WEEK 1

DAY 3

Working with Variables and Constants

Programs need a way to store the data they use or create so it can be used later in the program's execution. Variables and constants offer various ways to represent, store, and manipulate that data.

Today, you will learn

- How to declare and define variables and constants
- How to assign values to variables and manipulate those values
- How to write the value of a variable to the screen

What Is a Variable?

In C++, a *variable* is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

Notice that variables are used for temporary storage. When you exit a program or turn the computer off, the information in variables is lost. Permanent storage is a different matter. Typically, the values from variables are permanently stored either to a database or to a file on disk. Storing to a file on disk is discussed on Day 16, "Advanced Inheritance."

Storing Data in Memory

Your computer's memory can be viewed as a series of cubbyholes. Each cubbyhole is one of many, many such holes all lined up. Each cubbyhole—or memory location—is numbered sequentially. These numbers are known as *memory addresses*. A variable reserves one or more cubbyholes in which you can store a value.

Your variable's name (for example, `myVariable`) is a label on one of these cubbyholes so that you can find it easily without knowing its actual memory address. Figure 3.1 is a schematic representation of this idea. As you can see from the figure, `myVariable` starts at memory address 103. Depending on the size of `myVariable`, it can take up one or more memory addresses.

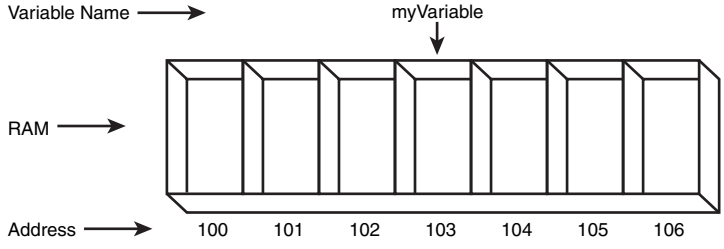


FIGURE 3.1
A schematic representation of memory.

NOTE

RAM stands for random access memory. When you run your program, it is loaded into RAM from the disk file. All variables are also created in RAM. When programmers talk about memory, it is usually RAM to which they are referring.

Setting Aside Memory

When you define a variable in C++, you must tell the compiler what kind of variable it is (this is usually referred to as the variable's "type"): an integer, a floating-point number, a character, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable. It also allows the compiler to warn you or produce an error message if you accidentally attempt to store a value of the

wrong type in your variable (this characteristic of a programming language is called "*strong typing*").

Each cubbyhole is one byte in size. If the type of variable you create is four bytes in size, it needs four bytes of memory, or four cubbyholes. The type of the variable (for example, integer) tells the compiler how much memory (how many cubbyholes) to set aside for the variable.

There was a time when it was imperative that programmers understood bits and bytes; after all, these are the fundamental units of storage. Computer programs have gotten better at abstracting away these details, but it is still helpful to understand how data is stored. For a quick review of the underlying concepts in binary math, please take a look at Appendix A, "Working with Numbers: Binary and Hexadecimal."

NOTE

If mathematics makes you want to run from the room screaming, don't bother with Appendix A; you won't really need it. The truth is that programmers no longer need to be mathematicians; though it is important to be comfortable with logic and rational thinking.

Size of Integers

On any one computer, each variable type takes up a single, unchanging amount of room. That is, an integer might be two bytes on one machine and four on another, but on either computer it is always the same, day in and day out.

Single characters—including letters, numbers, and symbols—are stored in a variable of type `char`. A `char` variable is most often one byte long.

NOTE

There is endless debate about how to pronounce `char`. Some say it as "car," some say it as "char"(coal), others say it as "care." Clearly, car is correct because that is how *I* say it, but feel free to say it however you like.

For smaller integer numbers, a variable can be created using the `short` type. A `short` integer is two bytes on most computers, a `long` integer is usually four bytes, and an integer (without the keyword `short` or `long`) is usually two or four bytes.

You'd think the language would specify the exact size that each of its types should be; however, C++ doesn't. All it says is that a `short` must be less than or equal to the size of an `int`, which, in turn, must be less than or equal to the size of a `long`.

That said, you're probably working on a computer with a two-byte `short` and a four-byte `int`, with a four-byte `long`.

The size of an integer is determined by the processor (16 bit, 32 bit, or 64 bit) and the compiler you use. On a 32-bit computer with an Intel Pentium processor, using modern compilers, integers are *four* bytes.

CAUTION

When creating programs, you should never assume the amount of memory that is being used for any particular type.

Compile and run Listing 3.1 and it will tell you the exact size of each of these types on your computer.

