```
int main()                          //function name "main"
   {                                //start function body
   cout << "Every age has a language of its own\n"; //statement
   return 0;                        //statement
   }                                //end function body
```

Comments start with a double slash symbol (//) and terminate at the end of the line. (This is one of the exceptions to the rule that the compiler ignores whitespace.) A comment can start at the beginning of the line or on the same line following a program statement. Both possibilities are shown in the COMMENTS example.

When to Use Comments

Comments are almost always a good thing. Most programmers don't use enough of them. If you're tempted to leave out comments, remember that not everyone is as smart as you; they may need more explanation than you do about what your program is doing. Also, you may not be as smart next month, when you've forgotten key details of your program's operation, as you are today.

Use comments to explain to the person looking at the listing what you're trying to do. The details are in the program statements themselves, so the comments should concentrate on the big picture, clarifying your reasons for using a certain statement or group of statements.

Alternative Comment Syntax

There's a second comment style available in C++:

```
/* this is an old-style comment */
```

This type of comment (the only comment originally available in C) begins with the /* character pair and ends with */ (not with the end of the line). These symbols are harder to type (since / is lowercase while * is uppercase) and take up more space on the line, so this style is not generally used in C++. However, it has advantages in special situations. You can write a multiline comment with only two comment symbols:

```
/* this
is a
potentially
very long
multiline
comment
*/
```

This is a good approach to making a comment out of a large text passage, since it saves inserting the // symbol on every line.

You can also insert a /* */ comment anywhere within the text of a program line:

```
func1()
   {  /* empty function body */  }
```

If you attempt to use the // style comment in this case, the closing brace won't be visible to the compiler—since a // style comment runs to the end of the line—and the code won't compile correctly.

Integer Variables

Variables are the most fundamental part of any language. A variable has a symbolic name and can be given a variety of values. Variables are located in particular places in the computer's memory. When a variable is given a value, that value is actually placed in the memory space assigned to the variable. Most popular languages use the same general variable types, such as integers, floating-point numbers, and characters, so you are probably already familiar with the ideas behind them.

Integer variables represent integer numbers like 1, 30,000, and –27. Such numbers are used for counting discrete numbers of objects, like 11 pencils or 99 bottles of beer. Unlike floating-point numbers, integers have no fractional part; you can express the idea of four using integers, but not four and one-half.

Defining Integer Variables

Integer variables exist in several sizes, but the most commonly used is type int. The amount of memory occupied by the integer types is system dependent. On a 32-bit system such as Windows, an int occupies 4 bytes (which is 32 bits) of memory. This allows an int to hold numbers in the range from –2,147,483,648 to 2,147,483,647. Figure 2.3 shows an integer variable in memory.

While type int occupies 4 bytes on current Windows computers, it occupied only 2 bytes in MS-DOS and earlier versions of Windows. The ranges occupied by the various types are listed in the header file LIMITS; you can also look them up using your compiler's help system.

Here's a program that defines and uses several variables of type int:

```
// intvars.cpp
// demonstrates integer variables
#include <iostream>
using namespace std;

int main()
   {
   int var1;               //define var1
   int var2;               //define var2
```

```
var1 = 20;              //assign value to var1
var2 = var1 + 10;     //assign value to var2
cout << "var1+10 is ";  //output text
cout << var2 << endl; //output value of var2
return 0;
}
```

