of `ara`? Yes, and you already have the tools to do so. The following `cout` prints `ara[2]` (the third element of `ara`) without using a subscript:

```
cout << *(ara+2) ;              // Prints ara[2].
```

The expression `*(ara+2)` is not vague at all, if you remember that an array name is just a pointer that always points to the array's first element. `*(ara+2)` takes the address stored in `ara`, adds two to the address, and dereferences that location. The following holds true:

`ara+0` points to `ara[0]`

`ara+1` points to `ara[1]`

`ara+2` points to `ara[2]`

`ara+3` points to `ara[3]`

`ara+4` points to `ara[4]`

Therefore, to print, store, or calculate with an array element, you can use either the subscript notation or the pointer notation. Because an array name contains the address of the array's first element, you must dereference the pointer to get the element's value.

**Internal Locations**

C++ knows the internal data size requirements of characters, integers, floating-points, and the other data types on your computer. Therefore, because `ara` is an integer array, and because each element in an integer array consumes two to four bytes of storage, depending on the computer, C++ adds two or four bytes to the address if you reference arrays as just shown.

If you write `*(ara+3)` to refer to `ara[3]`, C++ would add six or twelve bytes to the address of `ara` to get the third element. C++ does not add an actual three. You do not have to worry about this, because C++ handles these internals. When you write `*(ara+3)`, you are actually requesting that C++ add three integer addresses to the address of `ara`. If `ara` were a floating-point array, C++ would add three floating-point addresses to `ara`.

# Pointer Advantages

An array name is a pointer constant.

Although arrays are actually pointers in disguise, they are special types of pointers. An array name is a *pointer constant*, not a pointer variable. You cannot change the value of an array name, because you cannot change constants. This explains why you cannot assign an array new values during a program's execution. For instance, even if `cname` is a character array, the following is not valid in C++:

```
cname = "Christine Chambers";  // Invalid array assignment.
```

The array name, `cname`, cannot be changed because it is a constant. You would not attempt the following

```
5 = 4 + 8 * 21;                // Invalid assignment
```

because you cannot change the constant `5` to any other value. C++ knows that you cannot assign anything to `5`, and C++ prints an error message if you attempt to change `5`. C++ also knows an array name is a constant and you cannot change an array to another value. (You can assign values to an array only at declaration time, one element at a time during execution, or by using functions such as `strcpy()`.)

This brings you to the most important reason to learn pointers: pointers (except arrays referenced as pointers) are variables. You can change a pointer variable, and being able to do so makes processing virtually any data, including arrays, much more powerful and flexible.

## Examples

1. By changing pointers, you make them point to different values in memory. The following program demonstrates how to change pointers. The program first defines two floating-point values. A floating-point pointer points to the first variable, `v1`, and is used in the `cout`. The pointer is then changed so it points to the second floating-point variable, `v2`.

```
// Filename: C27PTRCH.CPP
// Changes the value of a pointer variable.
#include <iostream.h>
```

```
#include <iomanip.h>
void main()
{
   float v1=676.54;                          // Defines two
   float v2=900.18;              // floating-point variables.
   float * p_v;          / Defines a floating-point pointer.

   p_v = &v1;                   // Makes pointer point to v1.
   cout << "The first value is " << setprecision(2) <<
           *p_v << "\n";                  // Prints 676.54.

   p_v = &v2;              // Changes the pointer so it
                           // points to v2.
   cout << "The second value is " << setprecision(2) <<
           *p_v << "\n";                  // Prints 900.18.
   return;
}
```

Because they can change pointers, most C++ programmers use pointers rather than arrays. Because arrays are easy to declare, C++ programmers sometimes declare arrays and then use pointers to reference those arrays. If the array data changes, the pointer helps to change it.

2. You can use pointer notation and reference pointers as arrays with array notation. The following program declares an integer array and an integer pointer that points to the start of the array. The array and pointer values are printed using subscript notation. Afterwards, the program uses array notation to print the array and pointer values.

