Variables and Literals

To understand data processing with C++, you must understand how C++ creates, stores, and manipulates data. This chapter teaches you how C++ handles data by introducing the following topics:

- The concepts of variables and literals
- The types of C++ variables and literals
- Special literals
- Constant variables
- Naming and using variables
- Declaring variables
- Assigning values to variables

Garbage in, garbage out!

Now that you have seen an overview of the C++ programming language, you can begin writing C++ programs. In this chapter, you begin to write your own programs from scratch.

You learned in Chapter 3, "Your First C++ Program," that C++ programs consist of commands and data. Datum is the heart of all C++ programs; if you do not correctly declare or use variables and literals, your data are inaccurate and your results are going to be

inaccurate as well. A computer adage says the if you put garbage in, you are going to get garbage out. This is very true. People usually blame computers for mistakes, but the computers are not always at fault. Rather, their data are often not entered properly into their programs.

This chapter spends a long time focusing on numeric variables and numeric literals. If you are not a "numbers" person, do not fret. Working with numbers is the computer's job. You have to understand only how to tell the computer what you want it to do.

Variables

Variables have characteristics. When you decide your program needs another variable, you simply declare a new variable and C++ ensures that you get it. In C++, variable declarations can be placed anywhere in the program, as long as they are not referenced until after they are declared. To declare a variable, you must understand the possible characteristics, which follow.

- Each variable has a name.
- Each variable has a type.
- Each variable holds a value that you put there, by assigning it to that variable.

The following sections explain each of these characteristics in detail.

Naming Variables

Because you can have many variables in a single program, you must assign names to them to keep track of them. Variable names are unique, just as house addresses are unique. If two variables have the same name, C++ would not know to which you referred when you request one of them.

Variable names can be as short as a single letter or as long as 32 characters. Their names must begin with a letter of the alphabet but, after the first letter, they can contain letters, numbers, and underscore (_) characters.

TIP: Spaces are not allowed in a variable name, so use the underscore character to separate parts of the name.

The following list of variable names are all valid:

```
salary   aug91_sales   i   index_age   amount
```

It is traditional to use lowercase letters for C++ variable names. You do not have to follow this tradition, but you should know that uppercase letters in variable names are different from lowercase letters. For example, each of the following four variables is viewed differently by your C++ compiler.

```
sales   Sales   SALES   sALES
```

Be very careful with the Shift key when you type a variable name. Do not inadvertently change the case of a variable name throughout a program. If you do, C++ interprets them as distinct and separate variables.

Do not give variables the same name as a command or built-in function.

Variables cannot have the same name as a C++ command or function. Appendix E, “Keyword and Function Reference,” shows a list of all C++ command and function names.

The following are *invalid* variable names:

```
81_sales      Aug91+Sales      MY AGE      printf
```

TIP: Although you can call a variable any name that fits the naming rules (as long as it is not being used by another variable in the program), you should always use meaningful variable names. Give your variables names that help describe the values they are holding.

For example, keeping track of total payroll in a variable called `total_payroll` is much more descriptive than using the variable name `XYZ34`. Even though both names are valid, `total_payroll` is easier to remember and you have a good idea of what the variable holds by looking at its name.

Variable Types

Variables can hold different types of data. Table 4.1 lists the different types of C++ variables. For instance, if a variable holds an integer, C++ assumes no decimal point or fractional part (the part to the right of the decimal point) exists for the variable's value. A large number of types are possible in C++. For now, the most important types you should concentrate on are `char`, `int`, and `float`. You can append the prefix `long` to make some of them hold larger values than they would otherwise hold. Using the `unsigned` prefix enables them to hold only positive numbers.

Table 4.1. Some C++ variable types.

Declaration Name	*Type*
`char`	Character
`unsigned char`	Unsigned character
`signed char`	Signed character (same as `char`)
`int`	Integer
`unsigned int`	Unsigned integer
`signed int`	Signed integer (same as `int`)
`short int`	Short integer
`unsigned short int`	Unsigned short integer
`signed short int`	Signed short integer (same as `short int`)
`long`	Long integer
`long int`	Long integer (same as `long`)
`signed long int`	Signed long integer (same as `long int`)
`unsigned long int`	Unsigned long integer
`float`	Floating-point
`double`	Double floating-point
`long double`	Long double floating-point

The next section more fully describes each of these types. For now, you have to concentrate on the importance of declaring them before using them.

