C++ Basics

C++ Basics

Don't imagine you know what a computer terminal is. A computer terminal is not some clunky old television with a typewriter in front of it. It is an interface where the mind and the body can connect with the universe and move bits of it about.

DOUGLAS ADAMS, *MOSTLY HARMLESS*
(*the fifth volume in THE HITCHHIKER'S TRILOGY*)

Introduction

In this chapter we explain some additional sample C++ programs and present enough details of the C++ language to allow you to write simple C++ programs.

Prerequisites

In Chapter 1 we gave a brief description of one sample C++ program. (If you have not read the description of that program, you may find it helpful to do so before reading this chapter.)

2.1 Variables and Assignments

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*

Programs manipulate data such as numbers and letters. C++ and most other common programming languages use programming constructs known as *variables* to name and store data. Variables are at the very heart of a programming language like C++, so that is where we start our description of C++. We will use the program in Display 2.1 for our discussion and will explain all the items in that program. While the general idea of that program should be clear, some of the details are new and will require some explanation.

Variables

A C++ variable can hold a number or data of other types. For the moment, we will confine our attention to variables that hold only numbers. These variables are like

small blackboards on which the numbers can be written. Just as the numbers written on a blackboard can be changed, so too can the number held by a C++ variable be changed. Unlike a blackboard that might possibly contain no number at all, a C++ variable is guaranteed to have some value in it, if only a garbage number left in the computer's memory by some previously run program. The number or other type of data held in a variable is called its **value;** that is, the value of a variable is the item written on the figurative blackboard. In the program in Display 2.1, `number_of_bars`, `one_weight`, and `total_weight` are variables. For example, when this program is run with the input shown in the sample dialogue, `number_of_bars` has its value set equal to the number 11 with the statement

value of a variable

```
cin >> number_of_bars;
```

Later, the value of the variable `number_of_bars` is changed to 12 when a second copy of the same statement is executed. We will discuss exactly how this happens a little later in this chapter.

Of course, variables are not blackboards. In programming languages, variables are implemented as memory locations. The compiler assigns a memory location (of the kind discussed in Chapter 1) to each variable name in the program. The value of the variable, in a coded form consisting of zeros and ones, is kept in the memory location assigned to that variable. For example, the three variables in the program shown in Display 2.1 might be assigned the memory locations with addresses 1001, 1003, and 1007. The exact numbers will depend on your computer, your compiler, and a number of other factors. We do not know, or even care, what addresses the compiler will choose for the variables in our program. We can think as though the memory locations were actually labeled with the variable names.

variables are memory locations

Cannot Get Programs to Run?

If you cannot get your C++ programs to compile and run, read the Pitfall section of Chapter 1 entitled "Getting Your Program to Run." This section has tips for dealing with variations in C++ compilers and C++ environments.

Names: Identifiers

The first thing you might notice about the names of the variables in our sample programs is that they are longer than the names normally used for variables in mathematics classes. To make your program easy to understand, you should always use meaningful names for variables. The name of a variable (or other item you might define in a program) is called an **identifier.** An 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 Big_Bonus
```

Display 2.1 A C++ Program (*part 1 of 2*)


```
#include <iostream>
using namespace std;
int main( )
{
    int number_of_bars;
    double one_weight, total_weight;

    cout << "Enter the number of candy bars in a package\n";
    cout << "and the weight in ounces of one candy bar.\n";
    cout << "Then press return.\n";
    cin >> number_of_bars;
    cin >> one_weight;

    total_weight = one_weight * number_of_bars;

    cout << number_of_bars << " candy bars\n";
    cout << one_weight << " ounces each\n";
    cout << "Total weight is " << total_weight << " ounces.\n";

    cout << "Try another brand.\n";
    cout << "Enter the number of candy bars in a package\n";
    cout << "and the weight in ounces of one candy bar.\n";
    cout << "Then press return.\n";
    cin >> number_of_bars;
    cin >> one_weight;

    total_weight = one_weight * number_of_bars;

    cout << number_of_bars << " candy bars\n";
    cout << one_weight << " ounces each\n";
    cout << "Total weight is " << total_weight << " ounces.\n";

    cout << "Perhaps an apple would be healthier.\n";

    return 0;
}
```

Display 2.1 A C++ Program (*part 2 of 2*)

Sample Dialogue

```
Enter the number of candy bars in a package
and the weight in ounces of one candy bar.
Then press return.
11 2.1
11 candy bars
2.1 ounces each
Total weight is 23.1 ounces.
Try another brand.
Enter the number of candy bars in a package
and the weight in ounces of one candy bar.
Then press return.
12 1.8
12 candy bars
1.8 ounces each
Total weight is 21.6 ounces.
Perhaps an apple would be healthier.
```

All of the previously mentioned names are legal and would be accepted by the compiler, but the first five are poor choices for identifiers, since 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.

uppercase and lowercase

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:

```
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 often spelled with all lowercase letters. The predefined identifiers, such as `main`, `cin`, `cout`, and so forth, must be spelled in all lowercase letters. We will see uses for identifiers spelled with uppercase letters later in this chapter.

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

Identifiers

Identifiers are used as names for variables and other items in a C++ program. An identifier must start with either a letter or the underscore symbol, and the remaining characters must all be letters, digits, or the underscore symbol.

keywords

There is a special class of identifiers, called **keywords** or **reserved words,** that have a predefined meaning in C++ and that you cannot use as names for variables or anything else. In this book keywords are written in a different type font like so: *`int`*, *`double`*. (And now you know why those words were written in a funny way.) A complete list of keywords is given in Appendix 1.

You may wonder why the other words that we defined as part of the C++ language are not on the list of keywords. What about words like `cin` and `cout`? The answer is that you are allowed to redefine these words, although it would be confusing to do so. These predefined words are not keywords; however, they are defined in libraries required by the C++ language standard. We will discuss libraries later in this book. For now, you need not worry about libraries. 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.

Variable Declarations

Every variable in a C++ program must be *declared*. 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 two declarations from the program in Display 2.1 declare the three variables used in that program:

```
int number_of_bars;
double one_weight, total_weight;
```

When there is more than one variable in a declaration, the variables are separated by commas. Also, note that each declaration ends with a semicolon.

The word *int* in the first of these two declarations is an abbreviation of the word *integer*. (But in a C++ program you must use the abbreviated form *int*. Do not write out the entire word *integer*.) This line declares the identifier number_of_bars to be a variable of *type int*. This means that the value of number_of_bars must be a whole number, such as 1, 2, –1, 0, 37, or –288.