   Study this program carefully. You see the inner workings of arrays and pointer notation.

```
// Filename: C27ARPTR.CPP
// References arrays like pointers and
// pointers like arrays.
#include <iostream.h>
void main()
{
   int ctr;
   int iara[5] = {10, 20, 30, 40, 50};
```

```
   int *iptr;

   iptr = iara;        // Make iptr point to array's first
                       // element. This would work also:
                       // iptr = &iara[0];

   cout << "Using array subscripts:\n";
   cout << "iara\tiptr\n";
   for (ctr=0; ctr<5; ctr++)
      { cout << iara[ctr] << "\t" << iptr[ctr] << "\n";  }

   cout << "\nUsing pointer notation:\n";
   cout << "iara\tiptr\n";
   for (ctr=0; ctr<5; ctr++)
      { cout << *(iara+ctr) << "\t" << *(iptr+ctr) << "\n";  }

 return;
}
```

Here is the program's output:

```
Using array subscripts:
iara    iptr
10      10
20      20
30      30
40      40
50      50

Using pointer notation:
iara    iptr
10      10
20      20
30      30
40      40
50      50
```

## Using Character Pointers

The ability to change pointers is best seen when working with character strings in memory. You can store strings in character arrays, or point to them with character pointers. Consider the following two string definitions:

```
char cara[] = "C++ is fun";     // An array holding a string

char *cptr = "C++ By Example";    // A pointer to the string
```

Character pointers can point to the first character of a string.

Figure 27.2 shows how C++ stores these two strings in memory. C++ stores both in basically the same way. You are familiar with the array definition. When assigning a string to a character pointer, C++ finds enough free memory to hold the string and assign the address of the first character to the pointer. The previous two string definition statements do almost exactly the same thing; the only difference between them is that the two pointers can easily be exchanged (the array name and the character pointers).

Because `cout` prints strings starting at the array or pointer name until the null zero is reached, you can print each of these strings with the following `cout` statements:

```
cout << "String 1: " << cara << "\n";

cout << "String 2: " << cptr << "\n";
```

You print strings in arrays and pointed-to strings the same way. You might wonder what advantage one method of storing strings has over the other. The seemingly minor difference between these stored strings makes a big difference when you change them.

Suppose you want to store the string `Hello` in the two strings. You cannot assign the string to the array like this:

```
cara = "Hello";                          // Invalid
```

Because you cannot change the array name, you cannot assign it a new value. The only way to change the contents of the array is by assigning the array characters from the string an element at a time, or by using a built-in function such as `strcpy()`. You can, however, make the character array point to the new string like this:

```
cptr = “Hello”;          // Change the pointer so
                         // it points to the new string.
```


Figure 27.2. Storing two strings: One in an array and one pointed to by a pointer variable.

**TIP:** If you want to store user input in a string pointed to by a pointer, first you must reserve enough storage for that input string. The easiest way to do this is to reserve a character array, then assign a character pointer to the beginning element of that array like this:

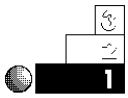
```
char input[81];          // Holds a string as long as
                         // 80 characters.
char *iptr=input;        // Also could have done this:
                         // char *iptr=&input[0];
```

Now you can input a string by using the pointer:

```
gets(iptr);              // Make sure iptr points to
                         // the string typed by the user.
```

You can use pointer manipulation, arithmetic, and modification on the input string.