Declaring Variables

There are two places you can declare a variable:

- Before the code that uses the variable
- Before a function name (such as before `main()` in the program)

The first of these is the most common, and is used throughout much of this book. (If you declare a variable before a function name, it is called a *global* variable. Chapter 17, "Variable Scope," addresses the pros and cons of global variables.) To declare a variable, you must state its type, followed by its name. In the previous chapter, you saw a program that declared four variables in the following way.

Start of the `main()` *function.*
Declare the variables `i` *and* `j` *as integers.*
Declare the variable `c` *as a character.*
Declare the variable `x` *as a floating-point variable.*

```
main()
{
   int i, j;    // These three lines declare four variables.
   char c;
   float x;
   // The rest of program follows.
```

This declares two integer variables named `i` and `j`. You have no idea what is inside those variables, however. You generally cannot assume a variable holds zero—or any other number—until you assign it a value. The first line basically tells C++ the following:

Declare all variables in a C++ program before you use them.

"I am going to use two integer variables somewhere in this program. Be expecting them. I want them named `i` and `j`. When I put a value into `i` or `j`, I ensure that the value is an integer."

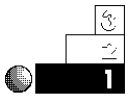
Without such a declaration, you could not assign `i` or `j` a value later. All variables must be declared before you use them. This does not necessarily hold true in other programming languages, such as BASIC, but it does for C++. You could declare each of these two variables on its own line, as in the following code:

```
main()
{
   int i;
   int j;
   // The rest of program follows.
```

You do not gain any readability by doing this, however. Most C++ programmers prefer to declare variables of the same type on the same line.

The second line in this example declares a character variable called `c`. Only single characters should be placed there. Next, a floating-point variable called `x` is declared.

Examples



1. Suppose you had to keep track of a person's first, middle, and last initials. Because an initial is obviously a character, it would be prudent to declare three character variables to hold the three initials. In C++, you could do that with the following statement:

```
main()
{
   char first, middle, last;
   // The rest of program follows.
```

This statement could go after the opening brace of `main()`. It informs the rest of the program that you require these three character variables.

2. You could declare these three variables also on three separate lines, although it does not necessarily improve readability to do so. This could be accomplished with:

```
main()
{
   char first;
   char middle;
   char last;
   // The rest of program follows.
```

3. Suppose you want to keep track of a person's age and weight. If you want to store these values as whole numbers, they would probably go in integer variables. The following statement would declare those variables:

```
main()
{
   int age, weight;
   // The rest of program follows.
```

Looking at Data Types

You might wonder why it is important to have so many variable types. After all, a number is just a number. C++ has more data types, however, than almost all other programming languages. The variable's type is critical, but choosing the type among the many offerings is not as difficult as it might first seem.

The character variable is easy to understand. A character variable can hold only a single character. You cannot put more than a single character into a character variable.

NOTE: Unlike many other programming languages, C++ does not have a string variable. Also, you cannot hold more than a single character in a C++ character variable. To store a string of characters, you must use an *aggregate* variable type that combines other fundamental types, such as an array. Chapter 5, "Character Arrays and Strings," explains this more fully.

Integers hold whole numbers. Although mathematicians might cringe at this definition, an integer is actually any number that does

not contain a decimal point. All the following expressions are integers:

```
45      -932      0      12      5421
```

Floating-point numbers contain decimal points. They are known as *real* numbers to mathematicians. Any time you have to store a salary, a temperature, or any other number that might have a fractional part (a decimal portion), you must store it in a floating-point variable. All the following expressions are floating-point numbers, and any floating-point variable can hold them:

```
45.12    -2344.5432    0.00      .04594
```

Sometimes you have to keep track of large numbers, and sometimes you have to keep track of smaller numbers. Table 4.2 shows a list of ranges that each C++ variable type can hold.

CAUTION: All true AT&T C++ programmers know that they cannot count on using the exact values in Table 4.2 on every computer that uses C++. These ranges are typical on a PC, but might be much different on another computer. Use this table only as a guide.

Table 4.2. Typical ranges that C++ variables hold.