The word *double* in the second of these two lines declares the two identifiers one_weight and total_weight to be variables of type *double*. A variable of type *double* can hold numbers with a fractional part, such as 1.75 or –0.55. The kind of data that is held in a variable is called its **type** and the name for the type, such as *int* or *double*, is called a **type name.**

type

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, number_of_dragons, number_of_trolls;
double distance;
```

where to place variable declarations

Every variable in a C++ program must be declared before the variable can be used. There are two natural places to declare a variable: either just before it is used or at the start of the main part of your program right after the lines

```
int main( )
{
```

Do whatever makes your program clearer.

Variable declarations provide information the compiler needs in order to implement the variables. Recall that the compiler implements variables as memory locations and that the value of a variable is stored in the memory location assigned to that variable. The value is coded as a string of zeros and ones. Different types of variables require different sizes of memory locations and different methods for coding their values as a string of zeros and ones. The computer uses one code to encode integers as a string of zeros and ones. It uses a different code to encode numbers that have a fractional part. It uses yet another code to encode letters as strings of zeros and ones. The variable declaration tells the compiler—and, ultimately, the computer—what size memory location to use for the variable and which code to use when representing the variable's value as a string of zeros and ones.

Syntax

The **syntax** for a programming language (or any other kind of language) is the set of grammar rules for that language. For example, when we talk about the syntax for a variable declaration (as in the box labeled "Variable Declarations"), we are talking about the rules for writing down a well-formed variable declaration. If you follow all the syntax rules for C++, then the compiler will accept your program. Of course, this only guarantees that what you write is legal. It guarantees that your program will do something, but it does not guarantee that your program will do what you want it to do.

Assignment Statements

The most direct way to change the value of a variable is to use an *assignment statement*. An **assignment statement** is an order to the computer saying, "set the value of this variable to what I have written down." The following line from the program in Display 2.1 is an example of an assignment statement:

```
total_weight = one_weight * number_of_bars;
```

This assignment statement tells the computer to set the value of `total_weight` equal to the number in the variable `one_weight` multiplied by the number in `number_of_bars`. (As we noted in Chapter 1, * is the sign used for multiplication in C++.)

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, and arithmetic operators such as * and +. 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. A few more examples may help to clarify the way these assignment statements work.

You may use any arithmetic operator in place of the multiplication sign. The following, for example, is also a valid assignment statement:

```
total_weight = one_weight + number_of_bars;
```

This statement is just like the assignment statements in our sample program, except that it performs addition rather than multiplication. This statement changes the value of `total_weight` to the sum of the values of `one_weight` and `number_of_bars`. Of course, if you made this change in the program in Display 2.1, the program would give incorrect output, but it would still run.

In an assignment statement, the expression on the right-hand side of the equal sign can simply be another variable. The statement

```
total_weight = one_weight;
```

changes the value of the variable `total_weight` so that it is the same as that of the variable `one_weight`. If you were to use this in the program in Display 2.1, it would give out incorrectly low values for the total weight of a package (assuming there is more than one candy bar in a package), but it might make sense in some other program.

As another example, the following assignment statement changes the value of `number_of_bars` to 37:

```
number_of_bars = 37;
```

constant

A number, like the 37 in this example, is called a **constant,** because unlike a variable, its value cannot change.

same variable on both sides of =

Since variables can change value over time and since the assignment operator is one vehicle for changing their values, there is an element of time involved in the meaning of an assignment statement. First, the expression on the right-hand side of the equal sign is evaluated. After that, the value of the variable on the left side of the equal sign is changed to the value that was obtained from that expression. This means that a variable can meaningfully occur on both sides of an assignment operator. For example, consider the assignment statement

```
number_of_bars = number_of_bars + 3;
```

This assignment statement may look strange at first. If you read it as an English sentence, it seems to say "the `number_of_bars` is equal to the `number_of_bars` plus three." It may seem to say that, but what it really says is, "Make the *new* value of `number_of_bars` equal to the *old* value of `number_of_bars` plus three." The equal sign in C++ is not used the same way that it is used in English or in simple mathematics.

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;
```

PITFALL Uninitialized Variables

A variable has no meaningful value until a program gives it one. For example, if the variable `minimum_number` 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:

```
desired_number = minimum_number + 10;
```

This is because `minimum_number` has no meaningful value, so the entire expression on the right-hand side of the equal sign has no meaningful value. A variable like `minimum_number` that has not been given a value is said to be **uninitialized.** This situation is, in fact, worse than it would be if `minimum_number` had no value at all. An uninitialized variable, like `minimum_number`, 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. Thus if the program is run twice, an uninitialized variable may receive a different value each time the program is run. Whenever a program gives different output on *exactly* the same input data and without *any* changes in the program itself, you should suspect an uninitialized variable.

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 minimum_number = 3;
```

This both declares `minimum_number` to be a variable of type `int` and sets the value of the variable `minimum_number` 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. However, a simple constant is what is most often used. You can initialize some, all, or none of the variables in a declaration that lists more than one variable. For example, the following declares three variables and initializes two of them:

```
double rate = 0.07, time, balance = 0.0;
```

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(0.0);
```

Whether you initialize a variable when it is declared or at some later point in the program depends on the circumstances. Do whatever makes your program the easiest to understand.

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, fudge_factor = 2;
double distance = 999.99;
```

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), fudge_factor(2);
double distance(999.99);
```

Programming 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 the following:

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

with the more suggestive:

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

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

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. Use 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`.

3 Give a C++ statement that will change the value of the variable `sum` to the sum of the values in the variables `n1` and `n2`. The variables are all of type *`int`*.

4 Give a C++ statement that will increase the value of the variable `length` by 8.3. The variable `length` is of type *`double`*.

5 Give a C++ statement that will change the value of the variable `product` to its old value multiplied by the value of the variable `n`. The variables are all of type *`int`*.

6 Write a program that contains statements that output the value of five or six variables that have been declared, but not initialized. Compile and run the program. What is the output? Explain.

7 Give good variable names for each of the following:

a. A variable to hold the speed of an automobile
b. A variable to hold the pay rate for an hourly employee
c. A variable to hold the highest score in an exam

2.2 Input and Output

Garbage in means garbage out.

PROGRAMMERS' SAYING

input stream

There are several different ways that a C++ program can perform input and output. We will describe what are called *streams*. An **input stream** is simply the stream of input that is being fed into the computer for the program to use. The word *stream* suggests that the program processes the input in the same way no matter where the input comes from. The intuition for the word *stream* is that the program sees only the stream of input and not the source of the stream, like a mountain stream whose water flows past you but whose source is unknown to you. In this section we will assume that the input comes from the keyboard. In Chapter 5 we will discuss how a program can read its input from a file; as you will see there, you can use the same kinds of input statements to read input from a file as those that you use for reading input from the keyboard. Similarly, an **output stream** is the stream of output generated by the program. In this section we will assume the output is going to a terminal screen; in Chapter 5 we will discuss output that goes to a file.

output stream

Output Using cout

The values of variables as well as strings of text may be output to the screen using `cout`. There may be any combination of variables and strings to be output. For example, consider the following line from the program in Display 2.1:

```
cout << number_of_bars << " candy bars\n";
```

This statement tells the computer to output two items: the value of the variable `number_of_bars` and the quoted string `" candy bars\n"`. Notice that you do not need a separate copy of the word `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 above single `cout` statement is equivalent to the following two `cout` statements:

```
cout << number_of_bars;
cout << " candy bars\n";
```

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);
```

The parentheses around arithmetic expressions, like `price + tax`, are required by some compilers, so it is best to include them.

The symbol < is the same as the "less than" symbol. 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.

insertion operator

Whenever you have two `cout` statements in a row, you can combine them into a single long `cout` statement. For example, consider the following lines from Display 2.1:

```
cout << number_of_bars << " candy bars\n";
cout << one_weight << " ounces each\n";
```

These two statements can be rewritten as the single statement shown below, and the program will perform exactly the same:

```
cout << number_of_bars << " candy bars\n" << one_weight
<< " ounces each\n";
```

If you want to keep your program lines from running off the screen, you will have to place such a long `cout` statement on two or more lines. A better way to write the above long `cout` statement is:

```
cout << number_of_bars << " candy bars\n"
     << one_weight << " ounces each\n";
```

You should not break a quoted string across two lines, but otherwise you can start a new line anywhere you can insert a space. Any reasonable pattern of spaces and line

breaks will be acceptable to the computer, but the above example and the sample programs are good models to follow. A good policy is to use one `cout` for each group of output that is intuitively considered a unit. Notice that there is just one semicolon for each `cout`, even if the `cout` statement spans several lines.

Pay particular attention to the quoted strings that are output in the program in Display 2.1. Notice that the strings must be included in double quotes. The double quote symbol used is a single key on your keyboard; do not type two single quotes. Also, notice that the same double quote symbol is used at each end of the string; there are not separate left and right quote symbols.

spaces in output

Also, notice the spaces inside the quotes. The computer does not insert any extra space before or after the items output by a `cout` statement. That is why the quoted strings in the samples often start and/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 << first_number << " " << second_number;
```

new lines in output

As we noted in Chapter 1, `\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 of the quotes. In C++, going to the next line is considered to be a special character (special symbol) 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 single character that is called the **new-line character.**

new-line character

Include Directives and Namespaces

We have started all of our programs with the following two lines:

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

These two lines make the library `iostream` available. This is the library that includes, among other things, the definitions of `cin` and `cout`. So if your program uses either `cin` or `cout`, you should have these two lines at the start of the file that contains your program.

include directive

The following line is known as an **include directive.** It "includes" the library `iostream` in your program so that you have `cin` and `cout` available:

```
#include <iostream>
```

The operators `cin` and `cout` are defined in a file named `iostream` and the above `include` directive is equivalent to copying that named file into your program. The second line is a bit more complicated to explain.

namespace

C++ divides names into **namespaces.** A namespace is a collection of names, such as the names `cin` and `cout`. A statement that specifies a namespace in the way illustrated by the following is called a *`using`* **directive.**

```
using namespace std;
```

This particular *`using`* directive says that your program is using the `std` ("standard") namespace. This means that the names you use will have the meaning defined for them in the `std` namespace. In this case, the important thing is that when names such as `cin` and `cout` were defined in `iostream`, their definitions said they were in the `std` namespace. So to use names like `cin` and `cout`, you need to tell the compiler you are *`using namespace`* `std;`.

That is all you need to know (for now) about namespaces, but a brief clarifying remark will remove some of the mystery that might surround the use of *`namespace`*. The reason that C++ has namespaces at all is because there are so many things to name. As a result, sometimes two or more items receive the same name; that is, a single name can get two different definitions. To eliminate these ambiguities, C++ divides items into collections so that no two items in the same collection (the same namespace) have the same name.

Note that a namespace is not simply a collection of names. It is a body of C++ code that specifies the meaning of some names, such as some definitions and/or declarations. The function of namespaces is to divide all the C++ name specifications into collections (called *namespaces*) such that each name in a namespace has only one specification (one "definition") in that namespace. A namespace divides up the names, but it takes a lot of C++ code along with the names.

What if you want to use two items in two different namespaces, such that both items have the same name? It can be done and is not too complicated, but that is a topic for later in the book. For now, we do not need to do this.

alternative form

Some versions of C++ use the following, older form of the `include` directive (without any *`using namespace`*):

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

If your programs do not compile or do not run with

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

then try using the following line instead of the previous two lines:

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

If your program requires `iostream.h` instead of `iostream`, then you have an old C++ compiler and should obtain a more recent compiler.

Escape Sequences

escape sequence

The \ preceding a character tells the compiler that the character following the \ 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 " 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, or \" means a real quote, not a string constant end.

A stray \, say \z, in a string constant will on one compiler simply give back a z; on another it will produce an error. The ANSI Standard provides that the 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. We list a few here.

new-line	`\n`
horizontal tab	`\t`
alert	`\a`
backslash	`\\`
double quote	`\"`

If you wish to insert a blank line in the output, you can output the new-line 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;
```

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

deciding between `\n` and `endl`

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;
```

Programming TIP

End Each Program with a \n or endl

It is a good idea to output a new-line 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 action in your program. This serves two purposes. Some compilers will not output the last line of your program unless you include a new-line instruction at the end. On other systems, your program may work fine without this final new-line 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 new-line 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 we will explain in section 2.3):

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

But it is extremely unlikely that the output will be the following, 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` statement that follows these three statements will output values of type *double* 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 5. 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 5 for all subsequent values of type *double* 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 that allows for digits after the decimal point) will be output in ordinary everyday notation with 2 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.

Input Using `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 2.1, the variables `number_of_bars` and `one_weight` were filled by the following `cin` statements (shown along with the `cout` statements that tell the user what to do):

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press return.\n";
cin >> number_of_bars;
cin >> one_weight;
```

You can list more than one variable in a single `cin` statement. So the above lines could be rewritten to the following:

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press return.\n";
cin >> number_of_bars >> one_weight;
```

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

```
cin >> number_of_bars
    >> one_weight;
```

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. If, for instance, you want to enter the two numbers **12** and **5** and instead you enter the numbers without any space between them, then the computer will think you have entered the single number **125.** 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.

`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 >> time_to_go
    >> points_needed;
```

Designing Input and Output

I/O

Input and output, or as it is often called **I/O,** is the part of the program that the user sees, so the user will not be happy with a program unless the program has well-designed I/O.

prompt lines

When the computer executes a `cin` statement, it expects some data to be typed in at the keyboard. If none is typed in, the computer simply waits for it. The program must tell the user when to type in a number (or other data item). The computer will not automatically ask the user to enter data. That is why the sample programs contain output statements like the following:

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press return.\n";
```

These output statements prompt the user to enter the input. Your programs should always prompt for input.

echoing the input

When entering input from a terminal, the input appears on the screen as it is typed in. Nonetheless, the program should always write out the input values some time before it ends. This is called **echoing the input,** and it serves as a check to see that the input was read in correctly. Just because the input looks good on the screen when it is typed in does not mean that it was read correctly by the computer. There could be an unnoticed typing mistake or other problem. Echoing input serves as a test of the integrity of the input data.

Programming 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 >> cost_per_person;
```

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

8 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.
```

9 Give an input statement that will fill the variable `the_number` (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.

10 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?

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

12 Write a complete C++ program that reads in two whole numbers and outputs their sum. Be sure to prompt for input, echo input, and label all output.

13 Give an output statement that produces the new-line character and a tab character.

14 Write a short program that declares and initializes *double* variables `one`, `two`, `three`, `four`, and `five` to the values `1.000`, `1.414`, `1.732`, `2.000`, and `2.236`, respectively. Then write output statements to generate the following legend and table. Use the tab escape sequence `\t` to line up the columns. If you are unfamiliar with the tab character, you should experiment with it while doing this exercise. A tab works like a mechanical stop on a typewriter. A tab causes output to begin in a next column, usually a multiple of eight spaces away. Many editors and most word processors will have adjustable tab stops. Our output does not.

The output should be:

```
N    Square Root
1    1.000
2    1.414
3    1.732
4    2.000
5    2.236
```

2.3 Data Types and Expressions

They'll never be happy together. He's not her type.