## Examples



1. Suppose you want to store your sister's full name and print it. Rather than using arrays, you can use a character pointer. The following program does just that.

```
// Filename: C27CP1.CPP
// Stores a name in a character pointer.
#include <iostream.h>
void main()
{
   char *c="Bettye Lou Horn";

   cout << "My sister's name is " << c << "\n";
   return;
}
```

This prints the following:

```
My sister's name is Bettye Lou Horn
```

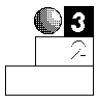
2. Suppose you must change a string pointed to by a character pointer. If your sister changed her last name to Henderson, your program can show both strings in the following manner:

*Identify the program and include the I/O header file. This program uses a character pointer,* c*, to point to a string literal in memory. Point to the string literal, and print the string. Make the character-pointer point to a new string literal, then print the new string.*

```
// Filename: C27CP2.CPP
// Illustrates changing a character string.
#include <iostream.h>
void main()
{
   char *c="Bettye Lou Horn";

   cout << "My sister's maiden name was " << c << "\n";

   c = "Bettye Lou Henderson";  // Assigns new string to c.

   cout << "My sister's married name is " << c << "\n";
   return;
}
```

The output is as follows:

```
My sister's maiden name was Bettye Lou Horn
My sister's married name is Bettye Lou Henderson
```



3. Do not use character pointers to change string constants. Doing so can confuse the compiler, and you probably will not get the results you expect. The following program is similar to those you just saw. Rather than making the character pointer point to a new string, this example attempts to change the contents of the original string.

```
// Filename: C27CP3.CPP
// Illustrates changing a character string improperly.
#include <iostream.h>
void main()
```

```
{
   char *c="Bettye Lou Horn";

   cout << "My sister's maiden name was " << c << "\n";

   c += 11;             // Makes c point to the last name
                        //   (the twelfth character).
   c = "Henderson";          // Assigns a new string to c.

   cout << "My sister's married name is " << c << "\n";
   return;
}
```

The program seems to change the last name from Horn to Henderson, but it does not. Here is the output of this program:

```
My sister's maiden name was Bettye Lou Horn
My sister's married name is Henderson
```

Why didn't the full string print? Because the address pointed to by `c` was incremented by 11, `c` still points to `Henderson`, so that was all that printed.

4. You might guess at a way to fix the previous program. Rather than printing the string stored at `c` after assigning it to `Henderson`, you might want to decrement it by 11 so it points to its original location, the start of the name. The code to do this follows, but it does not work as expected. Study the program before reading the explanation.

```
// Filename: C27CP4.C
// Illustrates changing a character string improperly.
#include <iostream.h>
void main()
{
   char *c="Bettye Lou Horn";

   cout << "My sister's maiden name was " << c << "\n";

   c += 11;                  // Makes c point to the last
                             // name (the twelfth character).
```

```
    c = "Henderson";          // Assigns a new string to c.
    c -= 11;                      // Makes c point to its
                                // original location (???).

    cout << "My sister's married name is " << c << "\n";
    return;
}
```

This program produces garbage at the second `cout`. There are actually two string literals in this program. When you first assign `c` to `Bettye Lou Horn`, C++ reserves space in memory for the constant string and puts the starting address of the string in `c`.

When the program then assigns `c` to `Henderson`, C++ finds room for *another* character constant, as shown in Figure 27.3. If you subtract 11 from the location of `c`, after it points to the new string `Henderson`, `c` points to an area of memory your program is not using. There is no guarantee that printable data appears before the string constant `Henderson`. If you want to manipulate parts of the string, you must do so an element at a time, just as you must with arrays.

## Pointer Arithmetic

You saw an example of pointer arithmetic when you accessed array elements with pointer notation. By now you should be comfortable with the fact that both of these array or pointer references are identical:

```
ara[sub] and *(ara + sub)
```

You can increment or decrement a pointer. If you increment a pointer, the address inside the pointer variable increments. The pointer does not always increment by one, however.