LISTING 3.1 Determining the Size of Variable Types on Your Computer

```
#include <iostream>

int main()
{
   using std::cout;

   cout << "The size of an int is:\t\t"
      << sizeof(int)    << " bytes.\n";
   cout << "The size of a short int is:\t"
      << sizeof(short)  << " bytes.\n";
   cout << "The size of a long int is:\t"
      << sizeof(long)   << " bytes.\n";
   cout << "The size of a char is:\t\t"
      << sizeof(char)   << " bytes.\n";
   cout << "The size of a float is:\t\t"
       << sizeof(float)  << " bytes.\n";
   cout << "The size of a double is:\t"
      << sizeof(double) << " bytes.\n";
   cout << "The size of a bool is:\t"
      << sizeof(bool)   << " bytes.\n";

   return 0;
}
```

OUTPUT

```
The size of an int is:          4 bytes.
The size of a short int is:     2 bytes.
The size of a long int is:      4 bytes.
The size of a char is:          1 bytes.
The size of a float is:         4 bytes.
The size of a double is:        8 bytes.
The size of a bool is:          1 bytes.
```

NOTE On your computer, the number of bytes presented might be different.

Most of Listing 3.1 should be pretty familiar. The lines have been split to make them fit for the book, so for example, lines 7 and 8 could really be on a single line. The compiler ignores whitespace (spaces, tabs, line returns) and so you can treat these as a single line. That's why you need a ";" at the end of most lines.

The new feature in this program to notice is the use of the `sizeof` operator on lines 7–20. The `sizeof` is used like a function. When called, it tells you the size of the item you pass to it as a parameter. On line 8, for example, the keyword `int` is passed to `sizeof`. You'll learn later in today's lesson that `int` is used to describe a standard integer variable. Using `sizeof` on a Pentium 4, Windows XP machine, an `int` is four bytes, which coincidentally also is the size of a `long int` on the same computer.

The other lines of Listing 3.1 show the sizes of other data types. You'll learn about the values these data types can store and the differences between each in a few minutes.

`signed` and `unsigned`

All integer types come in two varieties: `signed` and `unsigned`. Sometimes, you need negative numbers, and sometimes you don't. Any integer without the word "unsigned" is assumed to be `signed`. `signed` integers can be negative or positive. `unsigned` integers are always positive.

Integers, whether `signed` or `unsigned` are stored in the same amount of space. Because of this, part of the storage room for a `signed` integer must be used to hold information on whether the number is negative or positive. The result is that the largest number you can store in an `unsigned` integer is twice as big as the largest positive number you can store in a `signed` integer.

For example, if a `short` integer is stored in two bytes, then an `unsigned short` integer can handle numbers from 0 to 65,535. Alternatively, for a `signed short`, half the numbers that can be stored are negative; thus, a `signed short` can only represent positive numbers up to 32,767. The `signed short` can also, however, represent negative numbers giving it a total range from –32,768 to 32,767.

For more information on the precedence of operators, read Appendix C, "Operator Precedence."

Fundamental Variable Types

Several variable types are built in to C++. They can be conveniently divided into integer variables (the type discussed so far), floating-point variables, and character variables.

Floating-point variables have values that can be expressed as fractions—that is, they are real numbers. Character variables hold a single byte and are generally used for holding the 256 characters and symbols of the ASCII and extended ASCII character sets.

NOTE

The ASCII character set is the set of characters standardized for use on computers. ASCII is an acronym for American Standard Code for Information Interchange. Nearly every computer operating system supports ASCII, although many support other international character sets as well.

The types of variables used in C++ programs are described in Table 3.1. This table shows the variable type, how much room the type generally takes in memory, and what kinds of values can be stored in these variables. The values that can be stored are determined by the size of the variable types, so check your output from Listing 3.1 to see if your variable types are the same size. It is most likely that they are the same size unless you are using a computer with a 64-bit processor.

TABLE 3.1 Variable Types

Type	*Size*	*Values*
`bool`	1 byte	true or false
`unsigned short int`	2 bytes	0 to 65,535
`short int`	2 bytes	–32,768 to 32,767
`unsigned long int`	4 bytes	0 to 4,294,967,295
`long int`	4 bytes	–2,147,483,648 to 2,147,483,647
`int` (16 bit)	2 bytes	–32,768 to 32,767
`int` (32 bit)	4 bytes	–2,147,483,648 to 2,147,483,647
`unsigned int` (16 bit)	2 bytes	0 to 65,535
`unsigned int` (32 bit)	4 bytes	0 to 4,294,967,295
`char`	1 byte	256 character values
`float`	4 bytes	1.2e–38 to 3.4e38
`double`	8 bytes	2.2e–308 to 1.8e308

NOTE

The sizes of variables might be different from those shown in Table 3.1, depending on the compiler and the computer you are using. If your computer had the same output as was presented in Listing 3.1, Table 3.1 should

apply to your compiler. If your output from Listing 3.1 was different, you should consult your compiler's manual for the values that your variable types can hold.

Defining a Variable

Up to this point, you have seen a number of variables created and used. Now, it is time to learn how to create your own.