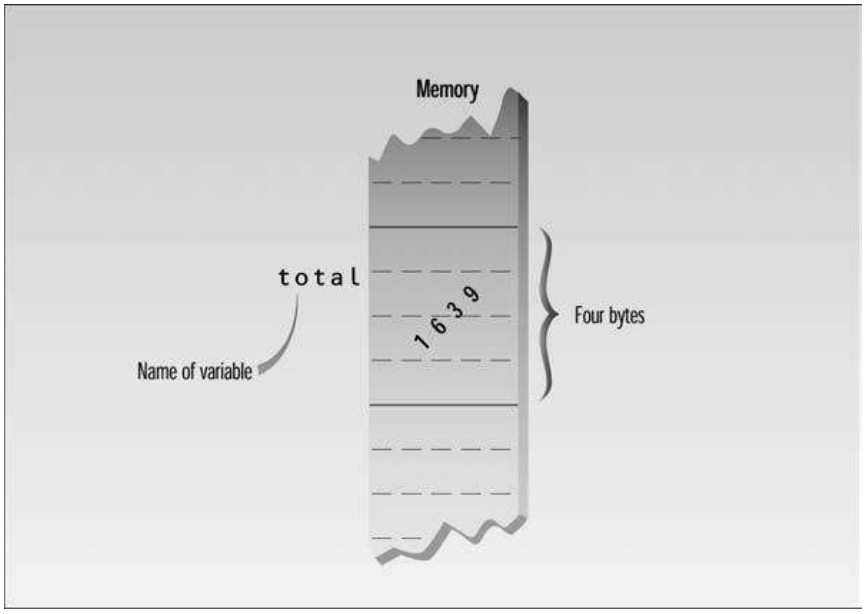

FIGURE 2.3
Variable of type `int` in memory.

Type this program into your compiler's edit screen (or load it from the Web site), compile and link it, and then run it. Examine the output window. The statements

```
int var1;
int var2;
```

define two integer variables, `var1` and `var2`. The keyword `int` signals the type of variable. These statements, which are called declarations, must terminate with a semicolon, like other program statements.

You must declare a variable before using it. However, you can place variable declarations anywhere in a program. It's not necessary to declare variables before the first executable statement (as was necessary in C). However, it's probably more readable if commonly-used variables are located at the beginning of the program.

Declarations and Definitions

Let's digress for a moment to note a subtle distinction between the terms definition and declaration as applied to variables.

A declaration introduces a variable's name (such as `var1`) into a program and specifies its type (such as `int`). However, if a declaration also sets aside memory for the variable, it is also called a definition. The statements

```
int var1;
int var2;
```

in the INTVARS program are definitions, as well as declarations, because they set aside memory for `var1` and `var2`. We'll be concerned mostly with declarations that are also definitions, but later on we'll see various kinds of declarations that are not definitions.

Variable Names

The program INTVARS uses variables named `var1` and `var2`. The names given to variables (and other program features) are called identifiers. What are the rules for writing identifiers? You can use upper- and lowercase letters, and the digits from 1 to 9. You can also use the underscore (_). The first character must be a letter or underscore. Identifiers can be as long as you like, but most compilers will only recognize the first few hundred characters. The compiler distinguishes between upper- and lowercase letters, so `Var` is not the same as `var` or `VAR`.

You can't use a C++ keyword as a variable name. A keyword is a predefined word with a special meaning. `int`, `class`, `if`, and `while` are examples of keywords. A complete list of keywords can be found in Appendix B, "C++ Precedence Table and Keywords," and in your compiler's documentation.

Many C++ programmers follow the convention of using all lowercase letters for variable names. Other programmers use a mixture of upper- and lowercase, as in `IntVar` or `dataCount`. Still others make liberal use of underscores. Whichever approach you use, it's good to be consistent throughout a program. Names in all uppercase are sometimes reserved for constants (see the discussion of `const` that follows). These same conventions apply to naming other program elements such as classes and functions.

A variable's name should make clear to anyone reading the listing the variable's purpose and how it is used. Thus `boilerTemperature` is better than something cryptic like `bT` or `t`.

Assignment Statements

The statements

```
var1 = 20;
var2 = var1 + 10;
```

assign values to the two variables. The equal sign (=), as you might guess, causes the value on the right to be assigned to the variable on the left. The = in C++ is equivalent to the := in Pascal or the = in BASIC. In the first line shown here, var1, which previously had no value, is given the value 20.

Integer Constants

The number 20 is an integer constant. Constants don't change during the course of the program. An integer constant consists of numerical digits. There must be no decimal point in an integer constant, and it must lie within the range of integers.

In the second program line shown here, the plus sign (+) adds the value of var1 and 10, in which 10 is another constant. The result of this addition is then assigned to var2.