Type	*Range**
`char`	–128 to 127
`unsigned char`	0 to 255
`signed char`	–128 to 127
`int`	–32768 to 32767
`unsigned int`	0 to 65535
`signed int`	–32768 to 32767
`short int`	–32768 to 32767
`unsigned short int`	0 to 65535

Type	*Range**
`signed short int`	–32768 to 32767
`long int`	–2147483648 to 2147483647
`signed long int`	–2147483648 to 2147483647
`float`	–3.4E–38 to 3.4E+38
`double`	–1.7E–308 to 1.7E+308
`long double`	–3.4E–4932 to 1.1E+4932

** Use this table only as a guide; different compilers and different computers can have different ranges.*

NOTE: The floating-point ranges in Table 4.2 are shown in scientific notation. To determine the actual range, take the number before the *E* (meaning *Exponent*) and multiply it by 10 raised to the power after the plus sign. For instance, a floating-point number (type `float`) can contain a number as small as -3.4^{38}.

Notice that long integers and long doubles tend to hold larger numbers (and therefore, have a higher precision) than regular integers and regular double floating-point variables. This is due to the larger number of memory locations used by many of the C++ compilers for these data types. Again, this is usually—but not always—the case.

Do Not Over Type a Variable

If the long variable types hold larger numbers than the regular ones, you might initially want to use long variables for all your data. This would not be required in most cases, and would probably slow your program's execution.

As Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," describes, the more memory locations used by data, the larger that data can be. However, every time your computer has to access more storage for a single variable (as is usually the case for long variables), it takes the CPU much longer to access it, calculate with it, and store it.

Use the long variables only if you suspect your data might overflow the typical data type ranges. Although the ranges differ between computers, you should have an idea of whether you numbers might exceed the computer's storage ranges. If you are working with extremely large (or extremely small and fractional) numbers, you should consider using the long variables.

Generally, all numeric variables should be signed (the default) unless you know for certain that your data contain only positive numbers. (Some values, such as age and distances, are always positive.) By making a variable an unsigned variable, you gain a little extra storage range (as explained in Appendix A, "Memory Addressing, Binary, and Hexadecimal Review"). That range of values must always be positive, however.

Obviously, you must be aware of what kinds of data your variables hold. You certainly do not always know exactly what each variable is holding, but you can have a general idea. For example, in storing a person's age, you should realize that a long integer variable would be a waste of space, because nobody can live to an age that can't be stored by a regular integer.

At first, it might seem strange for Table 4.2 to state that character variables can hold numeric values. In C++, integers and character variables frequently can be used interchangeably. As explained in Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," each ASCII table character has a unique number that corresponds to its location in the table. If you store a number in a character variable, C++ treats the data as if it were the ASCII character that matched that number in the table. Conversely, you can store character data in an integer variable. C++ finds that

character's ASCII number, and stores that number rather than the character. Examples that help illustrate this appear later in the chapter.

Designating Long, Unsigned, and Floating-Point Literals

When you type a number, C++ interprets its type as the smallest type that can hold that number. For example, if you print `63`, C++ knows that this number fits into a signed integer memory location. It does not treat the number as a long integer, because `63` is not large enough to warrant a long integer literal size.

However, you can append a suffix character to numeric literals to override the default type. If you put an `L` at the end of an integer, C++ interprets that integer as a long integer. The number `63` is an integer literal, but the number `63L` is a long integer literal.

Assign the `U` suffix to designate an unsigned integer literal. The number `63` is, by default, a signed integer literal. If you type `63U`, C++ treats it as an unsigned integer. The suffix `UL` indicates an unsigned long literal.

C++ interprets all floating-point literals (numbers that contain decimal points) as double floating-point literals (double floating-point literals hold larger numbers than floating-point literals). This process ensures the maximum accuracy in such numbers. If you use the literal `6.82`, C++ treats it as a double floating-point data type, even though it would fit in a regular `float`. You can append the floating-point suffix (`F`) or the long double floating-point suffix (`L`) to literals that contain decimal points to represent a floating-point literal or a long double floating-point literal.

You may rarely use these suffixes, but if you have to assign a literal value to an extended or unsigned variable, your literals might be a little more accurate if you add `U`, `L`, `UL`, or `F` (their lowercase equivalents work too) to their ends.