You create or *define* a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but it cannot contain spaces. Legal variable names include `x`, `J23qrsnf`, and `myAge`. Good variable names tell you what the variables are for; using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called `myAge`:


```
int myAge;
```

NOTE

When you declare a variable, memory is allocated (set aside) for that variable. The *value* of the variable will be whatever happened to be in that memory at that time. You will see in a moment how to assign a new value to that memory.

As a general programming practice, avoid such horrific names as `J23qrsnf`, and restrict single-letter variable names (such as `x` or `i`) to variables that are used only very briefly. Try to use expressive names such as `myAge` or `howMany`. Such names are easier to understand three weeks later when you are scratching your head trying to figure out what you meant when you wrote that line of code.

Try this experiment: Guess what these programs do, based on the first few lines of code:

Example 1

```
int main()
{
     unsigned short x;
     unsigned short y;
     unsigned short z;
     z = x * y;
     return 0;
}
```

Example 2

```
int main()
{
     unsigned short Width;
     unsigned short Length;
     unsigned short Area;
     Area = Width * Length;
     return 0;
}
```

NOTE

If you compile these programs, your compiler will warn that the values are not initialized. You'll see how to solve this problem shortly.

Clearly, the purpose of the second program is easier to guess, and the inconvenience of having to type the longer variable names is more than made up for by how much easier it is to understand, and thus maintain, the second program.

Case Sensitivity

C++ is case sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named `age` is different from `Age`, which is different from `AGE`.

CAUTION

Some compilers allow you to turn case sensitivity off. Don't be tempted to do this; your programs won't work with other compilers, and other C++ programmers will be very confused by your code.

Naming Conventions

Various conventions exist for how to name variables, and although it doesn't much matter which method you adopt, it is important to be consistent throughout your program. Inconsistent naming will confuse other programmers when they read your code.

Many programmers prefer to use all lowercase letters for their variable names. If the name requires two words (for example, my car), two popular conventions are used: `my_car` or `myCar`. The latter form is called camel notation because the capitalization looks something like a camel's hump.

Some people find the underscore character (`my_car`) to be easier to read, but others prefer to avoid the underscore because it is more difficult to type. This book uses camel notation, in which the second and all subsequent words are capitalized: `myCar`, `theQuickBrownFox`, and so forth.

Many advanced programmers employ a notation style referred to as Hungarian notation. The idea behind Hungarian notation is to prefix every variable with a set of characters that describes its type. Integer variables might begin with a lowercase letter i. Variables of type `long` might begin with a lowercase l. Other notations indicate different constructs within C++ that you will learn about later, such as constants, globals, pointers, and so forth.

NOTE

It is called Hungarian notation because the man who invented it, Charles Simonyi of Microsoft, is Hungarian. You can find his original monograph at http://www.strangecreations.com//library/c/naming.txt.

Microsoft has moved away from Hungarian notation recently, and the design recommendations for C# strongly recommend *not* using Hungarian notation. Their reasoning for C# applies equally well to C++.

Keywords

Some words are reserved by C++, and you cannot use them as variable names. These keywords have special meaning to the C++ compiler. Keywords include `if`, `while`, `for`, and `main`. A list of keywords defined by C++ is presented in Table 3.2 as well as in Appendix B, "C++ Keywords." Your compiler might have additional reserved words, so you should check its manual for a complete list.

TABLE 3.2 The C++ Keywords

`asm`	`else`	`new`	`this`
`auto`	`enum`	`operator`	`throw`
`bool`	`explicit`	`private`	`true`
`break`	`export`	`protected`	`try`
`case`	`extern`	`public`	`typedef`
`catch`	`false`	`register`	`typeid`
`char`	`float`	`reinterpret_cast`	`typename`
`class`	`for`	`return`	`union`
`const`	`friend`	`short`	`unsigned`
`const_cast`	`goto`	`signed`	`using`
`continue`	`if`	`sizeof`	`virtual`
`default`	`inline`	`static`	`void`

TABLE 3.2 continued

`delete`	`int`	`static_cast`	`volatile`
`do`	`long`	`struct`	`wchar_t`
`double`	`mutable`	`switch`	`while`
`dynamic_cast`	`namespace`	`template`	

In addition, the following words are reserved:

`And`	`bitor`	`not_eq`	`xor`
`and_eq`	`compl`	`or`	`xor_eq`
`bitand`	`not`	`or_eq`	

DO	DON'T
DO define a variable by writing the type, then the variable name.	**DON'T** use C++ keywords as variable names.
DO use meaningful variable names.	**DON'T** make assumptions about how many bytes are used to store a variable.
DO remember that C++ is case sensitive.	**DON'T** use `unsigned` variables for negative numbers.
DO understand the number of bytes each variable type consumes in memory and what values can be stored in variables of that type.	