Output Variations

The statement

```
cout << "var1+10 is ";
```

displays a string constant, as we've seen before. The next statement

```
cout << var2 << endl;
```

displays the value of the variable var2. As you can see in your console output window, the output of the program is

```
var1+10 is 30
```

Note that cout and the << operator know how to treat an integer and a string differently. If we send them a string, they print it as text. If we send them an integer, they print it as a number. This may seem obvious, but it is another example of operator overloading, a key feature of C++. (C programmers will remember that such functions as printf() need to be told not only the variable to be displayed, but the type of the variable as well, which makes the syntax far less intuitive.)

As you can see, the output of the two cout statements appears on the same line on the output screen. No linefeed is inserted automatically. If you want to start on a new line, you must insert a linefeed yourself. We've seen how to do this with the '\n' escape sequence. Now we'll see another way: using something called a manipulator.

The endl Manipulator

The last cout statement in the INTVARS program ends with an unfamiliar word: endl. This causes a linefeed to be inserted into the stream, so that subsequent text is displayed on the next line. It has the same effect as sending the '\n' character, but is somewhat clearer. It's an

example of a manipulator. Manipulators are instructions to the output stream that modify the output in various ways; we'll see more of them as we go along. Strictly speaking, `endl` (unlike `'\n'`) also causes the output buffer to be flushed, but this happens invisibly so for most purposes the two are equivalent.

Other Integer Types

There are several numerical integer types besides type `int`. The two most common types are `long` and `short`. (Strictly speaking type `char` is an integer type as well, but we'll cover it separately.) We noted that the size of type `int` is system dependent. In contrast, types `long` and `short` have fixed sizes no matter what system is used.

Type `long` always occupies four bytes, which is the same as type `int` on 32-bit Windows systems. Thus it has the same range, from –2,147,483,648 to 2,147,483,647. It can also be written as `long int`; this means the same as `long`. There's little point in using type `long` on 32-bit systems, since it's the same as `int`. However, if your program may need to run on a 16-bit system such as MS-DOS, or on older versions of Windows, specifying type `long` will guarantee a four-bit integer type. In 16-bit systems, type `int` has the same range as type `short`.

On all systems type `short` occupies two bytes, giving it a range of –32,768 to 32,767. There's probably not much point using type `short` on modern Windows systems unless it's important to save memory. Type `int`, although twice as large, is accessed faster than type `short`.

If you want to create a constant of type `long`, use the letter `L` following the numerical value, as in

```
longvar = 7678L; // assigns long constant 7678 to longvar
```

Many compilers offer integer types that explicitly specify the number of bits used. (Remember there are 8 bits to a byte.) These type names are preceded by two underscores. They are `__int8`, `int16`, `int32`, and `int64`. The `int8` type corresponds to `char`, and (at least in 32-bit systems) the type name `int16` corresponds to `short` and `int32` corresponds to both `int` and `long`. The `int64` type holds huge integers with up to 19 decimal digits. Using these type names has the advantage that the number of bytes used for a variable is not implementa- tion dependent. However, this is not usually an issue, and these types are seldom used.

Character Variables

Type `char` stores integers that range in value from –128 to 127. Variables of this type occupy only 1 byte (eight bits) of memory. Character variables are sometimes used to store numbers that confine themselves to this limited range, but they are much more commonly used to store ASCII characters.

As you may already know, the ASCII character set is a way of representing characters such as `'a'`, `'B'`, `'$'`, `'3'`, and so on, as numbers. These numbers range from 0 to 127. Most Windows systems extend this range to 255 to accommodate various foreign-language and graphics characters. Appendix A, "ASCII Table," shows the ASCII character set.

Complexities arise when foreign languages are used, and even when programs are transferred between computer systems in the same language. This is because the characters in the range 128 to 255 aren't standardized and because the one-byte size of type `char` is too small to accommodate the number of characters in many languages, such as Japanese. Standard C++ provides a larger character type called `wchar_t` to handle foreign languages. This is important if you're writing programs for international distribution. However, in this book we'll ignore type `wchar_t` and assume that we're dealing with the ASCII character set found in current versions of Windows.

Character Constants

Character constants use single quotation marks around a character, like `'a'` and `'b'`. (Note that this differs from string constants, which use double quotation marks.) When the C++ compiler encounters such a character constant, it translates it into the corresponding ASCII code. The constant `'a'` appearing in a program, for example, will be translated into 97, as shown in Figure 2.4.