Assigning Values to Variables

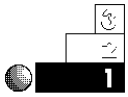
Now that you know about the C++ variable types, you are ready to learn the specifics of assigning values to those variables. You do this with the *assignment* statement. The equal sign (=) is used for assigning values to variables. The format of the assignment statement is


```
variable=expression;
```

The `variable` is any variable that you declared earlier. The `expression` is any variable, literal, expression, or combination that produces a resulting data type that is the same as the `variable`'s data type.

TIP: Think of the equal sign as a left-pointing arrow. Loosely, the equal sign means you want to take the number, variable, or expression on the right side of the equal sign and put it into the variable on the left side of the equal sign.

Examples



1. If you want to keep track of your current age, salary, and dependents, you could store these values in three C++ variables. You first declare the variables by deciding on correct types and good names for them. You then assign values to them. Later in the program, these values might change (for example, if the program calculates a new pay increase for you).

   Good variable names include `age`, `salary`, and `dependents`. To declare these three variables, the first part of the `main()` function would look like this:

```
// Declare and store three values.
main()
{
   int age;
   float salary;
   int dependents;
```

Notice that you do not have to declare all integer variables together. The next three statements assign values to the variables.

```
age=32;
salary=25000.00;
dependents=2;
// Rest of program follows.
```

This example is not very long and doesn't do much, but it illustrates the using and assigning of values to variables.

2. Do not put commas in values that you assign to variables. Numeric literals should never contain commas. The following statement is invalid:

```
salary=25,000.00;
```

3. You can assign variables or mathematical expressions to other variables. Suppose, earlier in a program, you stored your tax rate in a variable called `tax_rate`, then decided to use your tax rate for your spouse's rate as well. At the proper point in the program, you would code the following:

```
spouse_tax_rate = tax_rate;
```

(Adding spaces around the equal sign is acceptable to the C++ compiler, but you do not have to do so.) At this point in the program, the value in `tax_rate` is copied to a new variable named `spouse_tax_rate`. The value in `tax_rate` is still there after this line finishes. The variables were declared earlier in the program.

If your spouse's tax rate is 40 percent of yours, you can assign an expression to the spouse's variable, as in:

```
spouse_tax_rate = tax_rate * .40;
```

Any of the four mathematical symbols you learned in the previous chapter, as well as the additional ones you learn later in the book, can be part of the expression you assign to a variable.

4. If you want to assign character data to a character variable, you must enclose the character in single quotation marks. All C++ character literals must be enclosed in single quotation marks.

   The following section of a program declares three variables, then assigns three initials to them. The initials are character literals because they are enclosed in single quotation marks.

```
main()
{
   char first, middle, last;
   first = 'G';
   middle = 'M';
   last = 'P';
   // Rest of program follows.
```

   Because these are variables, you can reassign their values later if the program warrants it.

CAUTION: Do not mix types. C enables programmers to do this, but C++ does not. For instance, in the `middle` variable presented in the previous example, you could not have stored a floating-point literal:

```
middle = 345.43244;    // You cannot do this!
```

If you did so, `middle` would hold a strange value that would seem to be meaningless. Make sure that values you assign to variables match the variable's type. The only major exception to this occurs when you assign an integer to a character variable, or a character to an integer variable, as you learn shortly.

Literals

As with variables, there are several types of C++ literals. Remember that a literal does not change. Integer literals are whole numbers that do not contain decimal points. Floating-point literals

are numbers that contain a fractional portion (a decimal point with an optional value to the right of the decimal point).

Assigning Integer Literals

You already know that an integer is any whole number without a decimal point. C++ enables you to assign integer literals to variables, use integer literals for calculations, and print integer literals using the `cout` operator.

An octal integer literal contains a leading `0`, and a hexadecimal literal contains a leading `0x`.

A regular integer literal cannot begin with a leading `0`. To C++, the number `012` is not the number twelve. If you precede an integer literal with a `0`, C++ interprets it as an *octal* literal. An octal literal is a base-8 number. The octal numbering system is not used much in today's computer systems. The newer versions of C++ retain octal capabilities for compatibility with previous versions.