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas. For example:

```
unsigned int myAge, myWeight;     // two unsigned int variables
long int area, width, length;     // three long integers
```

As you can see, `myAge` and `myWeight` are each declared as `unsigned` integer variables. The second line declares three individual `long` variables named `area`, `width`, and `length`. The type (`long`) is assigned to all the variables, so you cannot mix types in one definition statement.

Assigning Values to Your Variables

You assign a value to a variable by using the assignment operator (`=`). Thus, you would assign `5` to `width` by writing

```
unsigned short width;
width = 5;
```

NOTE

`long` is a shorthand version of `long int`, and `short` is a shorthand version of `short int`.

You can combine the steps of creating a variable and assigning a value to it. For example, you can combine these two steps for the `width` variable by writing:

```
unsigned short width = 5;
```

This initialization looks very much like the earlier assignment, and when using integer variables like `width`, the difference is minor. Later, when `const` is covered, you will see that some variables must be initialized because they cannot be assigned a value at a later time.

Just as you can define more than one variable at a time, you can initialize more than one variable at creation. For example, the following creates two variables of type `long` and initializes them:

```
long width = 5, length = 7;
```

This example initializes the `long` integer variable `width` to the value `5` and the `long` integer variable `length` to the value `7`. You can even mix definitions and initializations:

```
int myAge = 39, yourAge, hisAge = 40;
```

This example creates three type `int` variables, and it initializes the first (`myAge`) and third (`hisAge`).

Listing 3.2 shows a complete program, ready to compile, that computes the area of a rectangle and writes the answer to the screen.

LISTING 3.2 A Demonstration of the Use of Variables

```
 1:  // Demonstration of variables
 2:  #include <iostream>
 3:
 4:  int main()
 5:  {
 6:     using std::cout;
 7:     using std::endl;
 8:
 9:     unsigned short int Width = 5, Length;
10:     Length = 10;
11:
12:     // create  an unsigned short and initialize with result
13:     // of multiplying Width by Length
14:     unsigned short int Area  = (Width * Length);
```

LISTING 3.2 continued

```
15:
16:     cout << "Width:" << Width << endl;
17:     cout << "Length: "  << Length << endl;
18:     cout << "Area: " << Area << endl;
19:     return 0;
20: }
```

OUTPUT

```
Width:5
Length: 10
Area: 50
```

ANALYSIS As you have seen in the previous listing, line 2 includes the required `include` statement for the `iostream`'s library so that `cout` will work. Line 4 begins the program with the `main()` function. Lines 6 and 7 define `cout` and `endl` as being part of the standard (`std`) namespace.

On line 9, the first variables are defined. `Width` is defined as an `unsigned short` integer, and its value is initialized to 5. Another `unsigned short` integer, `Length`, is also defined, but it is not initialized. On line 10, the value 10 is assigned to `Length`.

On line 14, an `unsigned short` integer, `Area`, is defined, and it is initialized with the value obtained by multiplying `Width` times `Length`. On lines 16–18, the values of the variables are printed to the screen. Note that the special word `endl` creates a new line.

Creating Aliases with `typedef`

It can become tedious, repetitious, and, most important, error-prone to keep writing `unsigned short int`. C++ enables you to create an alias for this phrase by using the keyword `typedef`, which stands for type definition.

In effect, you are creating a synonym, and it is important to distinguish this from creating a new type (which you will do on Day 6, "Understanding Object-Oriented Programming"). `typedef` is used by writing the keyword `typedef`, followed by the existing type, then the new name, and ending with a semicolon. For example,

```
typedef unsigned short int USHORT;
```

creates the new name `USHORT` that you can use anywhere you might have written `unsigned short int`. Listing 3.3 is a replay of Listing 3.2, using the type definition `USHORT` rather than `unsigned short int`.

LISTING 3.3 A Demonstration of *typedef*

```
 1:  // Demonstrates typedef keyword
 2:  #include <iostream>
 3:
 4:  typedef unsigned short int USHORT;   //typedef defined
 5:
 6:  int main()
 7:  {
 8:
 9:     using std::cout;
10:     using std::endl;
11:
12:     USHORT  Width = 5;
13:     USHORT Length;
14:     Length = 10;
15:     USHORT Area  = Width * Length;
16:     cout << "Width:" << Width << endl;
17:     cout << "Length: "  << Length << endl;
18:     cout << "Area: " << Area <<endl;
19:     return 0;
20:  }
```

OUTPUT

```
Width:5
Length: 10
Area: 50
```

NOTE An asterisk (*) indicates multiplication.

ANALYSIS On line 4, `USHORT` is `typedefined` (some programmers say "typedef'ed") as a synonym for `unsigned short int`. The program is very much like Listing 3.2, and the output is the same.

When to Use `short` and When to Use `long`

One source of confusion for new C++ programmers is when to declare a variable to be type `long` and when to declare it to be type `short`. The rule, when understood, is fairly straightforward: If any chance exists that the value you'll want to put into your variable will be too big for its type, use a larger type.