OVERHEARD AT A COCKTAIL PARTY

The Types *int* and *double*

Conceptually the numbers 2 and 2.0 are the same number. But C++ considers them to be of different types. The whole number `2` is of type *`int`*; the number `2.0` is of type *`double`*, because it contains a fraction part (even though the fraction is 0). Once again, the mathematics of computer programming is a bit different from what you may have learned in mathematics classes. Something about the practicalities of computers makes a computer's numbers differ from the abstract definitions of these numbers. The whole numbers in C++ behave as you would expect them to. The type *`int`* holds no surprises. But values of type *`double`* are more troublesome. Because it can store only a limited number of significant digits, the computer stores numbers of type *`double`* as approximate values. Numbers of type *`int`* are stored as exact values. The precision with which *`double`* values are stored varies from one computer to another, but you can expect them to be stored with 14 or more digits of accuracy. For most applications this is likely to be sufficient, though subtle problems can occur even in simple cases. Thus, if you know that the values in some variable will always be whole numbers in the range allowed by your computer, it is best to declare the variable to be of type *`int`*.

Number constants of type *`double`* are written differently from those of type *`int`*. Constants of type *`int`* 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. There is, however, one thing that constants of type *`double`* and constants of type *`int`* have in common: No number in C++ may contain a comma.

e notation

The more complicated notation for constants of type *`double`* is frequently 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`,

What is doubled?

Why is the type for numbers with a fraction 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*. These types are described in the subsection entitled "Other Number Types." However, we will have no occasion to use the types *float* and *long double* in this book.

is best expressed in C++ by the constants and 5.89e–6. The e stands for *exponent* and means "multiply by 10 to the power that follows."

This e notation is used because keyboards normally have no way to write exponents as superscripts. 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.

Since computers have size limitations on their memory, numbers are typically stored in a limited number of bytes (that is, a limited amount of storage). Hence, there is a limit to how large the magnitude of a number can be, and this limit is different for different number types. The largest allowable number of type *double* is always much larger than the largest allowable number of type *int*. Just about any implementation of C++ will allow values of type *int* as large as 32767 and values of type *double* up to about 10^{308}.

Other Number Types

C++ has other numeric types besides *int* and *double*. Some are described in Display 2.2. The various number types allow for different size numbers and for more or less precision (that is, more or fewer digits after the decimal point). In Display 2.2, the values given for memory used, size range, and precision are only

one sample set of values, intended to give you a general feel for how the types differ. The values vary from one system to another, and may be different on your system.

Although some of these other numeric types are spelled as two words, you declare variables of these other types just as you declare variables of types `int` and `double`. For example, the following declares one variable of type `long double`:

`long double`

```
long double big_number;
```

The type names `long` and `long int` are two names for the same type. Thus, the following two declarations are equivalent:

`long`

```
long big_total;
```

and the equivalent

```
long int big_total;
```

Of course, in any one program, you should use only one of the above two declarations for the variable `big_total`, but it does not matter which one you use. Also, remember that the type name `long` by itself means the same thing as `long int`, not the same thing as `long double`.

The types for whole numbers, such an `int` and similar types, are called **integer types.** The type for numbers with a decimal point—such as the type `double` and similar types—are called **floating-point types.** They are called *floating-point* because when the computer stores a number written in the usual way, like `392.123`, it first converts the number to something like `e` notation, in this case something like `3.92123e2`. When the computer performs this conversion, the decimal point *floats* (that is, moves) to a new position.

integer types

floating-point types

You should be aware of the fact that there are other numeric types in C++. However, in this book, we will use only the types `int`, `double`, and occasionally `long`. For most simple applications, you should not need any types except `int` and `double`. However, if you are writing a program that uses very large whole numbers, then you might need to use the type `long`.

Display 2.2 Some Number Types (*part 1 of 2*)

Syntax Type Name	Syntax Memory Used	Syntax Size Range	Syntax 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)

Display 2.2 Some Number Types (*part 2 of 2*)

`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

These 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.

The Type *char*

We do not want to give you the impression that computers and C++ are used only for numeric calculations, so we will introduce one nonnumeric type now, though eventually we will see other more complicated nonnumeric types. Values of the type *char*, which is short for *character*, are single symbols such as a letter, digit, or punctuation mark. Values of this type are frequently called *characters* in books and in conversation, but in a C++ program this type must always be spelled in the abbreviated fashion *char*. For example, the variables `symbol` and `letter` of type *char* are declared as follows:

```
char symbol, letter;
```

A variable of type *char* can hold any single character on the keyboard. So, for example, the variable `symbol` could hold an `'A'` or a `'+'` or an `'a'`. Note that uppercase and lowercase versions of a letter are considered different characters.

strings and characters

There is a type for strings of more than one character, but we will not introduce that type for a while, although you have seen, and even used, values that are strings.

The strings in double quotes that are output using `cout` are string values. For example, the following, which occurs in the program in Display 2.1, is a string:

```
"Enter the number of candy bars in a package\n"
```

quotes

Be sure to notice that string constants are placed inside of double quotes, while constants of type *char* are placed inside of 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 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.

The use of the type *char* is illustrated in the program shown in Display 2.3. Notice that the user types a space between the first and second initials. Yet the program skips over the blank and reads the letter **B** as the second input character. When you use `cin` to read input into a variable of type *char*, the computer skips over all blanks and line breaks until it gets to the first nonblank character and reads that nonblank character into the variable. It makes no difference whether there are blanks in the input or not. The program in Display 2.3 will give the same output whether the user types in a blank between initials, as shown in the sample dialogue, or the user types in the two initials without a blank, like so:

```
JB
```

The Type *bool*

The final type we discuss here is the type *bool*. This type was recently added to the C++ language by the ISO/ANSI (International Standards Organization/American National Standards Organization) committee. Expressions of type *bool* are called *Boolean* after the English mathematician George Boole (1815–1864) who formulated rules for mathematical logic.

Boolean expressions evaluate to one of the two values, *true* or *false*. Boolean expressions are used in branching and looping statements that we study in Section 2.4. We will say more about Boolean expressions and the type *bool* in that section.

Type Compatibilities

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 int_variable;
int_variable = 2.99;
```

Display 2.3 The type *char*


```
#include <iostream>
using namespace std;
int main( )
{
    char symbol1, symbol2, symbol3;

    cout << "Enter two initials, without any periods:\n";
    cin >> symbol1 >> symbol2;

    cout << "The two initials are:\n";
    cout << symbol1 << symbol2 << endl;

    cout << "Once more with a space:\n";
    symbol3 = ' ';
    cout << symbol1 << symbol3 << symbol2 << endl;

    cout << "That's all.";

    return 0;
}
```

Sample Dialogue

```
Enter two initials, without any periods:
J B
The two initials are:
JB
Once more with a space:
J B
That's all.
```

The problem is a type mismatch. The constant `2.99` is of type *`double`* and the variable `int_variable` 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. But even if the compiler does allow you to use the above assignment, it will probably give `int_variable` 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`*.

The same problem arises if you use a variable of type *double* instead of the constant 2.99. Most compilers will also object to the following:

```
int int_variable;
double double_variable;
double_variable = 2.00;
int_variable = double_variable;
```

The fact that the value 2.00 "comes out even" makes no difference. The value 2.00 is of type *double*, not of type *int*. As you will see shortly, you can replace 2.00 with 2 in the above assignment to the variable double_variable, but even that is not enough to make the above acceptable. The variables int_variable and double_variable are of different types, and that is the cause of the problem.

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. For example, if your compiler lets you assign 2.99 to a variable of type *int*, the variable will receive the value 2, rather than 2.99, which can be confusing since the program seems to say the value will be 2.99.

assigning *int* values to *double* variables

There are some special cases where it is permitted to assign a value of one type to a variable of another type. It is acceptable to assign a value of type *int* to a variable of type *double*. For example, the following is both legal and acceptable style:

```
double double_variable;
double_variable = 2;
```

The above will set the value of the variable named double_variable 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 the 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* and so 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, since the value must be changed to a value of the appropriate type and that value may not be what you would expect.

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 and you should not use these features. For completeness and to help you read other people's code, we do give 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.

Arithmetic Operators and Expressions

In a C++ program, you can combine variables and/or numbers using the arithmetic operators + for addition, – for subtraction, * for multiplication, and / for division. For example, the following assignment statement, which appears in the program in Display 2.1, uses the * operator to multiply the numbers in two variables. (The result is then placed in the variable on the left-hand side of the equal sign.)