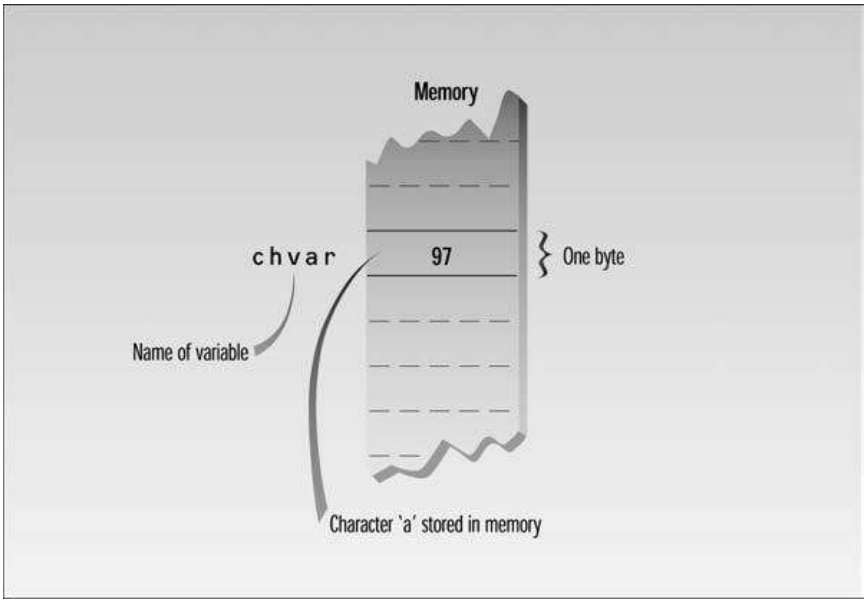

FIGURE 2.4
Variable of type `char` in memory.

Character variables can be assigned character constants as values. The following program shows some examples of character constants and variables.

```
// charvars.cpp
// demonstrates character variables
#include <iostream>        //for cout, etc.
using namespace std;

int main()
   {
   char charvar1 = 'A';    //define char variable as character
   char charvar2 = '\t';   //define char variable as tab

   cout << charvar1;       //display character
   cout << charvar2;       //display character
   charvar1 = 'B';         //set char variable to char constant
   cout << charvar1;       //display character
   cout << '\n';           //display newline character
   return 0;
   }
```

Initialization

Variables can be initialized at the same time they are defined. In this program two variables of type `char`—`charvar1` and `charvar2`—are initialized to the character constants `'A'` and `'\t'`.

Escape Sequences

This second character constant, `'\t'`, is an odd one. Like `'\n'`, which we encountered earlier, it's an example of an escape sequence. The name reflects the fact that the backslash causes an "escape" from the normal way characters are interpreted. In this case the `t` is interpreted not as the character 't' but as the tab character. A tab causes printing to continue at the next tab stop. In console-mode programs, tab stops are positioned every eight spaces. Another character constant, `'\n'`, is sent directly to `cout` in the last line of the program.

Escape sequences can be used as separate characters or embedded in string constants. Table 2.1 shows a list of common escape sequences.

TABLE 2.1 Common Escape Sequences

Escape Sequence	Character
`\ a`	Bell (beep)
`\ b`	Backspace
`\ f`	Formfeed

TABLE 2.1 Continued

Escape Sequence	Character
\ n	Newline
\ r	Return
\ t	Tab
\ \	Backslash
\ ‘	Single quotation mark
\ “	Double quotation marks
\ xdd	Hexadecimal notation

Since the backslash, the single quotation marks, and the double quotation marks all have specialized meanings when used in constants, they must be represented by escape sequences when we want to display them as characters. Here’s an example of a quoted phrase in a string constant:

```
cout << “\”Run, Spot, run,\” she said.”;
```

This translates to

```
“Run, Spot, run,” she said.
```

Sometimes you need to represent a character constant that doesn’t appear on the keyboard, such as the graphics characters above ASCII code 127. To do this, you can use the `‘\xdd’` representation, where each `d` stands for a hexadecimal digit. If you want to print a solid rectangle, for example, you’ll find such a character listed as decimal number 178, which is hexadecimal number `B2` in the ASCII table. This character would be represented by the character constant `‘\xB2’`. We’ll see some examples of this later.