As shown in Table 3.1, `unsigned short` integers, assuming that they are two bytes, can hold a value only up to 65,535. `signed short` integers split their values between

positive and negative numbers, and thus their maximum value is only half that of the `unsigned`.

Although `unsigned long` integers can hold an extremely large number (4,294,967,295), that is still quite finite. If you need a larger number, you'll have to go to `float` or `double`, and then you lose some precision. `Floats` and `doubles` can hold extremely large numbers, but only the first seven or nine digits are significant on most computers. This means that the number is rounded off after that many digits.

Shorter variables use up less memory. These days, memory is cheap and life is short. Feel free to use `int`, which is probably four bytes on your machine.

Wrapping Around an `unsigned` Integer

That `unsigned long` integers have a limit to the values they can hold is only rarely a problem, but what happens if you do run out of room?

When an `unsigned` integer reaches its maximum value, it wraps around and starts over, much as a car odometer might. Listing 3.4 shows what happens if you try to put too large a value into a short integer.

LISTING 3.4 A Demonstration of Putting Too Large a Value in an `unsigned short` Integer

```
 1:  #include <iostream>
 2:  int main()
 3:  {
 4:     using std::cout;
 5:     using std::endl;
 6:
 7:     unsigned short int smallNumber;
 8:     smallNumber = 65535;
 9:     cout << "small number:" << smallNumber << endl;
10:     smallNumber++;
11:     cout << "small number:" << smallNumber << endl;
12:     smallNumber++;
13:     cout << "small number:" << smallNumber << endl;
14:     return 0;
15:  }
```

OUTPUT

```
small number:65535
small number:0
small number:1
```

ANALYSIS On line 7, `smallNumber` is declared to be an `unsigned short int`, which on a Pentium 4 computer running Windows XP is a two-byte variable, able to hold a value between 0 and 65,535. On line 8, the maximum value is assigned to `smallNumber`, and it is printed on line 9.

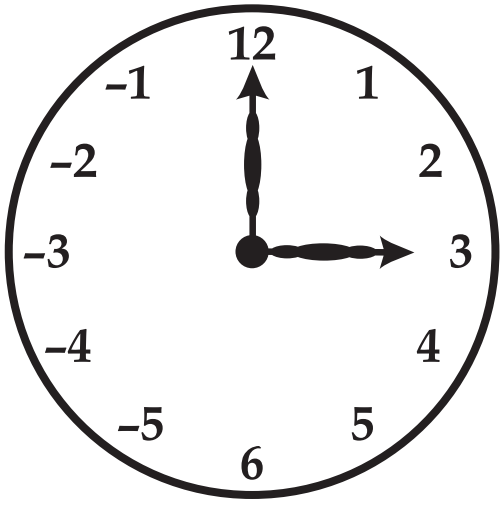
On line 10, `smallNumber` is incremented; that is, 1 is added to it. The symbol for incrementing is ++ (as in the name C++—an incremental increase from C). Thus, the value in `smallNumber` would be 65,536. However, `unsigned short` integers can't hold a number larger than 65,535, so the value is wrapped around to 0, which is printed on line 11.

On line 12 `smallNumber` is incremented again, and then its new value, 1, is printed.

Wrapping Around a `signed` Integer

A `signed` integer is different from an `unsigned` integer, in that half of the values you can represent are negative. Instead of picturing a traditional car odometer, you might picture a clock much like the one shown in Figure 3.2, in which the numbers count upward moving clockwise and downward moving counterclockwise. They cross at the bottom of the clock face (traditional 6 o'clock).

FIGURE 3.2
If clocks used `signed` *numbers.*



One number from 0 is either 1 (clockwise) or –1 (counterclockwise). When you run out of positive numbers, you run right into the largest negative numbers and then count back down to 0. Listing 3.5 shows what happens when you add 1 to the maximum positive number in a `short` integer.

LISTING 3.5 A Demonstration of Adding Too Large a Number to a `signed short` Integer

```
1:  #include <iostream>
2:  int main()
3:  {
4:     short int smallNumber;
```

LISTING 3.5 continued

```
 5:     smallNumber = 32767;
 6:     std::cout << "small number:" << smallNumber << std::endl;
 7:     smallNumber++;
 8:     std::cout << "small number:" << smallNumber << std::endl;
 9:     smallNumber++;
10:     std::cout << "small number:" << smallNumber << std::endl;
11:     return 0;
12: }
```

OUTPUT

```
small number:32767
small number:-32768
small number:-32767
```

ANALYSIS On line 4, `smallNumber` is declared this time to be a `signed short` integer (if you don't explicitly say that it is `unsigned`, an integer variable is assumed to be `signed`). The program proceeds much as the preceding one, but the output is quite different. To fully understand this output, you must be comfortable with how `signed` numbers are represented as bits in a two-byte integer.

