CHAPTER

C++ Basics

C++ Basics

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with.

Ada Augusta, Countess of Lovelace

INTRODUCTION

This chapter introduces the C++ language and gives enough detail to allow you to handle simple programs involving expressions, assignments, and console input/output (I/O). The details of assignments and expressions are similar to those of most other high-level languages. Every language has its own console I/O syntax, so if you are not familiar with C++, that may look new and different to you.

1.1 Introduction to C++

Language is the only instrument of science.

Samuel Johnson

This section gives an overview of the C++ programming language.

ORIGINS OF THE C++ LANGUAGE

The C++ programming language can be thought of as the C programming language with classes (and other modern features added). The C programming language was developed by Dennis Ritchie of AT&T Bell Laboratories in the 1970s. It was first used for writing and maintaining the UNIX operating system. (Up until that time, UNIX systems programs were written either in assembly language or in a language called B, a language developed by Ken Thompson, the originator of UNIX.) C is a general-purpose language that can be used for writing any sort of program, but its success and popularity are closely tied to the UNIX operating system. If you wanted to maintain your UNIX system, you needed to use C. C and UNIX fit together so well that soon not just systems programs but almost all commercial programs that ran under UNIX were written in the C language. C became so popular that versions of the language were written for other popular operating systems; its use is thus not limited to computers that use UNIX. However, despite its popularity, C was not without its shortcomings.

The C language is peculiar because it is a high-level language with many of the features of a low-level language. C is somewhere in between the two extremes of a very high-level language and a low-level language, and therein lies both its strengths and its weaknesses. Like (low-level) assembly language, C language programs can directly manipulate the computer's memory. On the other hand, C has the features of a high-level language, which makes it easier to read and write than assembly language. This makes C an excellent choice for writing systems programs, but for other programs (and in some sense even for systems programs) C is not as easy to understand as other languages; also, it does not have as many automatic checks as some other high-level languages.

To overcome these and other shortcomings of C, Bjarne Stroustrup of AT&T Bell Laboratories developed C++ in the early 1980s. Stroustrup designed C++ to be a better C. Most of C is a subset of C++, and so most C programs are also C++ programs. (The reverse is not true; many C++ programs are definitely not C programs.) Unlike C, C++ has facilities for classes and so can be used for object-oriented programming.

C++ AND OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a currently popular and powerful programming technique. The main characteristics of OOP are encapsulation, inheritance, and polymorphism. Encapsulation is a form of information hiding or abstraction. Inheritance has to do with writing reusable code. Polymorphism refers to a way that a single name can have multiple meanings in the context of inheritance. Having made those statements, we must admit that they will hold little meaning for readers who have not heard of OOP before. However, we will describe all these terms in detail later in this book. C++ accommodates OOP by providing classes, a kind of data type combining both data and algorithms. C++ is not what some authorities would call a "pure OOP language." C++ tempers its OOP features with concerns for efficiency and what some might call "practicality." This combination has made C++ currently the most widely used OOP language, although not all of its usage strictly follows the OOP philosophy.

THE CHARACTER OF C++

C++ has classes that allow it to be used as an object-oriented language. It allows for overloading of functions and operators. (All these terms will be explained eventually, so do not be concerned if you do not fully understand some terms.) C++'s connection to the C language gives it a more traditional look than newer object-oriented languages, yet it has more powerful abstraction mechanisms than many other currently popular languages. C++ has a template facility that allows for full and direct implementation of algorithm abstraction. C++ templates allow you to code using parameters for types. The newest C++ standard, and most C++ compilers, allow multiple namespaces to accommodate more reuse of class and function names. The exception handling facilities in C++ are similar to what you would find in other programming languages. Memory management in C++ is similar to that in C. The programmer must allocate his or her own memory and handle his or her own garbage

collection. Most compilers will allow you to do C-style memory management in C++, since C is essentially a subset of C++. However, C++ also has its own syntax for a C++ style of memory management, and you are advised to use the C++ style of memory management when coding in C++. This book uses only the C++ style of memory management.

C++ TERMINOLOGY

functions

All procedure-like entities are called **functions** in C++. Things that are called *procedures*, *methods*, *functions*, or *subprograms* in other languages are all called *functions* in C++. As we will see in the next subsection, a C++ **program** is basically just a function called `main`; when you run a program, the run-time system automatically invokes the function named `main`. Other C++ terminology is pretty much the same as most other programming languages, and in any case, will be explained when each concept is introduced.

program

A SAMPLE C++ PROGRAM

Display 1.1 contains a simple C++ program and two possible screen displays that might be generated when a user runs the program. A C++ program is really a function definition for a function named `main`. When the program is run, the function named `main` is invoked. The body of the function `main` is enclosed in braces, `{}`. When the program is run, the statements in the braces are executed.

The following two lines set up things so that the libraries with console input and output facilities are available to the program. The details concerning these two lines and related topics are covered in Section 1.3 and in Chapters 9, 11, and 12.

```
#include <iostream>
using namespace std;
```

`int main()`

The following line says that `main` is a function with no parameters that returns an `int` (integer) value:

```
int main( )
```

Some compilers will allow you to omit the `int` or replace it with `void`, which indicates a function that does not return a value. However, the above form is the most universally accepted way to start the `main` function of a C++ program.

`return 0;`

The program ends when the following statement is executed:

```
return 0;
```

This statement ends the invocation of the function `main` and returns 0 as the function's value. According to the ANSI/ISO C++ standard, this statement is not required, but many compilers still require it. Chapter 3 covers all these details about C++ functions.

Display 1.1 A Sample C++ Program


```
1  #include <iostream>
2  using namespace std;

3  int main( )
4  {
5      int numberOfLanguages;

6      cout << "Hello reader.\n"
7           << "Welcome to C++.\n";

8      cout << "How many programming languages have you used? ";
9      cin >> numberOfLanguages;

10     if (numberOfLanguages < 1)
11         cout << "Read the preface. You may prefer\n"
12              << "a more elementary book by the same author.\n";
13     else
14         cout << "Enjoy the book.\n";

15     return 0;
16 }
```

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to C++.
How many programming languages have you used? 0
Read the preface. You may prefer
a more elementary book by the same author.
```

User types in `0` on the keyboard.

SAMPLE DIALOGUE 2

```
Hello reader.
Welcome to C++.
How many programming languages have you used? 1
Enjoy the book
```

User types in `1` on the keyboard.

Variable declarations in C++ are similar to what they are in other programming languages. The following line from Display 1.1 declares the variable `numberOfLanguages`:

```
int numberOfLanguages;
```

The type `int` is one of the C++ types for whole numbers (integers).

If you have not programmed in C++ before, then the use of `cin` and `cout` for console I/O is likely to be new to you. That topic is covered a little later in this chapter, but the general idea can be observed in this sample program. For example, consider the following two lines from Display 1.1:

```
cout << "How many programming languages have you used? ";
cin >> numberOfLanguages;
```

The first line outputs the text within the quotation marks to the screen. The second line reads in a number that the user enters at the keyboard and sets the value of the variable `numberOfLanguages` to this number.

The lines

```
cout << "Read the preface. You may prefer\n"
     << "a more elementary book by the same author.\n";
```

output two strings instead of just one string. The details are explained in Section 1.3 later in this chapter, but this brief introduction will be enough to allow you to understand the simple use of `cin` and `cout` in the examples that precede Section 1.3. The symbolism `\n` is the newline character, which instructs the computer to start a new line of output.

Although you may not yet be certain of the exact details of how to write such statements, you can probably guess the meaning of the `if-else` statement. The details will be explained in the next chapter.

(By the way, if you have not had at least some experience with some programming languages, you should read the preface to see if you might not prefer the more elementary book discussed in this program. You need not have had any experience with C++ to read this book, but some minimal programming experience is strongly suggested.)

1.2 Variables, Expressions, and Assignment Statements

> *Once a person has understood the way variables are used in programming, he has understood the quintessence of programming.*
>
> E. W. Dijkstra, *Notes on Structured Programming*

Variables, expressions, and assignments in C++ are similar to those in most other general-purpose languages.

IDENTIFIERS

identifier

The name of a variable (or other item you might define in a program) is called an **identifier**. A C++ identifier must start with either a letter or the underscore symbol,

and all the rest of the characters must be letters, digits, or the underscore symbol. For example, the following are all valid identifiers:

```
x x1 x_1 _abc ABC123z7 sum RATE count data2 bigBonus
```

All the above names are legal and would be accepted by the compiler, but the first five are poor choices for identifiers because they are not descriptive of the identifier's use. None of the following are legal identifiers, and all would be rejected by the compiler:

```
12  3X  %change  data-1  myfirst.c  PROG.CPP
```

The first three are not allowed because they do not start with a letter or an underscore. The remaining three are not identifiers because they contain symbols other than letters, digits, and the underscore symbol.

Although it is legal to start an identifier with an underscore, you should avoid doing so, because identifiers starting with an underscore are informally reserved for system identifiers and standard libraries.

C++ is a **case-sensitive** language; that is, it distinguishes between uppercase and lowercase letters in the spelling of identifiers. Hence, the following are three distinct identifiers and could be used to name three distinct variables:

case-sensitive

```
rate  RATE  Rate
```

However, it is not a good idea to use two such variants in the same program, since that might be confusing. Although it is not required by C++, variables are usually spelled with their first letter in lowercase. The predefined identifiers, such as `main`, `cin`, `cout`, and so forth, must be spelled in all lowercase letters. The convention that is now becoming universal in object-oriented programming is to spell variable names with a mix of upper- and lowercase letters (and digits), to always start a variable name with a lowercase letter, and to indicate "word" boundaries with an uppercase letter, as illustrated by the following variable names:

```
topSpeed, bankRate1, bankRate2, timeOfArrival
```

This convention is not as common in C++ as in some other object-oriented languages, but is becoming more widely used and is a good convention to follow.

A C++ identifier can be of any length, although some compilers will ignore all characters after some (large) specified number of initial characters.

Identifiers

A C++ identifier must start with either a letter or the underscore symbol, and the remaining characters must all be letters, digits, or the underscore symbol. C++ identifiers are case sensitive and have no limit to their length.

keyword or reserved word

There is a special class of identifiers, called **keywords** or **reserved words**, that have a predefined meaning in C++ and cannot be used as names for variables or anything else. In the code displays of this book keywords are shown in a different color. A complete list of keywords is given in Appendix 1.

Some predefined words, such as `cin` and `cout`, are not keywords. These predefined words are not part of the core C++ language, and you are allowed to redefine them. Although these predefined words are not keywords, they are defined in libraries required by the C++ language standard. Needless to say, using a predefined identifier for anything other than its standard meaning can be confusing and dangerous and thus should be avoided. The safest and easiest practice is to treat all predefined identifiers as if they were keywords.

VARIABLES

declare

Every variable in a C++ program must be *declared* before it is used. When you **declare** a variable you are telling the compiler—and, ultimately, the computer—what kind of data you will be storing in the variable. For example, the following are two definitions that might occur in a C++ program:

```
int numberOfBeans;
double oneWeight, totalWeight;
```

The first defines the variable `numberOfBeans` so that it can hold a value of type `int`, that is, a whole number. The name `int` is an abbreviation for "integer." The type `int` is one of the types for whole numbers. The second definition declares `oneWeight` and `totalWeight` to be variables of type `double`, which is one of the types for numbers with a decimal point (known as **floating-point numbers**). As illustrated here, when there is more than one variable in a definition, the variables are separated by commas. Also, note that each definition ends with a semicolon.

floating-point number

Every variable must be declared before it is used; otherwise, variables may be declared any place. Of course, they should always be declared in a location that makes the program easier to read. Typically, variables are declared either just before they are used or at the start of a block (indicated by an opening brace, `{`). Any legal identifier, other than a reserved word, may be used for a variable name.[1]

C++ has basic types for characters, integers, and floating-point numbers (numbers with a decimal point). Display 1.2 lists the basic C++ types. The commonly used type for integers is `int`. The type `char` is the type for single characters. The type `char` can be treated as an integer type, but we do not encourage you to do so. The commonly used

[1] C++ makes a distinction between *declaring* and *defining* an identifier. When an identifier is declared, the name is introduced. When it is defined, storage for the named item is allocated. For the kind of variables we discuss in this chapter, and for much more of the book, what we are calling a *variable declaration* both declares the variable and defines the variable, that is, allocates storage for the variable. Many authors blur the distinction between variable definition and variable declaration. The difference between declaring and defining an identifier is more important for other kinds of identifiers, which we will encounter in later chapters.

Display 1.2 Simple Types

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,767 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,647 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,647 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits
`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true`, `false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

type for floating-point numbers is `double`, and so you should use `double` for floating-point numbers unless you have a specific reason to use one of the other floating-point types. The type `bool` (short for *Boolean*) has the values `true` and `false`. It is not an integer type, but to accommodate older code, you can convert back and forth between `bool` and any of the integer types. In addition, the standard library named `string` provides the type `string`, which is used for strings of characters. The programmer can define types for arrays, classes, and pointers, all of which are discussed in later chapters of this book.

Variable Declarations

All variables must be declared before they are used. The syntax for variable declarations is as follows.

SYNTAX

Type_Name Variable_Name_1, Variable_Name_2,. . .;

EXAMPLE

```
int count, numberOfDragons, numberOfTrolls;
double distance;
```

unsigned

Each of the integer types has an **unsigned** version that includes only nonnegative values. These types are `unsigned short`, `unsigned int`, and `unsigned long`. Their ranges do not exactly correspond to the ranges of the positive values of the types `short`, `int`, and `long`, but are likely to be larger (since they use the same storage as their corresponding types `short`, `int`, or `long`, but need not remember a sign). You are unlikely to need these types, but may run into them in specifications for predefined functions in some of the C++ libraries, which we discuss in Chapter 3.

ASSIGNMENT STATEMENTS

assignment statement

The most direct way to change the value of a variable is to use an **assignment statement**. In C++ the equal sign is used as the assignment operator. An assignment statement always consists of a variable on the left-hand side of the equal sign and an expression on the right-hand side. An assignment statement ends with a semicolon. The expression on the right-hand side of the equal sign may be a variable, a number, or a more complicated expression made up of variables, numbers, operators, and function invocations. An assignment statement instructs the computer to evaluate (that is, to compute the value of) the expression on the right-hand side of the equal sign and to set the value of the variable on the left-hand side equal to the value of that expression. The following are examples of C++ assignment statements:

```
totalWeight = oneWeight * numberOfBeans;
temperature = 98.6;
count = count + 2;
```

The first assignment statement sets the value of `totalWeight` equal to the number in the variable `oneWeight` multiplied by the number in `numberOfBeans`. (Multiplication is expressed using the asterisk, `*`, in C++.) The second assignment statement sets the

value of `temperature` to `98.6`. The third assignment statement increases the value of the variable `count` by `2`.

Assignment Statements

In an assignment statement, first the expression on the right-hand side of the equal sign is evaluated and then the variable on the left-hand side of the equal sign is set equal to this value.

SYNTAX

Variable = *Expression*;

EXAMPLES

```
distance = rate * time;
count = count + 2;
```

In C++, assignment statements can be used as expressions. When used as an expression, an assignment statement returns the value assigned to the variable. For example, consider

```
n = (m = 2);
```

The subexpression `(m = 2)` both changes the value of `m` to `2` and returns the value `2`. Thus, this sets both `n` and `m` equal to `2`. As you will see when we discuss precedence of operators in detail in Chapter 2, you can omit the parentheses, so the assignment statement under discussion can be written as

```
n = m = 2;
```

We advise you not to use an assignment statement as an expression, but you should be aware of this behavior because it will help you understand certain kinds of coding errors. For one thing, it will explain why you will not get an error message when you mistakenly write

```
n = m = 2;
```

when you meant to write

```
n = m + 2;
```

(This is an easy mistake to make since = and + are on the same keyboard key.)

Lvalues and Rvalues

Authors often refer to *lvalue* and *rvalue* in C++ books. An **lvalue** is anything that can appear on the left-hand side of an assignment operator (=), which means any kind of variable. An **rvalue** is anything that can appear on the right-hand side of an assignment operator, which means any expression that evaluates to a value.

PITFALL

Uninitialized Variables

A variable has no meaningful value until a program gives it one. For example, if the variable `minimumNumber` has not been given a value either as the left-hand side of an assignment statement or by some other means (such as being given an input value with a `cin` statement), then the following is an error:

```
desiredNumber = minimumNumber + 10;
```

uninitialized variable

This is because `minimumNumber` has no meaningful value, and so the entire expression on the right-hand side of the equal sign has no meaningful value. A variable like `minimumNumber` that has not been given a value is said to be **uninitialized**. This situation is, in fact, worse than it would be if `minimumNumber` had no value at all. An uninitialized variable, like `minimumNumber`, will simply have some garbage value. The value of an uninitialized variable is determined by whatever pattern of zeros and ones was left in its memory location by the last program that used that portion of memory.

One way to avoid an uninitialized variable is to initialize variables at the same time they are declared. This can be done by adding an equal sign and a value, as follows:

```
int minimumNumber = 3;
```

This both declares `minimumNumber` to be a variable of type `int` and sets the value of the variable `minimumNumber` equal to 3. You can use a more complicated expression involving operations such as addition or multiplication when you initialize a variable inside the declaration in this way. As another example, the following declares three variables and initializes two of them:

```
double rate = 0.07, time, balance = 0.00;
```

C++ allows an alternative notation for initializing variables when they are declared. This alternative notation is illustrated by the following, which is equivalent to the preceding declaration:

```
double rate(0.07), time, balance(0.00);
```

Initializing Variables in Declarations

You can initialize a variable (that is, give it a value) at the time that you declare the variable.

SYNTAX

```
Type_Name Variable_Name_1 = Expression_for_Value_1,
          Variable_Name_2 = Expresssion_for_Value_2,...;
```

EXAMPLES

```
int count = 0, limit = 10, fudgeFactor = 2;
double distance = 999.99;
```

SYNTAX

Alternative syntax for initializing in declarations:

```
Type_Name Variable_Name_1 (Expression_for_Value_1),
          Variable_Name_2 (Expression_for_Value_2),...;
```

EXAMPLES

```
int count(0), limit(10), fudgeFactor(2);
double distance(999.99);
```

TIP

Use Meaningful Names

Variable names and other names in a program should at least hint at the meaning or use of the thing they are naming. It is much easier to understand a program if the variables have meaningful names. Contrast

```
x = y * z;
```

with the more suggestive

```
distance = speed * time;
```

The two statements accomplish the same thing, but the second is easier to understand.

MORE ASSIGNMENT STATEMENTS

A shorthand notation exists that combines the assignment operator (=) and an arithmetic operator so that a given variable can have its value changed by adding, subtracting, multiplying by, or dividing by a specified value. The general form is

Variable Operator= *Expression*

which is equivalent to

Variable = *Variable Operator* (*Expression*)

The *Expression* can be another variable, a constant, or a more complicated arithmetic expression. The following list gives examples.

EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

SELF-TEST EXERCISES

1. Give the declaration for two variables called `feet` and `inches`. Both variables are of type `int` and both are to be initialized to zero in the declaration. Give both initialization alternatives.
2. Give the declaration for two variables called `count` and `distance`. `count` is of type `int` and is initialized to zero. `distance` is of type `double` and is initialized to `1.5`. Give both initialization alternatives.
3. Write a program that contains statements that output the values of five or six variables that have been defined, but not initialized. Compile and run the program. What is the output? Explain.

ASSIGNMENT COMPATIBILITY

As a general rule, you cannot store a value of one type in a variable of another type. For example, most compilers will object to the following:

```
int intVariable;
intVariable = 2.99;
```

The problem is a type mismatch. The constant `2.99` is of type `double`, and the variable `intVariable` is of type `int`. Unfortunately, not all compilers will react the same way to the above assignment statement. Some will issue an error message, some will give only a warning message, and some compilers will not object at all. Even if the compiler does allow you to use the above assignment, it will give `intVariable` the `int` value `2`, not the value `3`. Since you cannot count on your compiler accepting the above assignment, you should not assign a `double` value to a variable of type `int`.

Even if the compiler will allow you to mix types in an assignment statement, in most cases you should not. Doing so makes your program less portable, and it can be confusing.

assigning `int` values to `double` variables

There are some special cases in which it is permitted to assign a value of one type to a variable of another type. It is acceptable to assign a value of an integer type, such as `int`, to a variable of a floating-point type, such as type `double`. For example, the following is both legal and acceptable style:

```
double doubleVariable;
doubleVariable = 2;
```

The above will set the value of the variable named `doubleVariable` equal to `2.0`.

Although it is usually a bad idea to do so, you can store an `int` value such as `65` in a variable of type `char` and you can store a letter such as `'Z'` in a variable of type `int`. For many purposes, the C language considers characters to be small integers, and perhaps unfortunately, C++ inherited this from C. The reason for allowing this is that variables of type `char` consume less memory than variables of type `int`; thus, doing arithmetic with variables of type `char` can save some memory. However, it is clearer to use the type `int` when you are dealing with integers and to use the type `char` when you are dealing with characters.

mixing types

The general rule is that you cannot place a value of one type in a variable of another type—though it may seem that there are more exceptions to the rule than there are cases that follow the rule. Even if the compiler does not enforce this rule very strictly, it is a good rule to follow. Placing data of one type in a variable of another type can cause problems because the value must be changed to a value of the appropriate type and that value may not be what you would expect.

integers and Booleans

Values of type `bool` can be assigned to variables of an integer type (`short`, `int`, `long`), and integers can be assigned to variables of type `bool`. However, it is poor style to do this. For completeness and to help you read other people's code, here are the details: When assigned to a variable of type `bool`, any nonzero integer will be stored as the value `true`. Zero will be stored as the value `false`. When assigning a `bool` value to an integer variable, `true` will be stored as `1`, and `false` will be stored as `0`.

LITERALS

literal constant

A **literal** is a name for one specific value. Literals are often called **constants** in contrast to variables. Literals or constants do not change value; variables can change their values. Integer constants are written in the way you are used to writing numbers. Constants of type `int` (or any other integer type) must not contain a decimal point. Constants of type `double` may be written in either of two forms. The simple form for `double` constants is like the everyday way of writing decimal fractions. When written in this form a `double` constant must contain a decimal point. No number constant (either integer or floating-point) in C++ may contain a comma.

scientific notation or floating-point notation

A more complicated notation for constants of type `double` is called **scientific notation** or **floating-point notation** and is particularly handy for writing very large numbers and very small fractions. For instance, 3.67×10^{17}, which is the same as

```
367000000000000000.0
```

is best expressed in C++ by the constant `3.67e17`. The number 5.89×10^{-6}, which is the same as `0.00000589`, is best expressed in C++ by the constant `5.89e-6`. The `e` stands for *exponent* and means "multiply by 10 to the power that follows." The `e` may be either uppercase or lowercase.

Think of the number after the `e` as telling you the direction and number of digits to move the decimal point. For example, to change `3.49e4` to a numeral without an `e`, you move the decimal point four places to the right to obtain `34900.0`, which is another way of writing the same number. If the number after the `e` is negative, you move the decimal point the indicated number of spaces to the left, inserting extra zeros if need be. So, `3.49e-2` is the same as `0.0349`.

The number before the `e` may contain a decimal point, although it is not required. However, the exponent after the `e` definitely must *not* contain a decimal point.

What Is Doubled?

Why is the type for numbers with a fractional part called `double`? Is there a type called "single" that is half as big? No, but something like that is true. Many programming languages traditionally used two types for numbers with a fractional part. One type used less storage and was very imprecise (that is, it did not allow very many significant digits). The second type used *double* the amount of storage and so was much more precise; it also allowed numbers that were larger (although programmers tend to care more about precision than about size). The kind of numbers that used twice as much storage were called *double-precision* numbers; those that used less storage were called *single precision*. Following this tradition, the type that (more or less) corresponds to this double-precision type was named `double` in C++. The type that corresponds to single precision in C++ was called `float`. C++ also has a third type for numbers with a fractional part, which is called `long double`.

Constants of type `char` are expressed by placing the character in single quotes, as illustrated in what follows:

```
char symbol = 'Z';
```

Note that the left and right single quote symbol are the same symbol.

Constants for strings of characters are given in double quotes, as illustrated by the following line taken from Display 1.1:

```
cout << "How many programming languages have you used? ";
```

quotes

Be sure to notice that string constants are placed inside double quotes, while constants of type `char` are placed inside single quotes. The two kinds of quotes mean different things. In particular, `'A'` and `"A"` mean different things. `'A'` is a value of type `char` and can be stored in a variable of type `char`. `"A"` is a string of characters. The fact that the string happens to contain only one character does *not* make `"A"` a value of type `char`. Also notice that for both strings and characters, the left and right quotes are the same.

C-string

Strings in double quotes, like `"Hello"`, are often called **C-strings**. In Chapter 9 we will see that C++ has more than one kind of string, and this particular kind happens to be called C-strings.

The type `bool` has two constants, `true` and `false`. These two constants may be assigned to a variable of type `bool` or used anyplace else an expression of type `bool` is allowed. They must be spelled with all lowercase letters.

ESCAPE SEQUENCES

escape sequence

A backslash, `\` , preceding a character tells the compiler that the sequence following the backslash does not have the same meaning as the character appearing by itself. Such a sequence is called an **escape sequence**. The sequence is typed in as two characters with no space between the symbols. Several escape sequences are defined in C++.

If you want to put a backslash, `\`, or a quote symbol, `"`, into a string constant, you must escape the ability of the `"` to terminate a string constant by using `\"`, or the ability of the `\` to escape, by using `\\`. The `\\` tells the compiler you mean a real backslash, `\`, not an escape sequence; the `\"` tells it you mean a real quote, not the end of a string constant.

A stray `\`, say `\z`, in a string constant will have different effects on different compilers. One compiler may simply give back a `z`; another might produce an error. The ANSI/ISO standard states that unspecified escape sequences have undefined behavior. This means a compiler can do anything its author finds convenient. The consequence is that code that uses undefined escape sequences is not portable. You should not use any escape sequences other than those provided by the C++ standard. These C++ control characters are listed in Display 1.3.

NAMING CONSTANTS

Numbers in a computer program pose two problems. The first is that they carry no mnemonic value. For example, when the number `10` is encountered in a program, it gives no hint of its significance. If the program is a banking program, it might be the number of branch offices or the number of teller windows at the main office. To understand the program, you need to know the significance of each constant. The second problem is that when a program needs to have some numbers changed, the changing tends to introduce errors. Suppose that `10` occurs twelve times in a banking program—four of the times it represents the number of branch offices, and eight of the times it represents the number of teller windows at the main office. When the bank opens a new branch and the program needs to be updated, there is a good chance

Display 1.3 Some Escape Sequences

SEQUENCE	MEANING
\n	New line
\r	Carriage return (Positions the cursor at the start of the current line. You are not likely to use this very much.)
\t	(Horizontal) Tab (Advances the cursor to the next tab stop.)
\a	Alert (Sounds the alert noise, typically a bell.)
\\	Backslash (Allows you to place a backslash in a quoted expression.)
\'	Single quote (Mostly used to place a single quote inside single quotes.)
\"	Double quote (Mostly used to place a double quote inside a quoted string.)
The following are not as commonly used, but we include them for completeness:	
\v	Vertical tab
\b	Backspace
\f	Form feed
\?	Question mark

that some of the 10s that should be changed to 11 will not be, or some that should not be changed will be. The way to avoid these problems is to name each number and use the name instead of the number within your program. For example, a banking program might have two constants with the names BRANCH_COUNT and WINDOW_COUNT. Both these numbers might have a value of 10, but when the bank opens a new branch, all you need do to update the program is change the definition of BRANCH_COUNT.

How do you name a number in a C++ program? One way to name a number is to initialize a variable to that number value, as in the following example:

```
int BRANCH_COUNT = 10;
int WINDOW_COUNT = 10;
```

There is, however, one problem with this method of naming number constants: You might inadvertently change the value of one of these variables. C++ provides a way of marking an initialized variable so that it cannot be changed. If your program tries to change one of these variables, it produces an error condition. To mark a variable declaration so that the value of the variable cannot be changed, precede the declaration with the word `const` (which is an abbreviation of *constant*). For example:

const

```
const int BRANCH_COUNT = 10;
const int WINDOW_COUNT = 10;
```

If the variables are of the same type, it is possible to combine the above lines into one declaration, as follows:

```
const int BRANCH_COUNT = 10, WINDOW_COUNT = 10;
```

However, most programmers find that placing each name definition on a separate line is clearer. The word `const` is often called a **modifier**, because it modifies (restricts) the variables being declared.

modifier

A variable declared using the `const` modifier is often called a **declared constant**. Writing declared constants in all uppercase letters is not required by the C++ language, but it is standard practice among C++ programmers.

declared constant

Once a number has been named in this way, the name can then be used anywhere the number is allowed, and it will have exactly the same meaning as the number it names. To change a named constant, you need only change the initializing value in the `const` variable declaration. The meaning of all occurrences of `BRANCH_COUNT`, for instance, can be changed from `10` to `11` simply by changing the initializing value of `10` in the declaration of `BRANCH_COUNT`.

Display 1.4 contains a simple program that illustrates the use of the declaration modifier `const`.

ARITHMETIC OPERATORS AND EXPRESSIONS

As in most other languages, C++ allows you to form expressions using variables, constants, and the arithmetic operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulo, remainder). These expressions can be used anyplace it is legal to use a value of the type produced by the expression.

All the arithmetic operators can be used with numbers of type `int`, numbers of type `double`, and even with one number of each type. However, the type of the value produced and the exact value of the result depend on the types of the numbers being combined. If both operands (that is, both numbers) are of type `int`, then the result of combining them with an arithmetic operator is of type `int`. If one or both of the operands are of type `double`, then the result is of type `double`. For example, if the variables `baseAmount` and `increase` are of type `int`, then the number produced by the following expression is of type `int`:

mixing types

```
baseAmount + increase
```

However, if one or both of the two variables are of type `double`, then the result is of type `double`. This is also true if you replace the operator + with any of the operators –, *, or /.

More generally, you can combine any of the arithmetic types in expressions. If all the types are integer types, the result will be the integer type. If at least one of the subexpressions is of a floating-point type, the result will be a floating-point type. C++ tries its best to make the type of an expression either `int` or `double`, but if the value produced by the expression is not of one of these types because of the value's size, a suitable different integer or floating-point type will be produced.

precedence rules

You can specify the order of operations in an arithmetic expression by inserting parentheses. If you omit parentheses, the computer will follow rules called **precedence rules** that determine the order in which the operations, such as addition and multiplication, are performed. These precedence rules are similar to rules used in algebra and other mathematics classes. For example:

```
x + y * z
```

is evaluated by first doing the multiplication and then the addition. Except in some standard cases, such as a string of additions or a simple multiplication embedded

Display 1.4 Named Constant


```
1  #include <iostream>
2  using namespace std;
3
4  int main( )
5  {
6      const double RATE = 6.9;
7      double deposit;

8      cout << "Enter the amount of your deposit $";
9      cin >> deposit;

10     double newBalance;
11     newBalance = deposit + deposit*(RATE/100);
12     cout << "In one year, that deposit will grow to\n"
13          << "$" << newBalance << " an amount worth waiting for.\n";

14     return 0;
15 }
```

SAMPLE DIALOGUE

```
Enter the amount of your deposit $100
In one year, that deposit will grow to
$106.9 an amount worth waiting for.
```

Naming Constants with the const Modifier

When you initialize a variable inside a declaration, you can mark the variable so that the program is not allowed to change its value. To do this, place the word const in front of the declaration, as described below:

SYNTAX

```
const Type_Name Variable_Name = Constant;
```

EXAMPLES

```
const int MAX_TRIES = 3;
const double PI = 3.14159;
```

inside an addition, it is usually best to include the parentheses, even if the intended order of operations is the one dictated by the precedence rules. The parentheses make the expression easier to read and less prone to programmer error. A complete set of C++ precedence rules is given in Appendix 2.

INTEGER AND FLOATING-POINT DIVISION

integer division

When used with one or both operands of type `double`, the division operator, `/`, behaves as you might expect. However, when used with two operands of type `int`, the division operator yields the integer part resulting from division. In other words, integer division discards the part after the decimal point. So, `10/3` is `3` (not `3.3333...`), `5/2` is `2` (not `2.5`), and `11/3` is `3` (not `3.6666...`). Notice that the number *is not rounded*; the part after the decimal point is discarded no matter how large it is.

the % operator

The operator `%` can be used with operands of type `int` to recover the information lost when you use `/` to do division with numbers of type `int`. When used with values of type `int`, the two operators `/` and `%` yield the two numbers produced when you perform the long division algorithm you learned in grade school. For example, `17` divided by `5` yields `3` with a remainder of `2`. The `/` operation yields the number of times one number "goes into" another. The `%` operation gives the remainder. For example, the statements

```
cout << "17 divided by 5 is " << (17/5) << "\n";
cout << "with a remainder of " << (17%5) << "\n";
```

yield the following output:

```
17 divided by 5 is 3
with a remainder of 2
```

negative integers in division

When used with negative values of type `int`, the result of the operators `/` and `%` can be different for different implementations of C++. Thus, you should use `/`and `%` with `int` values only when you know that both values are nonnegative.

PITFALL

Division with Whole Numbers

When you use the division operator / on two integers, the result is an integer. This can be a problem if you expect a fraction. Moreover, the problem can easily go unnoticed, resulting in a program that looks fine but is producing incorrect output without you even being aware of the problem. For example, suppose you are a landscape architect who charges $5,000 per mile to landscape a highway, and suppose you know the length of the highway you are working on in feet. The price you charge can easily be calculated by the following C++ statement:

```
totalPrice = 5000 * (feet/5280.0);
```

This works because there are 5,280 feet in a mile. If the stretch of highway you are landscaping is 15,000 feet long, this formula will tell you that the total price is

```
5000 * (15000/5280.0)
```

Your C++ program obtains the final value as follows: `15000/5280.0` is computed as `2.84`. Then the program multiplies `5000` by `2.84` to produce the value `14200.00`. With the aid of your C++ program, you know that you should charge $14,200 for the project.
Now suppose the variable `feet` is of type `int`, and you forget to put in the decimal point and the zero, so that the assignment statement in your program reads

```
totalPrice = 5000 * (feet/5280);
```

It still looks fine, but will cause serious problems. If you use this second form of the assignment statement, you are dividing two values of type `int`, so the result of the division `feet/5280` is `15000/5280`, which is the `int` value 2 (instead of the value `2.84` that you think you are getting). The value assigned to `totalPrice` is thus `5000*2`, or `10000.00`. If you forget the decimal point, you will charge $10,000. However, as we have already seen, the correct value is $14,200. A missing decimal point has cost you $4,200. Note that this will be true whether the type of `totalPrice` is `int` or `double`; the damage is done before the value is assigned to `totalPrice`.

SELF-TEST EXERCISES

4. Convert each of the following mathematical formulas to a C++ expression.

$$3x \qquad 3x + y \qquad \frac{x + y}{7} \qquad \frac{3x + y}{z + 2}$$

5. What is the output of the following program lines when they are embedded in a correct program that declares all variables to be of type `char`?