A special integer in C++ that is still greatly used today is the base-16, or *hexadecimal*, literal. Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," describes the hexadecimal numbering system. If you want to represent a hexadecimal integer literal, add the `0x` prefix to it. The following numbers are hexadecimal numbers:

```
0x10      0x2C4      0xFFFF      0X9
```

Notice that it does not matter if you use a lowercase or uppercase letter *x* after the leading zero, or an uppercase or lowercase hexadecimal digit (for hex numbers A through F). If you write business-application programs in C++, you might think you never have the need for using hexadecimal, and you might be correct. For a complete understanding of C++ and your computer in general, however, you should become a little familiar with the fundamentals of hexadecimal numbers.

Table 4.3 shows a few integer literals represented in their regular decimal, hexadecimal, and octal notations. Each row contains the same number in all three bases.

Table 4.3. Integer literals represented in three bases.

Decimal (Base 10)	*Hexadecimal (Base 16)*	*Octal (Base 8)*
16	0x10	020
65536	0x10000	0100000
25	0x19	031

NOTE: Floating-point literals can begin with a leading zero, for example, 0.7. They are properly interpreted by C++. Only integers can be hexadecimal or octal literals.

Your Computer's Word Size Is Important

If you write many system programs that use hexadecimal numbers, you probably want to store those numbers in *unsigned* variables. This keeps C++ from improperly interpreting positive numbers as negative numbers.

For example, if your computer stores integers in 2-byte words (as most PCs do), the hexadecimal literal 0xFFFF represents either –1 or 65535, depending on how the sign bit is interpreted. If you declared an unsigned integer, such as

```
unsigned_int i_num = 0xFFFF;
```

C++ knows you want it to use the sign bit as data and not as the sign. If you declared the same value as a signed integer, however, as in

```
int i_num = 0xFFFF;    /* The word "signed" is optional.*/
```

C++ thinks this is a negative number (–1) because the sign bit is on. (If you were to convert 0xFFFF to binary, you would get sixteen *1*s.) Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," discusses these concepts in more detail.

Assigning String Literals

A string literal is always enclosed in double quotation marks.

One type of C++ literal, called the *string literal*, does not have a matching variable. A string literal is always enclosed in double quotation marks. Here are examples of string literals:

```
"C++ Programming"   "123"   " "   "4323 E. Oak Road"   "x"
```

Any string of characters between double quotation marks—even a single character—is considered to be a string literal. A single space, a word, or a group of words between double quotation marks are all C++ string literals.

If the string literal contains only numeric digits, it is *not* a number; it is a string of numeric digits that you cannot use to perform mathematics. You can perform math only on numbers, not on string literals.

NOTE: A string literal is *any* character, digit, or group of characters enclosed in double quotation marks. A character literal is any character enclosed in single quotation marks.

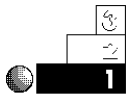
The double quotation marks are never considered part of the string literal. The double quotation marks surround the string and simply inform your C++ compiler that the code is a string literal and not another type of literal.

It is easy to print string literals. Simply put the string literals in a `cout` statement. The following code prints a string literal to the screen:

The following code prints the string literal, `C++ By Example`.

```
cout << "C++ By Example";
```

Examples

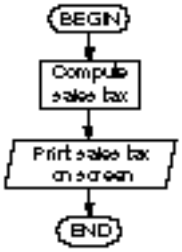


1. The following program displays a simple message on-screen. No variables are needed because no datum is stored or calculated.

```
// Filename: C4ST1.CPP
// Display a string on-screen.

#include <iostream.h>
main()
{
   cout << "C++ programming is fun!";
   return 0;
}
```

Remember to make the last line in your C++ program (before the closing brace) a `return` statement.

2. You probably want to label the output from your programs. Do not print the value of a variable unless you also print a string literal that describes that variable. The following program computes sales tax for a sale and prints the tax. Notice a message is printed first that tells the user what the next number means.




```
// Filename: C4ST2.CPP
// Compute sales tax and display it with an appropriate
message.

#include <iostream.h>
main()
{
   float sale, tax;
   float tax_rate = .08;    // Sales tax percentage

   // Determine the amount of the sale.
   sale = 22.54;

   // Compute the sales tax.
   tax = sale * tax_rate;

   // Print the results.
   cout << "The sales tax is " << tax << "\n";

   return 0;
}
```

Here is the output from the program:

```
The sales tax is 1.8032
```

You later learn how to print accurately to two decimal places to make the cents appear properly.