The bottom line, however, is that just like an `unsigned` integer, the `signed` integer wraps around from its highest positive value to its highest negative value.

Working with Characters

Character variables (type `char`) are typically 1 byte, enough to hold 256 values (see Appendix C). A `char` can be interpreted as a small number (0–255) or as a member of the ASCII set. The ASCII character set and its ISO equivalent are a way to encode all the letters, numerals, and punctuation marks.

NOTE

> Computers do not know about letters, punctuation, or sentences. All they understand are numbers. In fact, all they really know about is whether a sufficient amount of electricity is at a particular junction of wires. These two states are represented symbolically as a 1 and 0. By grouping ones and zeros, the computer is able to generate patterns that can be interpreted as numbers, and these, in turn, can be assigned to letters and punctuation.

In the ASCII code, the lowercase letter "a" is assigned the value `97`. All the lower- and uppercase letters, all the numerals, and all the punctuation marks are assigned values between 1 and 128. An additional 128 marks and symbols are reserved for use by the

computer maker, although the IBM extended character set has become something of a standard.

NOTE ASCII is usually pronounced "Ask-ee."

Characters and Numbers

When you put a character, for example, "a," into a `char` variable, what really is there is a number between 0 and 255. The compiler knows, however, how to translate back and forth between characters (represented by a single quotation mark and then a letter, numeral, or punctuation mark, followed by a closing single quotation mark) and the corresponding ASCII values.

The value/letter relationship is arbitrary; there is no particular reason that the lowercase "a" is assigned the value `97`. As long as everyone (your keyboard, compiler, and screen) agrees, no problem occurs. It is important to realize, however, that a big difference exists between the value `5` and the character '5'. The character '5' actually has an ASCII value of `53`, much as the letter "a" is valued at `97`. This is illustrated in Listing 3.6.

LISTING 3.6 Printing Characters Based on Numbers

```
1:  #include <iostream>
2:  int main()
3:  {
4:     for (int i = 32; i<128; i++)
5:        std::cout << (char) i;
6:     return 0;
7:  }
```

OUTPUT

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcde-
fghijklmno
pqrstuvwxyz{|}~?
```

ANALYSIS This simple program prints the character values for the integers 32 through 127. This listing uses an integer variable, `i`, on line 4 to accomplish this task. On line 5, the number in the variable `i` is forced to display as a character.

A character variable could also have been used as shown in Listing 3.7, which has the same output.

LISTING 3.7 Printing Characters Based on Numbers, Take 2

```
1:  #include <iostream>
2:  int main()
2:  {
4:     for (unsigned char i = 32; i<128; i++)
5:        std::cout <<  i;
6:     return 0;
7:  }
```

As you can see, an `unsigned` character is used on line 4. Because a character variable is being used instead of a numeric variable, the `cout` on line 5 knows to display the character value.

Special Printing Characters

The C++ compiler recognizes some special characters for formatting. Table 3.3 shows the most common ones. You put these into your code by typing the backslash (called the escape character), followed by the character. Thus, to put a tab character into your code, you enter a single quotation mark, the slash, the letter t, and then a closing single quotation mark:

```
char tabCharacter = '\t';
```

This example declares a `char` variable (`tabCharacter`) and initializes it with the character value `\t`, which is recognized as a tab. The special printing characters are used when printing either to the screen or to a file or other output device.

The escape character (`\`) changes the meaning of the character that follows it. For example, normally the character n means the letter n, but when it is preceded by the escape character, it means new line.

TABLE 3.3 The Escape Characters

Character	*What It Means*
`\a`	Bell (alert)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\'`	Single quote

TABLE 3.3 continued

Character	
`\"`	Double quote
`\?`	Question mark
`\\`	Backslash
`\000`	Octal notation
`\xhhh`	Hexadecimal notation

Constants

Like variables, *constants* are data storage locations. Unlike variables, and as the name implies, constants don't change—they remain constant. You must initialize a constant when you create it, and you cannot assign a new value later.

C++ has two types of constants: literal and symbolic.

Literal Constants

A *literal constant* is a value typed directly into your program wherever it is needed. For example:

```
int myAge = 39;
```

`myAge` is a variable of type `int`; `39` is a literal constant. You can't assign a value to `39`, and its value can't be changed.

Symbolic Constants

A *symbolic constant* is a constant that is represented by a name, just as a variable is represented. Unlike a variable, however, after a constant is initialized, its value can't be changed.

If your program has an integer variable named `students` and another named `classes`, you could compute how many students you have, given a known number of classes, if you knew each class consisted of 15 students:

```
students = classes * 15;
```

In this example, 15 is a literal constant. Your code would be easier to read, and easier to maintain, if you substituted a symbolic constant for this value:

```
students = classes * studentsPerClass
```

If you later decided to change the number of students in each class, you could do so where you define the constant `studentsPerClass` without having to make a change every place you used that value.

Two ways exist to declare a symbolic constant in C++. The old, traditional, and now obsolete way is with a preprocessor directive, `#define`. The second, and appropriate way to create them is using the `const` keyword.

Defining Constants with `#define`

Because a number of existing programs use the preprocessor `#define` directive, it is important for you to understand how it has been used. To define a constant in this obsolete manner, you would enter this:

```
#define studentsPerClass 15
```

Note that `studentsPerClass` is of no particular type (`int`, `char`, and so on). The preprocessor does a simple text substitution. In this case, every time the preprocessor sees the word `studentsPerClass`, it puts in the text `15`.

Because the preprocessor runs before the compiler, your compiler never sees your constant; it sees the number 15.

CAUTION

Although `#define` looks very easy to use, it should be avoided as it has been declared obsolete in the C++ standard.

Defining Constants with `const`

Although `#define` works, a much better way exists to define constants in C++:

```
const unsigned short int studentsPerClass = 15;
```

This example also declares a symbolic constant named `studentsPerClass`, but this time `studentsPerClass` is typed as an `unsigned short int`.

This method of declaring constants has several advantages in making your code easier to maintain and in preventing bugs. The biggest difference is that this constant has a type, and the compiler can enforce that it is used according to its type.

NOTE

Constants cannot be changed while the program is running. If you need to change `studentsPerClass`, for example, you need to change the code and recompile.

Do	Don't
DO watch for numbers overrunning the size of the integer and wrapping around incorrect values. **DO** give your variables meaningful names that reflect their use.	**DON'T** use keywords as variable names. **DON'T** use the `#define` preprocessor directive to declare constants. Use `const`.

Enumerated Constants

Enumerated constants enable you to create new types and then to define variables of those types whose values are restricted to a set of possible values. For example, you could create an enumeration to store colors. Specifically, you could declare `COLOR` to be an enumeration, and then you could define five values for `COLOR`: `RED`, `BLUE`, `GREEN`, `WHITE`, and `BLACK`.

The syntax for creating enumerated constants is to write the keyword `enum`, followed by the new type name, an opening brace, each of the legal values separated by a comma, and finally, a closing brace and a semicolon. Here's an example:

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

This statement performs two tasks:

1. It makes `COLOR` the name of an enumeration; that is, a new type.
2. It makes `RED` a symbolic constant with the value `0`, `BLUE` a symbolic constant with the value `1`, `GREEN` a symbolic constant with the value `2`, and so forth.

Every enumerated constant has an integer value. If you don't specify otherwise, the first constant has the value `0`, and the rest count up from there. Any one of the constants can be initialized with a particular value, however, and those that are not initialized count upward from the ones before them. Thus, if you write

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

then `RED` has the value `100`; `BLUE`, the value `101`; `GREEN`, the value `500`; `WHITE`, the value `501`; and `BLACK`, the value `700`.

You can define variables of type `COLOR`, but they can be assigned only one of the enumerated values (in this case, `RED`, `BLUE`, `GREEN`, `WHITE`, or `BLACK`. You can assign any color value to your `COLOR` variable.

It is important to realize that enumerator variables are generally of type `unsigned int`, and that the enumerated constants equate to integer variables. It is, however, very convenient to be able to name these values when working with information such as colors, days of the week, or similar sets of values. Listing 3.8 presents a program that uses an enumerated type.

LISTING 3.8 A Demonstration of Enumerated Constants

```
 1:  #include <iostream>
 2:  int main()
 3:  {
 4:     enum Days { Sunday, Monday, Tuesday,
 5:                 Wednesday, Thursday, Friday, Saturday };
 6:
 7:     Days today;
 8:     today = Monday;
 9:
10:     if (today == Sunday || today == Saturday)
11:        std::cout << "\nGotta' love the weekends!\n";
12:     else
13:        std::cout << "\nBack to work.\n";
14:
15:     return 0;
16:  }
```

OUTPUT

```
Back to work.
```

ANALYSIS On lines 4 and 5, the enumerated constant `Days` is defined, with seven values. Each of these evaluates to an integer, counting upward from 0; thus, `Monday`'s value is `1` (`Sunday` was `0`).

On line 7, a variable of type `Days` is created—that is, the variable contains a valid value from the list of enumerated constants defined on lines 4 and 5. The value `Monday` is assigned to the variable on line 8. On line 10, a test is done against the value.

The enumerated constant shown on line 8 could be replaced with a series of constant integers, as shown in Listing 3.9.

LISTING 3.9 Same Program Using Constant Integers

```
1:  #include <iostream>
2:  int main()
3:  {
4:     const int Sunday = 0;
5:     const int Monday = 1;
```

LISTING 3.9 continued