```
a = 'b';
b = 'c';
c = a;
cout << a << b << c << 'c';
```

SELF-TEST EXERCISES (continued)

6. What is the output of the following program lines when they are embedded in a correct program that declares `number` to be of type `int`?

```
number = (1/3) * 3;
cout << "(1/3) * 3 is equal to " << number;
```

7. Write a complete C++ program that reads two whole numbers into two variables of type `int` and then outputs both the whole number part and the remainder when the first number is divided by the second. This can be done using the operators `/` and `%`.

8. Given the following fragment that purports to convert from degrees Celsius to degrees Fahrenheit, answer the following questions:

```
double c = 20;
double f;
f = (9/5) * c + 32.0;
```

   a. What value is assigned to `f`?

   b. Explain what is actually happening, and what the programmer likely wanted.

   c. Rewrite the code as the programmer intended.

TYPE CASTING

A **type cast** is a way of changing a value of one type to a value of another type. A type cast is a kind of function that takes a value of one type and produces a value of another type that is C++'s best guess of an equivalent value. C++ has four to six different kinds of casts, depending on how you count them. There is an older form of type cast that has two notations for expressing it, and there are four new kinds of type casts introduced with the latest standard. The new kinds of type casts were designed as replacements for the older form; in this book, we will use the newer kinds. However, C++ retains the older kind(s) of cast along with the newer kinds, so we will briefly describe the older kind as well.

type cast

Let's start with the newer kinds of type casts. Consider the expression `9/2`. In C++ this expression evaluates to `4` because when both operands are of an integer type, C++ performs integer division. In some situations, you might want the answer to be the `double` value `4.5`. You can get a result of `4.5` by using the "equivalent" floating-point value `2.0` in place of the integer value `2`, as in `9/2.0`, which evaluates to `4.5`. But what if the `9` and the `2` are the values of variables of type `int` named `n` and `m`? Then, `n/m` yields `4`. If you want floating-point division in this case, you must do a type cast from `int` to `double` (or another floating-point type), such as in the following:

```
double ans = n/static_cast<double>(m);
```

The expression

```
static_cast<double>(m)
```

is a type cast. The expression `static_cast<double>` is like a function that takes an `int` argument (actually, an argument of almost any type) and returns an "equivalent" value of type `double`. So, if the value of `m` is `2`, the expression `static_cast<double>(m)` returns the `double` value `2.0`.

Note that `static_cast<double>(n)` does not change the value of the variable `n`. If `n` has the value `2` before this expression is evaluated, then `n` still has the value `2` after the expression is evaluated. (If you know what a function is in mathematics or in some programming language, you can think of `static_cast<double>` as a function that returns an "equivalent" value of type `double`.)

You may use any type name in place of `double` to obtain a type cast to another type. We said this produces an "equivalent" value of the target type. The word equivalent is in quotes because there is no clear notion of equivalent that applies between any two types. In the case of a type cast from an integer type to a floating-point type, the effect is to add a decimal point and a zero. The type cast in the other direction, from a floating-point type to an integer type, simply deletes the decimal point and all digits after the decimal point. Note that when type casting from a floating-point type to an integer type, the number is truncated, not rounded. `static_cast<int>(2.9)` is `2`; it is not `3`.

This `static_cast` is the most common kind of type cast and the only one we will use for some time. For completeness and reference value, we list all four kinds of type casts. Some may not make sense until you reach the relevant topics. If some or all of the remaining three kinds do not make sense to you at this point, do not worry. The four kinds of type cast are as follows:

```
static_cast<Type>(Expression)
const_cast<Type>(Expression)
dynamic_cast<Type>(Expression)
reinterpret_cast<Type>(Expression)
```

We have already discussed `static_cast`. It is a general-purpose type cast that applies in most "ordinary" situations. The `const_cast` is used to cast away constantness. The `dynamic_cast` is used for safe downcasting from one type to a descendent type in an inheritance hierarchy. The `reinterpret_cast` is an implementation-dependent cast that we will not discuss in this book and that you are unlikely to need. (These descriptions may not make sense until you cover the appropriate topics, where they will be discussed further. For now, we only use `static_cast`.)

The older form of type casting is approximately equivalent to the `static_cast` kind of type casting but uses a different notation. One of the two notations uses a type name as if it were a function name. For example `int(9.3)` returns the `int` value `9`; `double(42)` returns the value `42.0`. The second, equivalent, notation for the older form of type casting would write `(double)42` instead of `double(42)`. Either notation can be used with variables or other more complicated expressions instead of just with constants.

Although C++ retains this older form of type casting, you are encouraged to use the newer form of type casting. (Someday, the older form may go away, although there is, as yet, no such plan for its elimination.)

As we noted earlier, you can always assign a value of an integer type to a variable of a floating-point type, as in

```
double d = 5;
```

In such cases C++ performs an automatic type cast, converting the `5` to `5.0` and placing `5.0` in the variable `d`. You cannot store the `5` as the value of `d` without a type cast, but sometimes C++ does the type cast for you. Such an automatic conversion is sometimes called a **type coercion**.

type coercion

INCREMENT AND DECREMENT OPERATORS

The ++ in the name of the C++ language comes from the increment operator, ++. The **increment operator** adds 1 to the value of a variable. The **decrement operator**, --, subtracts 1 from the value of a variable. They are usually used with variables of type `int`, but they can be used with any numeric type. If `n` is a variable of a numeric type, then `n++` increases the value of `n` by 1 and `n--` decreases the value of `n` by 1. So `n++` and `n--` (when followed by a semicolon) are executable statements. For example, the statements

increment operator
decrement operator

```
int n = 1, m = 7;
n++;
cout << "The value of n is changed to " << n << "\n";
m--;
cout << "The value of m is changed to " << m << "\n";
```

yield the following output:

```
The value of n is changed to 2
The value of m is changed to 6
```

An expression like `n++` returns a value as well as changing the value of the variable `n`, so `n++` can be used in an arithmetic expression such as

```
2*(n++)
```

The expression `n++` first returns the value of the variable `n`, and *then* the value of `n` is increased by 1. For example, consider the following code:

```
int n = 2;
int valueProduced = 2*(n++);
cout << valueProduced << "\n";
cout << n << "\n";
```

This code will produce the output:

```
4
3
```

Notice the expression `2*(n++)`. When C++ evaluates this expression, it uses the value that `number` has *before* it is incremented, not the value that it has after it is incremented. Thus, the value produced by the expression `n++` is `2`, even though the increment operator changes the value of `n` to `3`. This may seem strange, but sometimes it is just what you want. And, as you are about to see, if you want an expression that behaves differently, you can have it.

v++ versus ++v

The expression `n++` evaluates to the value of the variable `n`, and *then* the value of the variable `n` is incremented by 1. If you reverse the order and place the `++` in front of the variable, the order of these two actions is reversed. The expression `++n` first increments the value of the variable `n` and then returns this increased value of `n`. For example, consider the following code:

```
int n = 2;
int valueProduced = 2*(++n);
cout << valueProduced << "\n";
cout << n << "\n";
```

This code is the same as the previous piece of code except that the `++` is before the variable, so this code will produce the following output:

```
6
3
```

Notice that the two increment operators in `n++` and `++n` have the same effect on a variable `n`: They both increase the value of `n` by 1. But the two expressions evaluate to different values. Remember, if the `++` is *before* the variable, the incrementing is done *before* the value is returned; if the `++` is *after* the variable, the incrementing is done *after* the value is returned.

Everything we said about the increment operator applies to the decrement operator as well, except that the value of the variable is decreased by 1 rather than increased by 1. For example, consider the following code:

```
int n = 8;
int valueProduced = n--;
cout << valueProduced << "\n";
cout << n << "\n";
```

This produces the output