The CHARVARS program prints the value of `charvar1` (`‘A’`) and the value of `charvar2` (a tab). It then sets `charvar1` to a new value (`‘B’`), prints that, and finally prints the newline. The output looks like this:

```
A       B
```

Input with `cin`

Now that we’ve seen some variable types in use, let’s see how a program accomplishes input. The next example program asks the user for a temperature in degrees Fahrenheit, converts it to Celsius, and displays the result. It uses integer variables.

```
// fahren.cpp
// demonstrates cin, newline
#include <iostream>
using namespace std;

int main()
   {
   int ftemp;  //for temperature in fahrenheit

   cout << "Enter temperature in fahrenheit: ";
   cin >> ftemp;
   int ctemp = (ftemp-32) * 5 / 9;
   cout << "Equivalent in Celsius is: " << ctemp << '\n';
   return 0;
   }
```

The statement

```
cin >> ftemp;
```

causes the program to wait for the user to type in a number. The resulting number is placed in the variable `ftemp`. The keyword `cin` (pronounced "C in") is an object, predefined in C++ to correspond to the standard input stream. This stream represents data coming from the keyboard (unless it has been redirected). The `>>` is the extraction or get from operator. It takes the value from the stream object on its left and places it in the variable on its right.

Here's some sample interaction with the program:

```
Enter temperature in fahrenheit: 212
Equivalent in Celsius is: 100
```

Figure 2.5 shows input using `cin` and the extraction operator `>>`.

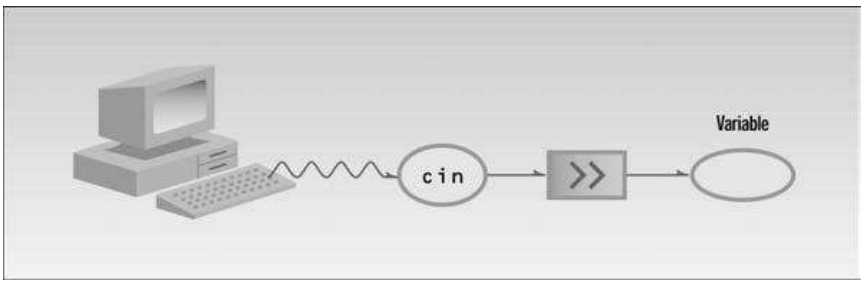

FIGURE 2.5
Input with `cin`.

Variables Defined at Point of Use

The FAHREN program has several new wrinkles besides its input capability. Look closely at the listing. Where is the variable `ctemp` defined? Not at the beginning of the program, but in the next-to-the-last line, where it's used to store the result of the arithmetic operation. As we noted earlier, you can define variables throughout a program, not just at the beginning. (Many languages, including C, require all variables to be defined before the first executable statement.)

Defining variables where they are used can make the listing easier to understand, since you don't need to refer repeatedly to the start of the listing to find the variable definitions. However, the practice should be used with discretion. Variables that are used in many places in a function are better defined at the start of the function.

Cascading <<

The insertion operator `<<` is used repeatedly in the second `cout` statement in FAHREN. This is perfectly legal. The program first sends the phrase Equivalent in Celsius is: to `cout`, then it sends the value of `ctemp`, and finally the newline character `'\n'`.

The extraction operator `>>` can be cascaded with `cin` in the same way, allowing the user to enter a series of values. However, this capability is not used so often, since it eliminates the opportunity to prompt the user between inputs.

Expressions

Any arrangement of variables, constants, and operators that specifies a computation is called an expression. Thus, `alpha+12` and `(alpha-37)*beta/2` are expressions. When the computations specified in the expression are performed, the result is usually a value. Thus if `alpha` is 7, the first expression shown has the value 19.

Parts of expressions may also be expressions. In the second example, `alpha-37` and `beta/2` are expressions. Even single variables and constants, like `alpha` and `37`, are considered to be expressions.

Note that expressions aren't the same as statements. Statements tell the compiler to do something and terminate with a semicolon, while expressions specify a computation. There can be several expressions in a statement.

Precedence

Note the parentheses in the expression

```
(ftemp-32) * 5 / 9
```