```
 6:     const int Tuesday = 2;
 7:     const int Wednesday = 3;
 8:     const int Thursday = 4;
 9:     const int Friday = 5;
10:     const int Saturday = 6;
11:
12:     int today;
13:     today = Monday;
14:
15:     if (today == Sunday || today == Saturday)
16:        std::cout << "\nGotta' love the weekends!\n";
17:     else
18:        std::cout << "\nBack to work.\n";
19:
20:     return 0;
21: }
```

OUTPUT

```
Back to work.
```

CAUTION

A number of the variables you declare in this program are not used. As such, your compiler might give you warnings when you compile this listing.

ANALYSIS The output of this listing is identical to Listing 3.8. Here, each of the constants (`Sunday`, `Monday`, and so on) was explicitly defined, and no enumerated `Days` type exists. Enumerated constants have the advantage of being self-documenting—the intent of the `Days` enumerated type is immediately clear.

Summary

Today's lesson discussed numeric and character variables and constants, which are used by C++ to store data during the execution of your program. Numeric variables are either integral (`char`, `short`, `int`, and `long int`) or they are floating point (`float`, `double`, and `long double`). Numeric variables can also be `signed` or `unsigned`. Although all the types can be of various sizes among different computers, the type specifies an exact size on any given computer.

You must declare a variable before it can be used, and then you must store the type of data that you've declared as correct for that variable. If you put a number that is too large into an integral variable, it wraps around and produces an incorrect result.

Today's lesson also presented literal and symbolic constants as well as enumerated constants. You learned two ways to declare a symbolic constant: using `#define` and using the keyword `const`; however, you learned that using `const` is the appropriate way.

Q&A

Q If a short `int` can run out of room and wrap around, why not always use `long` integers?

A All integer types can run out of room and wrap around, but a `long` integer does so with a much larger number. For example, a two-byte `unsigned short int` wraps around after 65,535, whereas a four-byte `unsigned long int` does not wrap around until 4,294,967,295. However, on most machines, a `long` integer takes up twice as much memory every time you declare one (such as four bytes versus two bytes), and a program with 100 such variables consumes an extra 200 bytes of RAM. Frankly, this is less of a problem than it used to be because most personal computers now come with millions (if not billions) of bytes of memory.

Using larger types than you need might also require additional time for your computer's processor to processes.

Q What happens if I assign a number with a decimal point to an integer rather than to a `float`? Consider the following line of code:

```
int aNumber = 5.4;
```

A A good compiler issues a warning, but the assignment is completely legal. The number you've assigned is truncated into an integer. Thus, if you assign 5.4 to an integer variable, that variable will have the value `5`. Information will be lost, however, and if you then try to assign the value in that integer variable to a `float` variable, the `float` variable will have only `5`.

Q Why not use literal constants; why go to the bother of using symbolic constants?

A If you use a value in many places throughout your program, a symbolic constant allows all the values to change just by changing the one definition of the constant. Symbolic constants also speak for themselves. It might be hard to understand why a number is being multiplied by 360, but it's much easier to understand what's going on if the number is being multiplied by `degreesInACircle`.

Q What happens if I assign a negative number to an `unsigned` variable? Consider the following line of code:

```
unsigned int aPositiveNumber = -1;
```

A A good compiler issues a warning, but the assignment is legal. The negative number is assessed as a bit pattern and is assigned to the variable. The value of that

variable is then interpreted as an `unsigned` number. Thus, –1, whose bit pattern is 11111111 11111111 (0xFF in hex), is assessed as the `unsigned` value `65,535`.

Q Can I work with C++ without understanding bit patterns, binary arithmetic, and hexadecimal?

A Yes, but not as effectively as if you do understand these topics. C++ does not do as good a job as some languages at "protecting" you from what the computer is really doing. This is actually a benefit because it provides you with tremendous power that other languages don't. As with any power tool, however, to get the most out of C++, you must understand how it works. Programmers who try to program in C++ without understanding the fundamentals of the binary system often are confused by their results.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to answer the quiz and exercise questions before checking the answers in Appendix D, and be certain that you understand the answers before continuing to tomorrow's lesson.

Quiz

1. What is the difference between an integer variable and a floating-point variable?
2. What are the differences between an `unsigned short int` and a `long int`?
3. What are the advantages of using a symbolic constant rather than a literal constant?
4. What are the advantages of using the `const` keyword rather than `#define`?
5. What makes for a good or bad variable name?
6. Given this `enum`, what is the value of `BLUE`?

```
enum COLOR { WHITE, BLACK = 100, RED, BLUE, GREEN = 300 };
```

7. Which of the following variable names are good, which are bad, and which are invalid?
   a. `Age`
   b. `!ex`
   c. `R79J`
   d. `TotalIncome`
   e. `__Invalid`

Exercises

1. What would be the correct variable type in which to store the following information?
    a. Your age
    b. The area of your backyard
    c. The number of stars in the galaxy
    d. The average rainfall for the month of January
2. Create good variable names for this information.
3. Declare a constant for pi as 3.14159.
4. Declare a `float` variable and initialize it using your pi constant.