```
8
7
```

On the other hand, the code

```
int n = 8;
int valueProduced = --n;
cout << valueProduced << "\n";
cout << n << "\n";
```

produces the output

```
7
7
```

n-- returns the value of n and then decrements n; on the other hand, --n first decrements n and then returns the value of n.

You cannot apply the increment and decrement operators to anything other than a single variable. Expressions such as `(x + y)++`, `--(x + y)`, `5++`, and so forth, are all illegal in C++.

The increment and decrement operators can be dangerous when used inside more complicated expressions, as explained in the Pitfall.

PITFALL

Order of Evaluation

For most operators, the order of evaluation of subexpressions is not guaranteed. In particular, you normally cannot assume that the order of evaluation is left to right. For example, consider the following expression:

```
n + (++n)
```

Suppose n has the value 2 before the expression is evaluated. Then, if the first expression is evaluated first, the result is `2 + 3`. If the second expression is evaluated first, the result is `3 + 3`. Since C++ does not guarantee the order of evaluation, the expression could evaluate to either 5 or 6. The moral is that you should not program in a way that depends on order of evaluation, except for the operators discussed in the next paragraph.

Some operators do guarantee that their order of evaluation of subexpressions is left to right. For the operators && (and), || (or), and the comma operator (which is discussed in Chapter 2), C++ guarantees that the order of evaluations is left to right. Fortunately, these are the operators for which you are most likely to want a predicable order of evaluation. For example, consider

```
(n <= 2) && (++n > 2)
```

Suppose n has the value 2, before the expression is evaluated. In this case you know that the subexpression `(n <= 2)` is evaluated before the value of n is incremented. You thus know that `(n <= 2)` will evaluate to `true` and so the entire expression will evaluate to `true`.

(continued)

PITFALL (continued)

Do not confuse order of operations (by precedence rules) with order of evaluation. For example,

```
(n + 2) * (++n) + 5
```

always means

```
((n + 2) * (++n)) + 5
```

However, it is not clear whether the ++n is evaluated before or after the n + 2. Either one could be evaluated first.
Now you know why we said that it is usually a bad idea to use the increment (++) and decrement (--) operators as subexpressions of larger expressions.
If this is too confusing, just follow the simple rule of not writing code that depends on the order of evaluation of subexpressions.

1.3 Console Input/Output

Garbage in means garbage out.

Programmer's saying

Simple console input is done with the objects `cin`, `cout`, and `cerr`, all of which are defined in the library `iostream`. In order to use this library, your program should contain the following near the start of the file containing your code:

```
#include <iostream>
using namespace std;
```

OUTPUT USING `cout`

`cout`

The values of variables as well as strings of text may be output to the screen using **`cout`**. Any combination of variables and strings can be output. For example, consider the following from the program in Display 1.1:

```
cout << "Hello reader.\n"
     << "Welcome to C++.\n";
```

This statement outputs two strings, one per line. Using `cout`, you can output any number of items, each either a string, variable, or more complicated expression. Simply insert a << before each thing to be output.

As another example, consider the following:

```
cout << numberOfGames << " games played.";
```

This statement tells the computer to output two items: the value of the variable `numberOfGames` and the quoted string `" games played."`.

Notice that you do not need a separate copy of the object `cout` for each item output. You can simply list all the items to be output, preceding each item to be output with the arrow symbols <<. The previous single `cout` statement is equivalent to the following two `cout` statements:

```
cout << numberOfGames;
cout << " games played.";
```

expression in a `cout` statement

You can include arithmetic expressions in a `cout` statement, as shown by the following example, where `price` and `tax` are variables:

```
cout << "The total cost is $" << (price + tax);
```

Parentheses around arithmetic expressions, such as `price + tax`, are required by some compilers, so it is best to include them.

The two < symbols should be typed without any space between them. The arrow notation << is often called the **insertion operator**. The entire `cout` statement ends with a semicolon.

spaces in output

Notice the spaces inside the quotes in our examples. The computer does not insert any extra space before or after the items output by a `cout` statement, which is why the quoted strings in the examples often start or end with a blank. The blanks keep the various strings and numbers from running together. If all you need is a space and there is no quoted string where you want to insert the space, then use a string that contains only a space, as in the following:

```
cout << firstNumber << " " << secondNumber;
```

NEW LINES IN OUTPUT

As noted in the subsection on escape sequences, `\n` tells the computer to start a new line of output. Unless you tell the computer to go to the next line, it will put all the output on the same line. Depending on how your screen is set up, this can produce anything from arbitrary line breaks to output that runs off the screen. Notice that the `\n` goes inside the quotes. In C++, going to the next line is considered to be a special character, and the way you spell this special character inside a quoted string is `\n`, with no space between the two symbols in `\n`. Although it is typed as two symbols, C++ considers `\n` to be a single character that is called the **newline character**.

newline character

If you wish to insert a blank line in the output, you can output the newline character `\n` by itself:

```
cout << "\n";
```

Another way to output a blank line is to use `endl`, which means essentially the same thing as `"\n"`. So you can also output a blank line as follows:

```
cout << endl;
```

deciding between `\n` and `endl`

Although `"\n"` and `endl` mean the same thing, they are used slightly differently; `\n` must always be inside quotes, and `endl` should not be placed in quotes.

A good rule for deciding whether to use `\n` or `endl` is the following: If you can include the `\n` at the end of a longer string, then use `\n`, as in the following:

```
cout << "Fuel efficiency is "
     << mpg << " miles per gallon\n";
```

On the other hand, if the `\n` would appear by itself as the short string `"\n"`, then use `endl` instead:

```
cout << "You entered " << number << endl;
```

Starting New Lines in Output

To start a new output line, you can include `\n` in a quoted string, as in the following example:

```
cout << "You have definitely won\n"
     << "one of the following prizes:\n";
```

Recall that `\n` is typed as two symbols with no space in between the two symbols.
Alternatively, you can start a new line by outputting `endl`. An equivalent way to write the above `cout` statement is as follows:

```
cout << "You have definitely won" << endl
     << "one of the following prizes:" << endl;
```

TIP

End Each Program with `\n` or `endl`

It is a good idea to output a newline instruction at the end of every program. If the last item to be output is a string, then include a `\n` at the end of the string; if not, output an `endl` as the last output action in your program. This serves two purposes. Some compilers will not output the last line of your program unless you include a newline instruction at the end. On other systems, your program may work fine without this final newline instruction, but the next program that is run will have its first line of output mixed with the last line of the previous program. Even if neither of these problems occurs on your system, putting a newline instruction at the end will make your programs more portable.

FORMATTING FOR NUMBERS WITH A DECIMAL POINT

format for `double` values

When the computer outputs a value of type `double`, the format may not be what you would like. For example, the following simple `cout` statement can produce any of a wide range of outputs:

```
cout << "The price is $" << price << endl;
```

If `price` has the value `78.5`, the output might be

```
The price is $78.500000
```

or it might be

```
The price is $78.5
```

or it might be output in the following notation (which was explained in the subsection entitled **Literals**):

```
The price is $7.850000e01
```

It is extremely unlikely that the output will be the following, however, even though this is the format that makes the most sense:

```
The price is $78.50
```

To ensure that the output is in the form you want, your program should contain some sort of instructions that tell the computer how to output the numbers.

magic formula

There is a "magic formula" that you can insert in your program to cause numbers that contain a decimal point, such as numbers of type `double`, to be output in everyday notation with the exact number of digits after the decimal point that you specify. If you want two digits after the decimal point, use the following magic formula:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

outputting money amounts

If you insert the preceding three statements in your program, then any `cout` statements that follow these statements will output values of any floating-point type in ordinary notation, with exactly two digits after the decimal point. For example, suppose the following `cout` statement appears somewhere after this magic formula and suppose the value of `price` is `78.5`.