String-Literal Endings

An additional aspect of string literals sometimes confuses beginning C++ programmers. All string literals end with a zero. You do not see the zero, but C++ stores the zero at the end of the string in memory. Figure 4.1 shows what the string `"C++ Program"` looks like in memory.

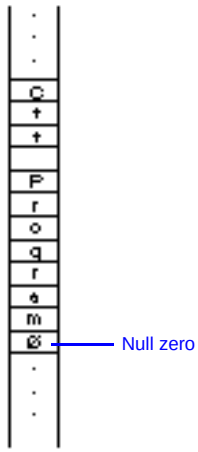


Figure 4.1. In memory, a string literal always ends with `0`.

You do not have to worry about putting the zero at the end of a string literal; C++ does it for you every time it stores a string. If your program contained the string `"C++ Program"`, for example, the compiler would recognize it as a string literal (from the double quotation marks) and store the zero at the end.

All string literals end in a null zero (also called binary zero or ASCII zero).

The zero is important to C++. It is called the *string delimiter*. Without it, C++ would not know where the string literal ended in memory. (Remember that the double quotation marks are not stored as part of the string, so C++ cannot use them to determine where the string ends.)

The string-delimiting zero is not the same as the character zero. If you look at the ASCII table in Appendix C, "ASCII Table," you can see that the first entry, ASCII number `0`, is the *null* character. (If you are unfamiliar with the ASCII table, you should read Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," for a brief description.) This string-delimiting zero is different from the from the character `'0'`, which has an ASCII value of 48.

As explained in Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," all memory locations in your computer actually hold bit patterns for characters. If the letter *A* is stored in memory, an *A* is not actually there; the binary bit pattern for the ASCII *A* (01000001) is stored there. Because the binary bit pattern for the null zero is 00000000, the string-delimiting zero is also called a *binary zero*.

To illustrate this further, Figure 4.2 shows the bit patterns for the following string literal when stored in memory: `"I am 30"`.

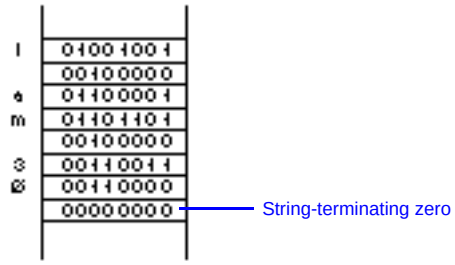


Figure 4.2. The bit pattern showing that a null zero and a character zero are different.

Figure 4.2 shows how a string is stored in your computer's memory at the binary level. It is important for you to recognize that the character 0, inside the number 30, is not the same zero (at the bit level) as the string-terminating null zero. If it were, C++ would think this string ended after the 3, which would be incorrect.

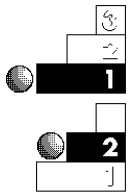
This is a fairly advanced concept, but you truly have to understand it before continuing. If you are new to computers, reviewing the material in Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," will help you understand this concept.

String Lengths

The length of a string literal does not include the null binary zero.

Many times, your program has to know the length of a string. This becomes critical when you learn how to accept string input from the keyboard. The length of a string is the number of characters up to, but not including, the delimiting null zero. Do *not* include the null character in that count, even though you know C++ adds it to the end of the string.

Examples



1. The following are all string literals:

```
"0"     "C"     "A much longer string literal"
```

2. The following table shows some string literals and their corresponding string lengths.

String	*Length*
`"C"`	1
`"0"`	21
`"Hello"`	5
`""`	0
`"30 oranges"`	10

Assigning Character Literals

All C character literals should be enclosed in single quotation marks. The single quotation marks are not part of the character, but they serve to delimit the character. The following are valid C++ character literals:

```
'w'  'W'  'C'  '7'  '*'  '='  '.'  'K'
```

C++ does not append a null zero to the end of character literals. You should know that the following are different to C++.

`'R'` and `"R"`

`'R'` is a single character literal. It is one character long, because *all* character literals (and variables) are one character long. `"R"` is a string literal because it is delimited by double quotation marks. Its length is also one, but it includes a null zero in memory so C++ knows where the string ends. Due to this difference, you cannot mix character literals and character strings. Figure 4.3 shows how these two literals are stored in memory.