Suppose `f_ptr` is a floating-point pointer indexing the first element of an array of floating-point numbers. You could initialize `f_ptr` as follows:

```
float fara[] = {100.5, 201.45, 321.54, 389.76, 691.34};
f_ptr = fara;
```

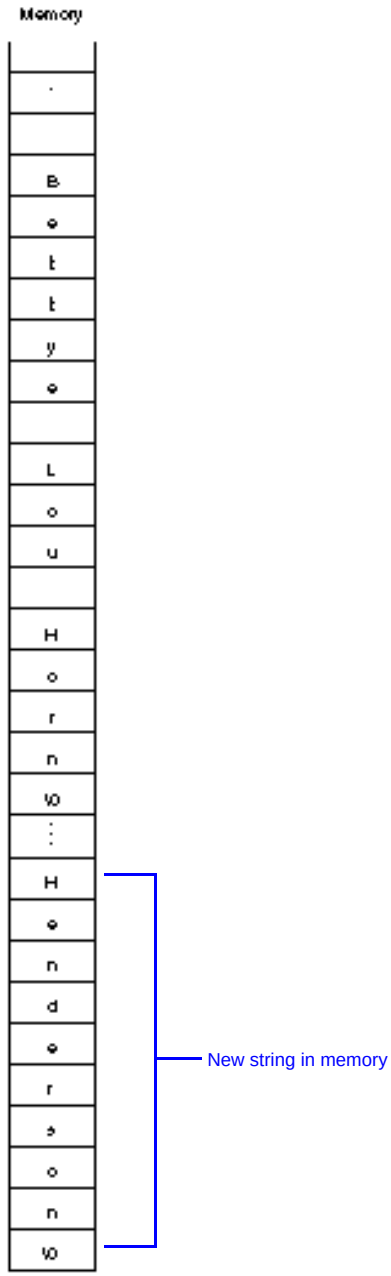



Figure 27.3. Two string constants appear in memory because two string constants are used in the program.

Figure 27.4 shows what these variables look like in memory. Each floating-point value in this example takes four bytes of memory.

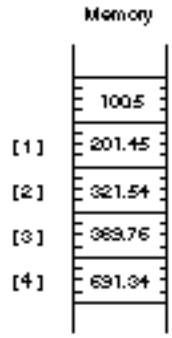



Figure 27.4. A floating-point array and a pointer.

Incrementing a pointer can add more than one byte to the pointer.

If you print the value of `*f_ptr`, you see 100.5. Suppose you increment `f_ptr` by one with the following statement:

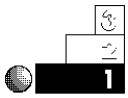
```
f_ptr++;
```

C++ does not add one to the address in `f_ptr`, even though it seems as though one should be added. In this case, because floating-point values take four bytes each on this machine, C++ adds four to `f_ptr`. How does C++ know how many bytes to add to `f_ptr`? C++ knows from the pointer's declaration how many bytes of memory pointers take. This is why you have to declare the pointer with the correct data type.

After incrementing `f_ptr`, if you were to print `*f_ptr`, you would see `201.45`, the second element in the array. If C++ added only one to the address in `f_ptr`, `f_ptr` would point only to the second byte, `100.5`. This would output garbage to the screen.

**NOTE:** When you increment a pointer, C++ adds one full data-type size (in bytes) to the pointer, not one byte. When you decrement a pointer, C++ subtracts one full data type size (in bytes) from the pointer.

## Examples



1. The following program defines an array with five values. An integer pointer is then initialized to point to the first element in the array. The rest of the program prints the dereferenced value of the pointer, then increments the pointer so it points to the next integer in the array.

   Just to show you what is going on, the size of integer values is printed at the bottom of the program. Because (in this case) integers take two bytes, C++ increments the pointer by two so it points to the next integer. (The integers are two bytes apart from each other.)

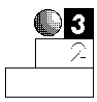
```
// Filename: C27PTI.CPP
// Increments a pointer through an integer array.
#include <iostream.h>
void main()
{
   int iara[] = {10,20,30,40,50};
   int *ip = iara;              // The pointer points to
                                // The start of the array.
   cout << *ip << "\n";
   ip++;                           // Two are actually added.
   cout << *ip << "\n";
   ip++;                           // Two are actually added.
   cout << *ip << "\n";
   ip++;                           // Two are actually added.
   cout << *ip << "\n";
   ip++;                           // Two are actually added.
   cout << *ip << "\n\n";
   cout << "The integer size is " << sizeof(int);
   cout << " bytes on this machine \n\n";
   return;
}
```