```
total_weight = one_weight * number_of_bars;
```

mixing types

All of 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 depends 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 is of type double, then the result is of type double. For example, if the variables base_amount and increase are of type int, then the number produced by the following expression is of type int:

```
base_amount + increase
```

However, if one or both of the two variables is 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 /.

division

The type of the result can be more significant than you might suspect. For example, 7.0/2 has one operand of type double, namely 7.0. Hence, the result is the type double number 3.5. However, 7/2 has two operands of type int and so it yields the type int result 3. Even if the result "comes out even," there is a difference. For example, 6.0/2 has one operand of type double, namely 6.0. Hence, the result is the type double number 3.0, which is only an approximate quantity. However, 6/2 has two operands of type int; so it yields the result 3, which is of type int and so is an exact quantity. The division operator is the operator that is affected most severely by the type of its arguments.

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

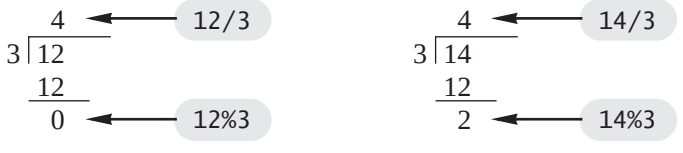
```
cout << "17 divided by 5 is " << (17/5) << endl;
cout << "with a remainder of " << (17%5) << endl;
```

yield the following output:

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

Display 2.4 illustrates how / and % work with values of type `int`.

Display 2.4 Integer Division



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.

spacing

Any reasonable spacing will do in arithmetic expressions. You can insert spaces before and after operations and parentheses, or you can omit them. Do whatever produces a result that is easy to read.

parentheses

You can specify the order of operations by inserting parentheses, as illustrated in the following two expressions:

```
(x + y) * z
x + (y * z)
```

To evaluate the first expression, the computer first adds x and y and then multiplies the result by z. To evaluate the second expression, it multiplies y and z and then adds the result to x. Although you may be used to using mathematical formulas that contain square brackets and various other forms of parentheses, that is not allowed in C++. C++ allows only one kind of parentheses in arithmetic expressions. The other varieties are reserved for other purposes.

precedence rules

If you omit parentheses, the computer will follow rules called **precedence rules** that determine the order in which the operators, such as + and *, 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 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 are given in Appendix 2.

Display 2.5 shows some examples of common kinds of arithmetic expressions and how they are expressed in C++.

Display 2.5 Arithmetic Expressions

Mathematical Formula	C++ Expression
$b^2 - 4ac$	`b*b - 4*a*c`
$x(y + z)$	`x*(y + z)`
$\frac{1}{x^2 + x + 3}$	`1/(x*x + x + 3)`
$\frac{a + b}{c - d}$	`(a + b)/(c - d)`

PITFALL Whole Numbers in Division

When you use the division operator / on two whole numbers, the result is a whole number. 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 your 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:

```
total_price = 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:

```
total_price = 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`, which you think you are getting). So the value assigned to `total_cost` is `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 `total_price` is *int* or *double*; the damage is done before the value is assigned to `total_price`.

SELF-TEST EXERCISES

15 Convert each of the following mathematical formulas to C++ expression:

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

16 What is the output of the following program lines, when 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';
```

17 What is the output of the following program lines (when 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;
```

18 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 `%`.

19 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.

More Assignment Statements

There is a shorthand notation 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 Op = *Expression*

which is equivalent to

Variable = *Variable Op* (*Expression*)

Op is an operator such as +, –, or *. The *Expression* can be another variable, a constant, or a more complicated arithmetic expression. Below are examples:

Example:	Equivalent to:
`count += 2;`	`count = count + 2;`
`total –= discount;`	`total = total – discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rush_factor;`	`time = time/rush_factor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

2.4 Simple Flow of Control

"If you think we're wax-works," he said, "you ought to pay, you know. Wax-works weren't made to be looked at for nothing. Nohow!"
"Contrariwise," added the one marked "DEE," "if you think we're alive, you ought to speak."

LEWIS CARROLL, *THROUGH THE LOOKING-GLASS*

The programs you have seen thus far each consist of a simple list of statements to be executed in the order given. However, to write more sophisticated programs, you will also need some way to vary the order in which statements are executed. The order in which statements are executed is often referred to as **flow of control.** In this section we will present two simple ways to add some flow of control to your programs. We will discuss a branching mechanism that lets your program choose between two alternative actions, choosing one or the other depending on the values of variables. We will also present a looping mechanism that lets your program repeat an action a number of times.

flow of control

A Simple Branching Mechanism

Sometimes it is necessary to have a program choose one of two alternatives, depending on the input. For example, suppose you want to design a program to compute a week's salary for an hourly employee. Assume the firm pays an overtime

rate of one-and-one-half times the regular rate for all hours after the first 40 hours worked. As long as the employee works 40 or more hours, the pay is then equal to

```
rate*40 + 1.5*rate*(hours - 40)
```

However, if there is a possibility that the employee will work less than 40 hours, this formula will unfairly pay a negative amount of overtime. (To see this, just substitute 10 for `hours`, 1 for `rate`, and do the arithmetic. The poor employee will get a negative paycheck.) The correct pay formula for an employee who works less than forty hours is simply:

```
rate*hours
```

If both more than 40 hours and less than 40 hours of work are possible, then the program will need to choose between the two formulas. In order to compute the employee's pay, the program action should be

> Decide whether or not `(hours > 40)` is true.
>
> If it is, do the following assignment statement:
> ```
> gross_pay = rate*40 + 1.5*rate*(hours - 40);
> ```
> If it is not, do the following:
> ```
> gross_pay = rate*hours;
> ```

There is a C++ statement that does exactly this kind of branching action. The *`if-else`* **statement** chooses between two alternative actions. For example, the wage calculation we have been discussing can be accomplished with the following C++ statement:

```
if (hours > 40)
    gross_pay = rate*40 + 1.5*rate*(hours - 40);
else
    gross_pay = rate*hours;
```

A complete program that uses this statement is given in Display 2.6.

Two forms of an *`if-else`* statement are described in Display 2.7. The first is the simple form of an *`if-else`* statement; the second form will be discussed in the subsection entitled "Compound Statements." In the first form shown, the two statements may be any executable statements. The *Boolean_Expression* is a test that can be checked to see if it is true or false, that is, to see if it is satisfied or not. For example, the *Boolean_Expression* in the above *`if-else`* statement is

```
hours > 40
```

Display 2.6 An *if-else* Statement


```
#include <iostream>
using namespace std;
int main( )
{
    int hours;
    double gross_pay, rate;

    cout << "Enter the hourly rate of pay: $";
    cin >> rate;
    cout << "Enter the number of hours worked,\n"
         << "rounded to a whole number of hours: ";
    cin >> hours;

    if (hours > 40)
        gross_pay = rate*40 + 1.5*rate*(hours - 40);
    else
        gross_pay = rate*hours;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Hours = " << hours << endl;
    cout << "Hourly pay rate = $" << rate << endl;
    cout << "Gross pay = $" << gross_pay << endl;

    return 0;
}
```

Sample Dialogue 1

```
Enter the hourly rate of pay: $20.00
Enter the number of hours worked,
rounded to a whole number of hours: 30
Hours = 30
Hourly pay rate = $20.00
Gross pay = $600.00
```

Sample Dialogue 2