```
cout << "The price is $" << price << endl;
```

The output will then be as follows:

```
The price is $78.50
```

You may use any other nonnegative whole number in place of 2 to specify a different number of digits after the decimal point. You can even use a variable of type `int` in place of the 2.

We will explain this magic formula in detail in Chapter 12. For now, you should think of this magic formula as one long instruction that tells the computer how you want it to output numbers that contain a decimal point.

If you wish to change the number of digits after the decimal point so that different values in your program are output with different numbers of digits, you can repeat the magic formula with some other number in place of 2. However, when you repeat the magic formula, you only need to repeat the last line of the formula. If the magic formula has already occurred once in your program, then the following line will change the number of digits after the decimal point to five for all subsequent values of any floating-point type that are output:

```
cout.precision(5);
```

Outputting Values of Type double

If you insert the following "magic formula" in your program, then all numbers of type `double` (or any other type of floating-point number) will be output in ordinary notation with two digits after the decimal point:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

You can use any other nonnegative whole number in place of the 2 to specify a different number of digits after the decimal point. You can even use a variable of type `int` in place of the 2.

OUTPUT WITH cerr

cerr

The object `cerr` is used in the same way as `cout`. The object **`cerr`** sends its output to the standard error output stream, which normally is the console screen. This gives you a way to distinguish two kinds of output: `cout` for regular output, and `cerr` for error message output. If you do nothing special to change things, then `cout` and `cerr` will both send their output to the console screen, so there is no difference between them.

On some systems you can redirect output from your program to a file. This is an operating system instruction, not a C++ instruction, but it can be useful. On systems that allow for output redirection, `cout` and `cerr` may be redirected to different files.

INPUT USING cin

cin

You use **`cin`** for input more or less the same way you use `cout` for output. The syntax is similar, except that `cin` is used in place of `cout` and the arrows point in the opposite

direction. For example, in the program in Display 1.1, the variable numberOfLanguages was filled by the following cin statement:

```
cin >> numberOfLanguages;
```

You can list more than one variable in a single cin statement, as illustrated by the following:

```
cout << "Enter the number of dragons\n"
     << "followed by the number of trolls.\n";
cin >> dragons >> trolls;
```

If you prefer, the above cin statement can be written on two lines, as follows:

```
cin >> dragons
    >> trolls;
```

Notice that, as with the cout statement, there is just one semicolon for each occurrence of cin.

how cin works

When a program reaches a cin statement, it waits for input to be entered from the keyboard. It sets the first variable equal to the first value typed at the keyboard, the second variable equal to the second value typed, and so forth. However, the program does not read the input until the user presses the Return key. This allows the user to backspace and correct mistakes when entering a line of input.

separate numbers with spaces

Numbers in the input must be separated by one or more spaces or by a line break. When you use cin statements, the computer will skip over any number of blanks or line breaks until it finds the next input value. Thus, it does not matter whether input numbers are separated by one space or several spaces or even a line break.

You can read in integers, floating-point numbers, or characters using cin. Later in this book we will discuss the reading in of other kinds of data using cin.

cin Statements

A cin statement sets variables equal to values typed in at the keyboard.

SYNTAX

```
cin >> Variable_1 >> Variable_2 >>...;
```

EXAMPLES

```
cin >> number >> size;
cin >> timeLeft
    >> pointsNeeded;
```

TIP

Line Breaks in I/O

It is possible to keep output and input on the same line, and sometimes it can produce a nicer interface for the user. If you simply omit a \n or `endl` at the end of the last prompt line, then the user's input will appear on the same line as the prompt. For example, suppose you use the following prompt and input statements:

```
cout << "Enter the cost per person: $";
cin >> costPerPerson;
```

When the `cout` statement is executed, the following will appear on the screen:

```
Enter the cost per person: $
```

When the user types in the input, it will appear on the same line, like this:

```
Enter the cost per person: $1.25
```

SELF-TEST EXERCISES

9. Give an output statement that will produce the following message on the screen.

```
The answer to the question of
Life, the Universe, and Everything is 42.
```

10. Give an input statement that will fill the variable `theNumber` (of type `int`) with a number typed in at the keyboard. Precede the input statement with a prompt statement asking the user to enter a whole number.

11. What statements should you include in your program to ensure that when a number of type `double` is output, it will be output in ordinary notation with three digits after the decimal point?

12. Write a complete C++ program that writes the phrase `Hello world` to the screen. The program does nothing else.

13. Give an output statement that produces the letter `'A'`, followed by the newline character, followed by the letter `'B'`, followed by the tab character, followed by the letter `'C'`.

1.4 Program Style

In matters of grave importance, style, not sincerity, is the vital thing.

Oscar Wilde, *The Importance of Being Earnest*

C++ programming style is similar to that used in other languages. The goal is to make your code easy to read and easy to modify. We will say a bit about indenting in the next chapter. We have already discussed defined constants. Most, if not all, literals in a program should be defined constants. Choice of variable names and careful indenting should eliminate the need for very many comments, but any points that still remain unclear deserve a comment.

COMMENTS

There are two ways to insert comments in a C++ program. In C++, two slashes, `//`, are used to indicate the start of a comment. All the text between the `//` and the end of the line is a comment. The compiler simply ignores anything that follows `//` on a line. If you want a comment that covers more than one line, place a `//` on each line of the comment. The symbols `//` do not have a space between them.

Another way to insert comments in a C++ program is to use the symbol pairs `/*` and `*/`. Text between these symbols is considered a comment and is ignored by the compiler. Unlike the `//` comments, which require an additional `//` on each line, the `/*`-to-`*/` comments can span several lines, like so:

```
/*This is a comment that spans
three lines. Note that there is no comment
symbol of any kind on the second line.*/
```

Comments of the `/* */` type may be inserted anywhere in a program that a space or line break is allowed. However, they should not be inserted anywhere except where they are easy to read and do not distract from the layout of the program. Usually, comments are placed at the ends of lines or on separate lines by themselves.

Opinions differ regarding which kind of comment is best to use. Either variety (the `//` kind or the `/* */` kind) can be effective if used with care. One approach is to use the `//` comments in final code and reserve the `/**/`-style comments for temporarily commenting out code while debugging.

It is difficult to say just how many comments a program should contain. The only correct answer is "just enough," which of course conveys little to the novice programmer. It will take some experience to get a feel for when it is best to include a comment. Whenever something is important and not obvious, it merits a comment. However, too many comments are as bad as too few. A program that has a comment on each line will be so buried in comments that the structure of the program is hidden in a sea of

when to comment

obvious observations. Comments like the following contribute nothing to understanding and should not appear in a program:

```
distance = speed * time; //Computes the distance traveled.
```

1.5 Libraries and Namespaces

C++ comes with a number of standard libraries. These libraries place their definitions in a *namespace*, which is simply a name given to a collection of definitions. The techniques for including libraries and dealing with namespaces will be discussed in detail later in this book. This section discusses enough details to allow you to use the standard C++ libraries.

LIBRARIES AND `include` DIRECTIVES

#include

C++ includes a number of standard libraries. In fact, it is almost impossible to write a C++ program without using at least one of these libraries. The normal way to make a library available to your program is with an `include` directive. An `include` directive for a standard library has the form:

```
#include <Library_Name>
```

For example, the library for console I/O is `iostream`. So, most of our demonstration programs will begin

```
#include <iostream>
```

Compilers (preprocessors) can be very fussy about spacing in `include` directives. Thus, it is safest to type an `include` directive with no extra space: no space before the `#`, no space after the `#`, and no spaces inside the `<>`.

An `include` directive is simply an instruction to include the text found in a file at the location of the `include` directive. A library name is simply the name of a file that includes all the definition of items in the library. We will eventually discuss using `include` directives for things other than standard libraries, but for now we only need `include` directives for standard C++ libraries. A list of some standard C++ libraries is given in Appendix 4.

preprocessor

C++ has a **preprocessor** that handles some simple textual manipulation before the text of your program is given to the compiler. Some people will tell you that `include` directives are not processed by the compiler but are processed by a preprocessor. They're right, but the difference is more of a word game than anything that need concern you. On almost all compilers, the preprocessor is called automatically when you compile your program.

Technically speaking only part of the library definition is given in the header file. However, at this stage, that is not an important distinction, since using the `include`

directive with the header file for a library will (on almost all systems) cause C++ to automatically add the rest of the library definition.

NAMESPACES

A **namespace** is a collection of name definitions. One name, such as a function name, can be given different definitions in two namespaces. A program can then use one of these namespaces in one place and the other in another location. We will discuss namespaces in detail later in this book. For now, we only need to discuss the namespace `std`. All the standard libraries we will be using place their definitions in the `std` (standard) namespace. To use any of these definitions in your program, you must insert the following `using` directive:

namespace

```
using namespace std;
```

using namespace

Thus, a simple program that uses console I/O would begin

```
#include <iostream>
using namespace std;
```

If you want to make some, but not all, names in a namespace available to your program, there is a form of the `using` directive that makes just one name available. For example, if you only want to make the name `cin` from the `std` namespace available to your program, you could use the following `using` directive:

```
using std::cin;
```

Thus, if the only names from the `std` namespace that your program uses are `cin`, `count`, and `endl`, you might start your program with the following:

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
```

instead of

```
#include <iostream>
using namespace std;
```

Older C++ header files for libraries did not place their definitions in the `std` namespace, so if you look at older C++ code, you will probably see that the header file names are spelled slightly differently and the code does not contain any `using` directive. This is allowed for backward compatibility. However, you should use the newer library header files and the `std` namespace directive.