Here is the output from the program:

```
10
20
30
40
50

The integer size is two bytes on this machine
```

2. Here is the same program using a character array and a character pointer. Because a character takes only one byte of storage, incrementing a character pointer actually adds just one to the pointer; only one is needed because the characters are only one byte apart.

```
// Filename: C27PTC.CPP
// Increments a pointer through a character array.
#include <iostream.h>
void main()
{
   char cara[] = {'a', 'b', 'c', 'd', 'e'};
   char *cp = cara;             // The pointers point to
                                // the start of the array.
   cout << *cp << "\n";
   cp++;                           // One is actually added.
   cout << *cp << "\n";
   cp++;                           // One is actually added.
   cout << *cp << "\n";
   cp++;                           // One is actually added.
   cout << *cp << "\n";
   cp++;                           // One is actually added.
   cout << *cp << "\n\n";
   cout << "The character size is " << sizeof(char);
   cout << " byte on this machine\n";
   return;
}
```



3. The next program shows the many ways you can add to, subtract from, and reference arrays and pointers. The program defines a floating-point array and a floating-point pointer. The body of the program prints the values from the array using array and pointer notation.

```
// Filename: C27ARPT2.CPP
// Comprehensive reference of arrays and pointers.
#include <iostream.h>
void main()
{
   float ara[] = {100.0, 200.0, 300.0, 400.0, 500.0};
   float *fptr;                  // Floating-point pointer.

   // Make pointer point to array's first value.
   fptr = &ara[0];           // Also could have been this:
                             // fptr = ara;

   cout << *fptr << "\n";                  // Prints 100.0
   fptr++;          // Points to next floating-point value.
   cout << *fptr << "\n";                  // Prints 200.0
   fptr++;          // Points to next floating-point value.
   cout << *fptr << "\n";                  // Prints 300.0
   fptr++;          // Points to next floating-point value.
   cout << *fptr << "\n";                  // Prints 400.0
   fptr++;          // Points to next floating-point value.
   cout << *fptr << "\n";                  // Prints 500.0

   fptr = ara;            // Points to first element again.
   cout << *(fptr+2) << "\n";          // Prints 300.00 but
                                  // does not change fptr.

   // References both array and pointer using subscripts.
   cout << (fptr+0)[0] << " " << (ara+0)[0] << "\n";
   // 100.0  100.0
   cout << (fptr+1)[0] << " " << (ara+1)[0] << "\n";
   // 200.0  200.0
   cout << (fptr+4)[0] << " " << (ara+4)[0] << "\n";
   // 500.0  500.0
   return;
}
```

The following is the output from this program:

```
100.0
200.0
300.0
400.0
```

```
500.0
300.0
100.0   100.0
200.0   200.0
500.0   500.0
```

## Arrays of Strings

You now are ready for one of the most useful applications of character pointers: storing arrays of strings. Actually, you cannot store an array of strings, but you can store an array of character pointers, and each character pointer can point to a string in memory.

An array that a character pointer defines is a ragged-edge array.

By defining an array of character pointers, you define a *ragged-edge array*. A ragged-edge array is similar to a two-dimensional table, except each row contains a different number of characters (instead of being the same length).

The word *ragged-edge* derives from the use of word processors. A word processor typically can print text fully justified or with a ragged-right margin. The columns of this paragraph are fully justified, because both the left and the right columns align evenly. Letters you write by hand and type on typewriters (remember what a typewriter is?) generally have ragged-right margins. It is difficult to type so each line ends in exactly the same right column.

All two-dimensional tables you have seen so far have been fully justified. For example, if you declared a character table with five rows and 20 columns, each row would contain the same number of characters. You could define the table with the following statement:

```
char names[5][20]={ {“George”},
                    {“Michelle”},
                    {“Joe”},
                    {“Marcus”},
                    {“Stephanie”} };
```

This table is shown in Figure 27.5. Notice that much of the table is wasted space. Each row takes 20 characters, even though the data in each row takes far fewer characters. The unfilled elements contain null zeros because C++ nullifies all elements you do not initialize in arrays. This type of table uses too much memory.