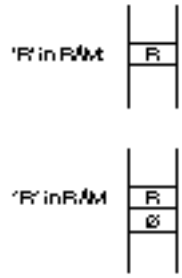


Figure 4.3. The difference in memory between `'R'` as a character literal and `"R"` as a string literal.

All the alphabetic, numeric, and special characters on your keyboard can be character literals. Some characters, however, cannot be represented with your keyboard. They include some of the higher ASCII characters (such as the Spanish Ñ). Because you do not have keys for every character in the ASCII table, C++ enables you to represent these characters by typing their ASCII hexadecimal number inside single quotation marks.

For example, to store the Spanish *Ñ* in a variable, look up its hexadecimal ASCII number from Appendix C, "ASCII Table." You find that it is `A5`. Add the prefix `\x` to it and enclose it in single quotation marks, so C++ will know to use the special character. You could do that with the following code:

```
char sn='\xA5'; // Puts the Spanish Ñ into a variable called sn.
```

This is the way to store (or print) any character from the ASCII table, even if that character does not have a key on your keyboard.

The single quotation marks still tell C++ that a *single* character is inside the quotation marks. Even though `'\xA5'` contains four characters inside the quotation marks, those four characters represent a single character, not a character string. If you were to include those four characters inside a string *literal*, C++ would treat `\xA5` as a single character in the string. The following string literal,

```
"An accented a is \xA0"
```

is a C++ string that is 18 characters, not 21 characters. C++ interprets the `\xA0` character as the á, just as it should.

CAUTION: If you are familiar with entering ASCII characters by typing their ASCII numbers with the Alt-keypad combination, do not do this in your C++ programs. They might work on your computer (not all C++ compilers support this), but your program might not be portable to another computer's C++ compiler.

Any character preceded by a backslash, `\`, (such as these have been) is called an *escape sequence*, or *escape character*. Table 4.4 shows some additional escape sequences that come in handy when you want to print special characters.

TIP: Include `"\n"` in a `cout` if you want to skip to the next line when printing your document.

Table 4.4. Special C++ escape-sequence characters.

Escape Sequence	*Meaning*
`\a`	Alarm (the terminal's bell)
`\b`	Backspace
`\f`	Form feed (for the printer)

continues

Table 4.4. Continued.

Escape Sequence	*Meaning*
\n	Newline (carriage return and line feed)
\r	Carriage return
\t	Tab
\v	Vertical tab
\\	Backslash (\)
\?	Question mark
\'	Single quotation mark
\"	Double quotation mark
\000	Octal number
\xhh	Hexadecimal number
\0	Null zero (or binary zero)

Math with C++ Characters

Because C++ links characters so closely with their ASCII numbers, you can perform arithmetic on character data. The following section of code,

```
char c;
c = 'T' + 5;      // Add five to the ASCII character.
```

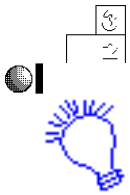
actually stores a *Y* in `c`. The ASCII value of the letter *T* is 84. Adding 5 to 84 produces 89. Because the variable `c` is not an integer variable, but is a character variable, C++ adds the ASCII character for 89, not the actual number.

Conversely, you can store character literals in integer variables. If you do, C++ stores the matching ASCII number for that character. The following section of code

```
int i='P';
```

does not put a letter *P* in `i` because `i` is not a character variable. C++ assigns the number 80 in the variable because 80 is the ASCII number for the letter *P*.

Examples



1. To print two names on two different lines, include the `\n` between them.

   Print the name `Harry`*; drop the cursor down to a new line and print* `Jerry`.

   ```
   cout << "Harry\nJerry";
   ```

   When the program reaches this line, it prints

   ```
   Harry
   Jerry
   ```

   You also could separate the two names by appending more of the `cout` operator, such as:

   ```
   cout << "Harry" << "\n" << "Jerry";
   ```

   Because the `\n` only takes one byte of storage, you can output it as a character literal by typing `'\n'` in place of the preceding `"\n"`.

2. The following short program rings the bell on your computer by assigning the `\a` escape sequence to a variable, then printing that variable.