PITFALL

Problems with Library Names

The C++ language is currently in transition. A new standard has come out with, among other things, new names for libraries. If you are using a compiler that has not yet been revised to meet the new standard, then you will need to use different library names.
If the following does not work

```
#include <iostream>
```

use

```
#include <iostream.h>
```

Similarly, other library names are different for older compilers. Appendix 5 gives the correspondence between older and newer library names. This book always uses the new compiler names. If a library name does not work with your compiler, try the corresponding older library name. In all probability, either all the new library names will work or you will need to use all old library names. It is unlikely that only some of the library names have been made up to date on your system.
If you use the older library names (the ones that end in `.h`), you do *not* need the `using` directive

```
using namespace std;
```

CHAPTER SUMMARY

- C++ is case sensitive. For example, `count` and `COUNT` are two different identifiers.
- Use meaningful names for variables.
- Variables must be declared before they are used. Other than following this rule, a variable declaration may appear anyplace.
- Be sure that variables are initialized before the program attempts to use their value. This can be done when the variable is declared or with an assignment statement before the variable is first used.
- You can assign a value of an integer type, like `int`, to a variable of a floating-point type, like `double`, but not vice versa.
- Almost all number constants in a program should be given meaningful names that can be used in place of the numbers. This can be done by using the modifier `const` in a variable declaration.
- Use enough parentheses in arithmetic expressions to make the order of operations clear.
- The object `cout` is used for console output.

- A `\n` in a quoted string or an `endl` sent to console output starts a new line of output.
- The object `cerr` is used for error messages. In a typical environment, `cerr` behaves the same as `cout`.
- The object `cin` is used for console input.
- In order to use `cin`, `cout`, or `cerr`, you should place the following directives near the beginning of the file with your program:

```
#include <iostream>
using namespace std;
```

- There are two forms of comments in C++: Everything following `//` on the same line is a comment, and anything enclosed in `/*` and `*/` is a comment.
- Do not overcomment.

ANSWERS TO SELF-TEST EXERCISES

1.
```
int feet = 0, inches = 0;
int feet(0), inches(0);
```

2.
```
int count = 0;
double distance = 1.5;
int count(0);
double distance(1.5);
```

3. The actual output from a program such as this is dependent on the system and the history of the use of the system.

```
#include <iostream>
using namespace std;

int main( )
{
  int first, second, third, fourth, fifth;
  cout << first << " " << second << " " << third
       << " " << fourth << " " << fifth << "\n";
  return 0;
}
```

4.
```
3*x
3*x + y
(x + y)/7
(3*x + y)/(z + 2)
```
Note that `x + y/7` is not correct.

5. bcbc

6. `(1/3) * 3 is equal to 0`

   Since `1` and `3` are of type `int`, the `/` operator performs integer division, which discards the remainder, so the value of `1/3` is `0`, not `0.3333`…. This makes the value of the entire expression `0 * 3`, which of course is `0`.

7. ```
   #include <iostream>
   using namespace std;
   int main( )
   {
       int number1, number2;
       cout << "Enter two whole numbers: ";
       cin >> number1 >> number2;
       cout << number1 << " divided by " << number2
            << " equals " << (number1/number2) << "\n"
            << "with a remainder of " << (number1%number2)
            << "\n";
       return 0;
   }
   ```

8. a. `52.0`

   b. `9/5` has `int` value `1`. Since the numerator and denominator are both `int`, integer division is done; the fractional part is discarded. The programmer probably wanted floating-point division, which does not discard the part after the decimal point.

   c. `f = (9.0/5) * c + 32.0;`
   or
   `f = 1.8 * c + 32.0;`

9. ```
   cout << "The answer to the question of\n"
       << "Life, the Universe, and Everything is 42.\n";
   ```

10. ```
    cout << "Enter a whole number and press Return: ";
    cin >> theNumber;
    ```

11. ```
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(3);
    ```

12. ```
    #include <iostream>
    using namespace std;

    int main( )
    {
        cout << "Hello world\n";
        return 0;
    }
    ```
    ```

13. `cout << 'A' << endl << 'B' << '\t' << 'C';`

    Other answers are also correct. For example, the letters could be in double quotes instead of single quotes. Another possible answer is the following:

    ```
    cout << "A\nB\tC";
    ```

PROGRAMMING PROJECTS

Many of these Programming Projects can be solved using AW's CodeMate. To access these please go to: www.aw-bc.com/codemate.

1. A metric ton is 35,273.92 ounces. Write a program that will read the weight of a package of breakfast cereal in ounces and output the weight in metric tons as well as the number of boxes needed to yield one metric ton of cereal.

2. A government research lab has concluded that an artificial sweetener commonly used in diet soda will cause death in laboratory mice. A friend of yours is desperate to lose weight but cannot give up soda. Your friend wants to know how much diet soda it is possible to drink without dying as a result. Write a program to supply the answer. The input to the program is the amount of artificial sweetener needed to kill a mouse, the weight of the mouse, and the weight of the dieter. To ensure the safety of your friend, be sure the program requests the weight at which the dieter will stop dieting, rather than the dieter's current weight. Assume that diet soda contains one-tenth of 1% artificial sweetener. Use a variable declaration with the modifier `const` to give a name to this fraction. You may want to express the percentage as the `double` value `0.001`.

3. Workers at a particular company have won a 7.6% pay increase retroactive for six months. Write a program that takes an employee's previous annual salary as input and outputs the amount of retroactive pay due the employee, the new annual salary, and the new monthly salary. Use a variable declaration with the modifier `const` to express the pay increase.

4. Negotiating a consumer loan is not always straightforward. One form of loan is the discount installment loan, which works as follows. Suppose a loan has a face value of $1,000, the interest rate is 15%, and the duration is 18 months. The interest is computed by multiplying the face value of $1,000 by 0.15, yielding $150. That figure is then multiplied by the loan period of 1.5 years to yield $225 as the total interest owed. That amount is immediately deducted from the face value, leaving the consumer with only $775. Repayment is made in equal monthly installments based on the face value. So the monthly loan payment will be $1,000 divided by 18, which is $55.56. This method of calculation may not be too bad if the consumer needs $775 dollars, but the calculation is a bit more complicated if the consumer needs $1,000. Write a program that will take three inputs: the amount the consumer needs to receive, the interest rate, and the duration of the loan in months. The program should then calculate the face value required in order for the consumer to receive the amount needed. It should also calculate the monthly payment.

5. Write a program that determines whether a meeting room is in violation of fire law regulations regarding the maximum room capacity. The program will read in the maximum room capacity and the number of people to attend the meeting. If the number of people is less than or equal to the maximum room capacity, the program announces that it is legal to hold the meeting and tells how many additional people may legally attend. If the number of people exceeds the maximum room capacity, the program announces that the meeting cannot be held as planned due to fire regulations and tells how many people must be excluded in order to meet the fire regulations.

6. An employee is paid at a rate of $16.78 per hour for regular hours worked in a week. Any hours over that are paid at the overtime rate of one and one-half times that. From the worker's gross pay, 6% is withheld for Social Security tax, 14% is withheld for federal income tax, 5% is withheld for state income tax, and $10 per week is withheld for union dues. If the worker has three or more dependents, then an additional $35 is withheld to cover the extra cost of health insurance beyond what the employer pays. Write a program that will read in the number of hours worked in a week and the number of dependents as input and that will then output the worker's gross pay, each withholding amount, and the net take-home pay for the week.

7. One way to measure the amount of energy that is expended during exercise is to use metabolic equivalents (MET). Here are some METS for various activities:

   Running 6 MPH: 10 METS

   Basketball: 8 METS

   Sleeping: MET

   The number of calories burned per minute may be estimated using the formula

   Calories/Minute = 0.0175 × MET × Weight(Kg)

   Write a program that inputs a subject's weight in pounds, the number of METS for an activity, and the number of minutes spent on that activity, and then outputs an estimate for the total number of calories burned. One kilogram is equal to 2.2 pounds.

8. The Babylonian algorithm to compute the square root of a number ***n*** is as follows:

   1. Make a ***guess*** at the answer (you can pick n/2 as your initial guess).
   2. Compute ***r = n / guess.***
   3. Set ***guess = (guess + r) / 2.***
   4. Go back to step 2 for as many iterations as necessary. The more steps 2 and 3 are repeated, the closer ***guess*** will become to the square root of ***n***.

   Write a program that inputs an integer for ***n***, iterates through the Babylonian algorithm five times, and outputs the answer as a `double` to two decimal places. Your answer will be most accurate for small values of ***n***.

9. The video game machines at your local arcade output coupons depending on how well you play the game. You can redeem 10 coupons for a candy bar or 3 coupons for a gumball. You prefer candy bars to gumballs. Write a program that inputs the number of coupons you win and outputs how many candy bars and gumballs you can get if you spend all of your coupons on candy bars first and any remaining coupons on gumballs.

For additional online Programming Projects, click the CodeMate icons below.

1.10