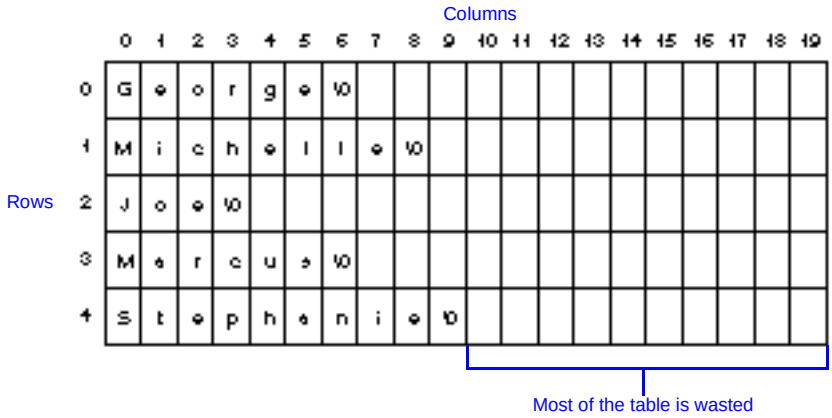



Figure 27.5. A fully justified table.

To fix the memory-wasting problem of fully justified tables, you should declare a single-dimensional array of character pointers. Each pointer points to a string in memory, and the strings do not have to be the same length.

Here is the definition for such an array:

```
char *names[5]={ {“George”},
                 {“Michelle”},
                 {“Joe”},
                 {“Marcus”},
                 {“Stephanie”} };
```

This array is single-dimensional. The definition should not confuse you, although it is something you have not seen. The asterisk before `names` makes this an array of pointers. The data type of the pointers is character. The strings are not being assigned to the array elements, but they are being pointed to by the array elements. Figure 27.6 shows this array of pointers. The strings are stored elsewhere in memory. Their actual locations are not critical because each pointer points to the starting character. The strings waste no data. Each string takes only as much memory as needed by the string and its terminating zero. This gives the data its ragged-right appearance.

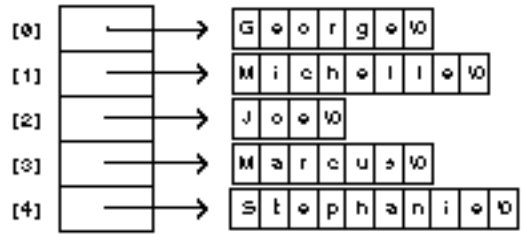



Figure 27.6. The array that points to each of the five strings.

To print the first string, you would use this `cout`:

```
cout << *names;              // Prints George
```

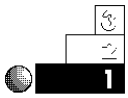
To print the second string, you would use this `cout`:

```
cout << *(names+1);          // Prints Michelle
```

Whenever you dereference any pointer element with the `*` dereferencing operator, you access one of the strings in the array. You can use a dereferenced element any place you use a string constant or character array (with `strcpy()`, `strcmp()`, and so on).

**TIP:** Working with pointers to strings is much more efficient than working directly with the strings. For instance, sorting a list of strings takes much time if they are stored as a fully justified table. Sorting strings pointed to by a pointer array is much faster. You swap only pointers during the sort, not entire strings.

## Examples



1. Here is a full program that uses the pointer array with five names. The `for` loop controls the `cout` function, printing each name in the string data. Now you can see why learning about pointer notation for arrays pays off!