```
Enter the hourly rate of pay: $10.00
Enter the number of hours worked,
rounded to a whole number of hours: 41
Hours = 41
Hourly pay rate = $10.00
Gross pay = $415.00
```

Display 2.7 Syntax for an *if-else* Statement

A Single Statement for Each Alternative:

```
if (Boolean_Expression)
    Yes_Statement
else
    No_Statement
```

A Sequence of Statements for Each Alternative:

```
if (Boolean_Expression)
{
    Yes_Statement_1
    Yes_Statement_2
      ...
    Yes_Statement_Last
}
else
{
    No_Statement_1
    No_Statement_2
      ...
    No_Statement_Last
}
```

When the program reaches the *if-else* statement, exactly one of the two embedded statements is executed. If the *Boolean_Expression* is true (that is, if it is satisfied), then the *Yes_Statement* is executed; if the *Boolean_Expression* is false (that is, if it is not satisfied), then the *No_Statement* is executed. Notice that the *Boolean_Expression* must be enclosed in parentheses. (This is required by the syntax rules for `if-else` statements in C++.) Also notice that an *if-else* statement has two smaller statements embedded in it.

Boolean expression

A **Boolean expression** is any expression that is either true or false. An *if-else* statement always contains a *Boolean_Expression*. The simplest form for a *Boolean_Expression* consists of two expressions, such as numbers or variables, that are compared with one of the comparison operators shown in Display 2.8. Notice that some of the operators are spelled with two symbols, for example, ==, !=, <=, >=. Be sure to notice that you use a double equal == for the equal sign, and you use the two symbols != for not equal. Such two-symbol operators should not have any space between the two symbols. The part of the compiler that separates the characters into C++ names and symbols will see the !=, for example, and tell the rest of the compiler

Display 2.8 Comparison Operators

Math Symbol	English	C++ Notation	C++ Sample	Math Equivalent
=	equal to	==	`x + 7 == 2*y`	$x + 7 = 2y$
≠	not equal to	!=	`ans != 'n'`	ans ≠ 'n'
<	less than	<	`count < m + 3`	count < m + 3
≤	less than or equal to	<=	`time <= limit`	time ≤ limit
>	greater than	>	`time > limit`	time > limit
≥	greater than or equal to	>=	`age >= 21`	age ≥ 21

that the programmer meant to test for INEQUALITY. When an *if-else* statement is executed, the two expressions being compared are evaluated and compared using the operator. If the comparison turns out to be true, then the first statement is performed. If the comparison fails, then the second statement is executed.

&& means "and"

You can combine two comparisons using the "and" operator, which is spelled && in C++. For example, the following Boolean expression is true (that is, is satisfied) provided x is greater than 2 *and* x is less than 7:

```
(2 < x) && (x < 7)
```

When two comparisons are connected using a &&, the entire expression is true, provided both of the comparisons are true (that is, provided both are satisfied); otherwise, the entire expression is false.

|| means "or"

You can also combine two comparisons using the "or" operator, which is spelled || in C++. For example, the following is true provided y is less than 0 *or* y is greater than 12:

```
(y < 0) || (y > 12)
```

When two comparisons are connected using a ||, the entire expression is true provided that one or both of the comparisons are true (that is, satisfied); otherwise, the entire expression is false.

parentheses

Remember that when you use a Boolean expression in an *if-else* statement, the Boolean expression must be enclosed in parentheses. Therefore, an *if-else* statement that uses the && operator and two comparisons is parenthesized as follows:

```
if ( (temperature >= 95) && (humidity >= 90) )
     ...
```

The inner parentheses around the comparisons are not required, but they do make the meaning clearer, and we will normally include them.

You can negate any Boolean expression using the ! operator. If you want to negate a Boolean expression, place the expression in parentheses and place the ! operator in front of it. For example, `!(x < y)` means "x is *not* less than y." Since the Boolean expression in an *if-else* statement must be enclosed in parentheses, you should place a second pair of parentheses around the negated expression when the negated expression is used in an *if-else* statement. For example, an *if-else* statement might begin as follows:

```
if (!(x < y))
     ...
```

The ! operator can usually be avoided. For example, our hypothetical *if-else* statement can instead begin with the following, which is equivalent and easier to read:

```
if (x >= y)
     ...
```

The "and" operator &&

You can form a more elaborate Boolean expression by combining two simple tests using the "and" operator &&.

Syntax (for a Boolean Expression Using &&)

```
(Comparison_1) && (Comparison_2)
```

Example (within an *if-else* statement)

```
if ( (score > 0) && (score < 10) )
    cout << "score is between 0 and 10\n";
else
    cout << "score is not between 0 and 10.\n";
```

If the value of `score` is greater than 0 and the value of `score` is also less than 10, then the first `cout` statement will be executed; otherwise, the second `cout` statement will be executed.

The "or" operator ||

You can form a more elaborate Boolean expression by combining two simple tests using the "or" operator ||.

Syntax (for a Boolean Expression Using ||)

```
(Comparison_1) || (Comparison_2)
```

Example (within an *if-else* statement)

```
if ( (x == 1) || (x == y) )
    cout << "x is 1 or x equals y.\n";
else
    cout << "x is neither 1 nor equal to y.\n";
```

If the value of x is equal to 1 or the value of x is equal to the value of y (or both), then the first `cout` statement will be executed; otherwise, the second `cout` statement will be executed.

We will not have much call to use the ! operator until later in this book, so we will postpone any detailed discussion of the ! operator until then.

omitting *else*

Sometimes you want one of the two alternatives in an *if-else* statement to do nothing at all. In C++ this can be accomplished by omitting the *else* part. These sorts of statements are referred to as *if* **statements** to distinguish them from *if-else* statements. For example, the first of the following two statements is an *if* statement:

```
if (sales >= minimum)
    salary = salary + bonus;
cout << "salary = $" << salary;
```

If the value of `sales` is greater than or equal to the value of `minimum`, the assignment statement is executed and then the following `cout` statement is executed. On the other hand, if the value of `sales` is less than `minimum`, then the embedded assignment statement is not executed, so the *if* statement causes no change (that is, no bonus is added to the base salary), and the program proceeds directly to the `cout` statement.

PITFALL Strings of Inequalities

Do not use a string of inequalities such as the following in your program:

```
if (x < z < y)
    cout << "z is between x and y.";
```

Do not do this!

If you do use the above, your program will probably compile and run, but it will undoubtedly give incorrect output. We will explain why this happens after we learn more details about the C++ language. The same problem will occur with a string of comparisons using any of the comparison operators; the problem is not limited to < comparisons. The correct way to express a string of inequalities is to use the "and" operator && as follows:

```
if ( (x < z) && (z < y) )          <-- correct form
    cout << "z is between x and y.";
```

PITFALL Using = in place of ==

Unfortunately, you can write many things in C++ that you would think are incorrectly formed C++ statements but turn out to have some obscure meaning. This means that if you mistakenly write something that you would expect to produce an error message, you may find out that the program compiles and runs with no error messages, but gives incorrect output. Since you may not realize you wrote something incorrectly, this can cause serious problems. By the time you realize something is wrong, the mistake may be very hard to find. One common mistake is to use the symbol = when you mean ==. For example, consider an *if-else* statement that begins as follows:

```
if (x = 12)
    Do_Something
else
    Do_Something_Else
```

Suppose you wanted to test to see if the value of x is equal to 12 so that you really meant to use == rather than =. You might think the compiler will catch your mistake. The expression

```
x = 12
```

is not something that is satisfied or not. It is an assignment statement, so surely the compiler will give an error message. Unfortunately, that is not the case. In C++ the expression `x = 12` is an expression that returns (or has) a value, just like `x + 12` or `2 + 3`. An assignment expression's value is the value transferred to the variable on the left. For example, the value of `x = 12` is 12. We saw in our discussion of Boolean value compatibility that *int* values may be converted to *true* or *false*. Since 12 is not zero, it is converted to *true*. If you use `x = 12` as the Boolean expression in an *if* statement, the Boolean expression is always *true*, so the first branch (*Do_Something*) is always executed.

This error is very hard to find, because it *looks right*! The compiler can find the error without any special instructions if you put the 12 on the left side of the comparison, as in:

```
if (12 == x)
    Do_Something;
else
    Do_Something_Else;
```

Then, the compiler will give an error message if you mistakenly use = instead of ==.

Remember that dropping one of the = in an == is a common error that is not caught by many compilers, is very hard to see, and is almost certainly not what you wanted. In C++, many executable statements can also be used as almost any kind of expression, including as a Boolean expression for an *if-else* statement. If you put an assignment statement where a Boolean expression is expected, the assignment statement will be interpreted as a Boolean expression. Of course the result of the "test" will undoubtedly not be what you intended as the Boolean expression. The above *if-else* statement looks fine at a quick glance and it will compile and run. But, in all likelihood, it will produce puzzling results when it is run.

Compound Statements

if-else with multiple statements

compound statement

You will often want the branches of an *if-else* statement to execute more than one statement each. To accomplish this, enclose the statements for each branch between a pair of braces, { and }, as indicated in the second syntax template in Display 2.7. This is illustrated in Display 2.9. A list of statements enclosed in a pair of braces is called a **compound statement.** A compound statement is treated as a single statement by C++ and may be used anywhere that a single statement may be used. (Thus, the second syntax template in Display 2.7 is really just a special case of the first one.) Display 2.9 contains two compound statements, embedded in an *if-else* statement. The compound statements are in color.

Display 2.9 Compound Statements Used with *if-else*

```
if (my_score > your_score)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

Syntax rules for *if-else* demand that the Yes Statement and No Statement be exactly one statement. If more statements are desired for a branch, the statements must be enclosed in braces to convert them to one compound statement. If two or more statements not enclosed by braces are placed between the *if* and the *else*, then the compiler will give an error message.

SELF-TEST EXERCISES

20 Write an *if-else* statement that outputs the word `High` if the value of the variable `score` is greater than `100` and `Low` if the value of `score` is at most `100`. The variable `score` is of type *int*.

21 Suppose `savings` and `expenses` are variables of type *double* that have been given values. Write an *if-else* statement that outputs the word `Solvent`, decreases the value of `savings` by the value of `expenses`, and sets the value of `expenses` to 0, provided that `savings` is at least as large as `expenses`. If, however, `savings` is less than `expenses`, the *if-else* statement simply outputs the word `Bankrupt`, and does not change the value of any variables.

22 Write an *if-else* statement that outputs the word `Passed` provided the value of the variable `exam` is greater than or equal to `60` and the value of the variable `programs_done` is greater than or equal to `10`. Otherwise, the *if-else* statement outputs the word `Failed`. The variables `exam` and `programs_done` are both of type *int*.

23 Write an *if-else* statement that outputs the word `Warning` provided that either the value of the variable `temperature` is greater than or equal to `100`, or the value of the variable `pressure` is greater than or equal to `200`, or both. Otherwise, the *if-else* statement outputs the word `OK`. The variables `temperature` and `pressure` are both of type *int*.

24 Consider a quadratic expression, say

$x^2 - x - 2$

Describing where this quadratic is positive (that is, greater than 0), involves describing a set of numbers that are either less than the smaller root (which is –1) or greater than the larger root (which is +2). Write a C++ Boolean expression that is true when this formula has positive values.

25 Consider the quadratic expression

$x^2 - 4x + 3$

Describing where this quadratic is negative involves describing a set of numbers that are simultaneously greater than the smaller root (+1) and less than the larger root (+3). Write a C++ Boolean expression that is true when the value of this quadratic is negative.

26 What is the output of the following `cout` statements embedded in these *`if-else`* statements? You are to assume that these are embedded in a complete correct program. Explain your answer.

a.
```
if(0)
  cout << "0 is true";
else
  cout << "0 is false";
cout << endl;
```

b.
```
if(1)
  cout << "1 is true";
else
  cout << "1 is false";
cout << endl;
```

c.
```
if(-1)
  cout << "-1 is true";
else
  cout << "-1 is false";
cout << endl;
```

Note: This is an exercise only. This is *not* intended to illustrate programming style you should follow.

Simple Loop Mechanisms

Most programs include some action that is repeated a number of times. For example, the program in Display 2.6 computes the gross pay for one worker. If the company employs 100 workers, then a more complete payroll program would repeat this calculation 100 times. A portion of a program that repeats a statement or group of statements is called a **loop.** The C++ language has a number of ways to create loops. One of these constructions is called a *`while`* **statement** or *`while`* **loop.** We will first illustrate its use with a short toy example and then do a more realistic example.

`while` statement

The program in Display 2.10 contains a simple *`while`* statement shown in color. The portion between the braces, { and }, is called the **body** of the *`while`* loop; it is the

loop body

Display 2.10 A *while* Loop


```
#include <iostream>
using namespace std;
int main( )
{
    int count_down;

    cout << "How many greetings do you want? ";
    cin >> count_down;

    while (count_down > 0)
    {
        cout << "Hello ";
        count_down = count_down - 1;
    }

    cout << endl;
    cout << "That's all!\n";

    return 0;
}
```

Sample Dialogue 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

Sample Dialogue 2

```
How many greetings do you want? 1
Hello
That's all!
```

Sample Dialogue 3

```
How many greetings do you want? 0

That's all!
```

The loop body is executed zero times.

action that is repeated. The statements inside the braces are executed in order, then they are executed again, then again, and so forth until the *while* loop ends. In the first sample dialogue, the body is executed three times before the loop ends, so the program outputs Hello three times. Each repetition of the loop body is called an **iteration** of the loop, and so the first sample dialogue shows three iterations of the loop.

iteration

The meaning of a *while* statement is suggested by the English word *while*. The loop is repeated *while the Boolean expression in the parentheses is satisfied*. In Display 2.10 this means that the loop body is repeated as long as the value of the variable count_down is greater than 0. Let's consider the first sample dialogue and see how the *while* loop performs. The user types in **3** so the cin statement sets the value of count_down to 3. Thus, in this case, when the program reaches the *while* statement, it is certainly true that count_down is greater than 0, so the statements in the loop body are executed. Every time the loop body is repeated, the following two statements are executed:

```
cout << "Hello ";
count_down = count_down - 1;
```

Therefore, every time the loop body is repeated, "Hello " is output and the value of the variable count_down is decreased by one. After the computer repeats the loop body three times, the value of count_down is decreased to 0 and the Boolean expression in parentheses is no longer satisfied. So, this *while* statement ends after repeating the loop body three times.