   ```
   // Filename: C4BELL.CPP
   // Rings the bell
   #include <iostream.h>
   main()
   {
      char bell='\a';
      cout << bell;    // No newline needed here.
      return 0;
   }
   ```

Constant Variables

The term *constant variable* might seem like a contradiction. After all, a constant never changes and a variable holds values that change. In C++ terminology, you can declare variables to be constants with the `const` keyword. Throughout your program, the constants act like variables; you can use a constant variable anywhere you can use a variable, but you cannot change constant variables. To declare a constant, put the keyword `const` in front of the variable declaration, for instance:

```
const int days_of_week = 7;
```

C++ offers the `const` keyword as an improvement of the `#define` preprocessor directive that C uses. Although C++ supports `#define` as well, `const` enables you to specify constant values with specific data types.

The `const` keyword is appropriate when you have data that does not change. For example, the mathematical π is a good candidate for a constant. If you accidently attempt to store a value in a constant, C++ will let you know. Most C++ programmers choose to type their constant names in uppercase characters to distinguish them from regular variables. This is the one time when uppercase reigns in C++.

NOTE: This book reserves the name *constant* for C++ program constants declared with the `const` keyword. The term *literal* is used for numeric, character, and string data values. Some books choose to use the terms constant and literal interchangeably, but in C++, the difference can be critical.

Example

Suppose a teacher wanted to compute the area of a circle for the class. To do so, the teacher needs the value of π (mathematically, π is approximately 3.14159). Because π remains constant, it is a good candidate for a const keyword, as the following program shows:

Comment for the program filename and description.

Declare a constant value for π.

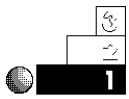
Declare variables for radius and area.

Compute and print the area for both radius values.

```
// Filename: C4AREAC.CPP
// Computes a circle with radius of 5 and 20.
#include <iostream.h>
main()
{
   const float PI=3.14159;
   float radius = 5;
   float area;

   area = radius * radius * PI;  // Circle area calculation
   cout << "The area is " << area << " with a radius of 5.\n";

   radius = 20;      // Compute area with new radius.
   area = radius * radius * PI;
   cout << "The area is " << area << " with a radius of 20.\n";

   return 0;
}
```

Review Questions

The answers to the review questions are in Appendix B.



1. Which of the following variable names are valid?

   `my_name  89_sales  sales_89  a-salary`

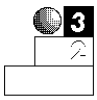
2. Which of the following literals are characters, strings, integers, and floating-point literals?

   `0  -12.0  "2.0"  "X"  'X'  65.4  -708  '0'`

3. How many variables do the following statements declare, and what are their types?

```
int i, j, k;
char c, d, e;
float x=65.43;
```

4. With what do all string literals end?
5. True or false: An unsigned variable can hold a larger value than a signed variable.



6. How many characters of storage does the following literal take?

```
'\x41'
```

7. How is the following string stored at the bit level?

```
"Order 10 of them."
```

8. How is the following string (called a *null string*) stored at the bit level? (*Hint:* The length is zero, but there is still a terminating character.)

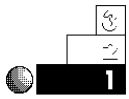
```
""
```

9. What is wrong with the following program?

```
#include <iostream.h>
main()
{
   const int age=35;
   cout << age << "\n";

   age = 52;
   cout << age << "\n";
   return 0;
}
```

Review Exercises

Now that you have learned some basic C++ concepts, the remainder of the book will include this section of review exercises so you can practice your programming skills.



1. Write the C++ code to store three variables: your weight (you can fib), height in feet, and shoe size. Declare the variables, then assign their values in the body of your program.

2. Rewrite your program from Exercise 1, adding proper `cout` statements to print the values to the screen. Use appropriate messages (by printing string literals) to describe the numbers that are printed.

3. Write a program that stores a value and prints each type of variable you learned in this chapter.
4. Write a program that stores a value into every type of variable C++ allows. You must declare each variable at the beginning of your program. Give them values and print them.

Summary

A firm grasp of C++'s fundamentals is critical to a better understanding of the more detailed material that follows. This is one of the last general-topic chapters in the book. You learned about variable types, literal types, how to name variables, how to assign variable values, and how to declare constants. These issues are *critical* to understanding the remaining concepts in C++.

This chapter taught you how to store almost every type of literal into variables. There is no string variable, so you cannot store string literals in string variables (as you can in other programming languages). However, you can "fool" C++ into *thinking* it has a string variable by using a character array to hold strings. You learn this important concept in the next chapter.