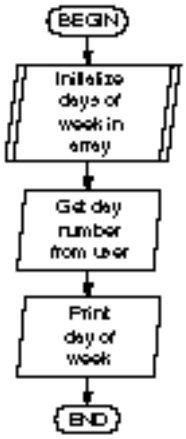
```
// Filename: C27PTST1.CPP
// Prints strings pointed to by an array.
#include <iostream.h>
```

```
void main()
{
   char *name[5]={ {"George"},     // Defines a ragged-edge
                   {"Michelle"},   // array of pointers to
                   {"Joe"},        // strings.
                   {"Marcus"},
                   {"Stephanie"} };
   int ctr;

   for (ctr=0; ctr<5; ctr++)
      { cout << "String #" << (ctr+1) <<
                "  is " << *(name+ctr) << "\n"; }

   return;
}
```

The following is the output from this program:

```
String #1 is George
String #2 is Michelle
String #3 is Joe
String #4 is Marcus
String #5 is Stephanie
```


2. The following program stores the days of the week in an array. When the user types a number from 1 to 7, the day of the week that matches that number (with Sunday being 1) displays by dereferencing the pointer referencing that string.

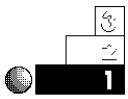



```
// Filename: C27PTST2.CPP
// Prints the day of the week based on an input value.
#include <iostream.h>
void main()
{
   char *days[] = {"Sunday",    // The seven separate sets
              "Monday",         // of braces are optional.
              "Tuesday",
              "Wednesday",
              "Thursday",
              "Friday",
              "Saturday"};
   int day_num;
```

```
    do
      { cout << "What is a day number (from 1 to 7)? ";
        cin >> day_num;
      } while ((day_num<1) || (day_num>7));      // Ensures
                                          // an accurate number.


    day_num--;                        // Adjusts for subscript.
    cout << "The day is " << *(days+day_num) << "\n";
    return;
}
```

## Review Questions

The answers to the review questions are in Appendix B.



1. What is the difference between an array name and a pointer?
2. If you performed the following statement (assume `ipointer` points to integers that take four bytes of memory),

   ```
   ipointer += 2;
   ```

   how many bytes are added to `ipointer`?

3. Which of the following are equivalent, assuming `iary` is an integer array and `iptr` is an integer pointer pointing to the start of the array?

   a. `iary` and `iptr`

   b. `iary[1]` and `iptr+1`

   c. `iary[3]` and `*(iptr + 3)`

   d. `*iary` and `iary[0]`

   e. `iary[4]` and `*iptr+4`

4. Why is it more efficient to sort a ragged-edge character array than a fully justified string array?

5. Given the following array and pointer definition

```
int ara[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *ip1, *ip2;
```

which of the following is allowed?

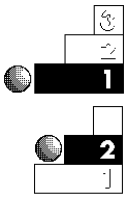
a. `ip1 = ara;`

b. `ip2 = ip1 = &ara[3];`

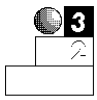
c. `ara = 15;`

d. `*(ip2 + 2) = 15;  // Assuming ip2 and ara are equal.`

## Review Exercises



1. Write a program to store your family members' names in a character array of pointers. Print the names.
2. Write a program that asks the user for 15 daily stock market averages and stores those averages in a floating-point array. Using only pointer notation, print the array forward and backward. Again using only pointer notation, print the highest and lowest stock market quotes in the list.
3. Modify the bubble sort shown in Chapter 24, "Array Processing," so that it sorts using pointer notation. Add this bubble sort to the program in Exercise 2 to print the stock market averages in ascending order



4. Write a program that requests 10 song titles from the user. Store the titles in an array of character pointers (a ragged-edge array). Print the original titles, print the alphabetized titles, and print the titles in reverse alphabetical order (from *Z* to *A*).

## Summary

You deserve a break! You now understand the foundation of C++'s pointers and array notation. When you have mastered this section, you are on your way to thinking in C++ as you design your programs. C++ programmers know that C++'s arrays are pointers in disguise, and they program them accordingly.

Being able to use ragged-edge arrays offers two advantages: You can hold arrays of string data without wasting extra space, and you can quickly change the pointers without having to move the string data around in memory.

As you progress into advanced C++ concepts, you will appreciate the time you spend mastering pointer notation. The next chapter introduces a new topic called *structures*. Structures enable you to store data in a more unified manner than simple variables have allowed.