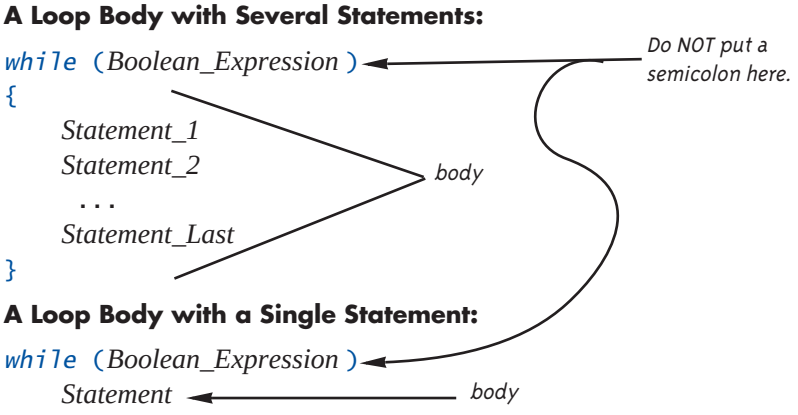
The syntax for a *while* statement is given in Display 2.11. The *Boolean_Expressions* allowed are exactly the same as the Boolean expressions allowed in an *if-else* statement. Just as in *if-else* statements, the Boolean expression in a *while* statement must be enclosed in parentheses. In Display 2.11 we have given the syntax templates for two cases: the case when there is more than one statement in the loop body and the case when there is just a single statement in the loop body. Note that when there is only a single statement in the loop body, you need not include the braces { and }.

Let's go over the actions performed by a *while* statement in greater detail. When the *while* statement is executed, the first thing that happens is that the Boolean expression following the word *while* is checked. It is either true or false. For example, the comparison

```
count_down > 0
```

is true if the value of count_down is positive. If it is false, then no action is taken and the program proceeds to the next statement after the *while* statement. If the comparison is true, then the entire body of the loop is executed. At least one of the expressions being compared typically contains something that might be changed by the loop body, such as the value of count_down in the *while* statement in Display 2.10. After the body of the

Display 2.11 Syntax of the *while* Statement



loop is executed, the comparison is again checked. This process is repeated again and again as long as the comparison continues to be true. After each iteration of the loop body, the comparison is again checked and if it is true, then the entire loop body is executed again. When the comparison is no longer true, the *while* statement ends.

executing the loop body zero times

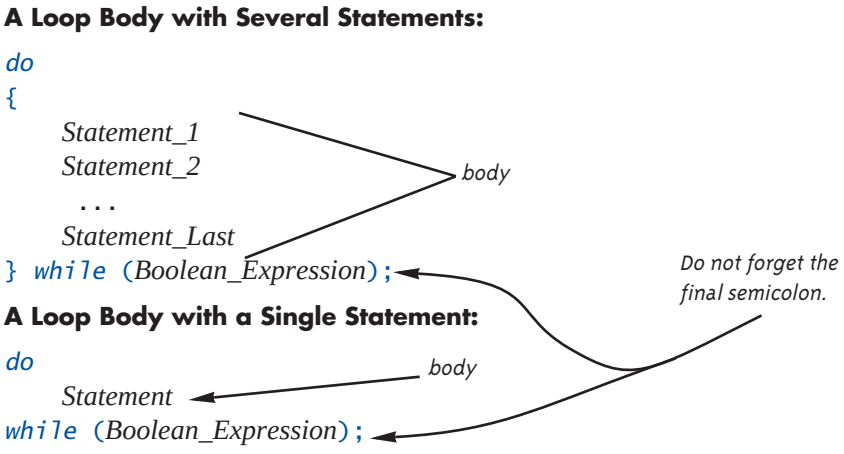
The first thing that happens when a *while* statement is executed is that the Boolean expression is checked. If the Boolean expression is not true when the *while* statement begins, then the loop body is never executed. That is exactly what happens in Sample Dialogue 3 of Display 2.10. In many programming situations you want the possibility of executing the loop body zero times. For example, if your *while* loop is reading a list consisting of all the failing scores on an exam and nobody failed the exam, then you want the loop body to be executed zero times.

do-while statement

As we just noted, a *while* loop might execute its loop body zero times, which is often what you want. If on the other hand you know that *under all circumstances* your loop body should be executed at least one time, then you can use a *do-while* statement. A *do-while* statement is similar to a *while* statement except that the loop body is always executed at least once. The syntax for a *do-while* statement is given in Display 2.12. A program with a sample *do-while* loop is given in Display 2.13. In that *do-while* loop, as in any *do-while* loop, the first thing that happens is that the statements in the loop body are executed. After that first iteration of the loop body, the *do-while* statement behaves the same as a *while* loop. The Boolean expression is checked. If the Boolean expression is true, the loop body is executed again; the Boolean expression is checked again, and so forth.

Display 2.12 Syntax of the *do-while* Statement

A Loop Body with Several Statements:



A Loop Body with a Single Statement:

Increment and Decrement Operators

++ and --

We discussed binary operators in the section entitled "Arithmetic Operators and Expressions." Binary operators have two operands. Unary operators have only one operand. You already know of two unary operators, + and –, as used in the expressions +7 and –7. The C++ language has two other very common unary operators, ++ and --. The ++ operator is called the **increment operator** and the -- operator is called the **decrement operator.** They are usually used with variables of type *int*. If n is a variable of type *int*, then n++ increases the value of n by one and n-- decreases the value of n by one. So n++ and n-- (when followed by a semicolon) are executable statements. For example, the statements

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

yield the following output:

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

Display 2.13 A *do-while* Loop


```
#include <iostream>
using namespace std;
int main( )
{
    char ans;

    do
    {
        cout << "Hello\n";
        cout << "Do you want another greeting?\n"
             << "Press y for yes, n for no,\n"
             << "and then press return: ";
        cin >> ans;
    } while (ans == 'y' || ans == 'Y');

    cout << "Good-Bye\n";

    return 0;
}
```

Sample Dialogue

```
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press return: y
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press return: Y
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press return: n
Good-Bye
```

And now you know where the "++" came from in the name "C++."

Increment and decrement statements are often used in loops. For example, we used the following statement in the *while* loop in Display 2.10:

```
count_down = count_down – 1;
```

However, most experienced C++ programmers would use the decrement operator rather than the assignment statement, so that the entire *while* loop would read as follows:

```
while (count_down > 0)
{
    cout << "Hello ";
    count_down--;
}
```

Programming EXAMPLE
Charge Card Balance

Suppose you have a bank charge card with a balance owed of $50 and suppose the bank charges you 2% per month interest. How many months can you let pass without making any payments before your balance owed will exceed $100? One way to solve this problem is to simply read each monthly statement and count the number of months that go by until your balance reaches $100 or more. Better still, you can calculate the monthly balances with a program rather than waiting for the statements to arrive. In this way you will obtain an answer without having to wait so long (and without endangering your credit rating).

After one month the balance would be $50 plus 2% of $50, which is $51. After two months the balance would be $51 plus 2% of $51, which is $52.02. After three months the balance would be $52.02 plus 2% of $52.02, and so on. In general, each month increases the balance by 2%. The program could keep track of the balance by storing it in a variable called `balance`. The change in the value of `balance` for one month can be calculated as follows:

```
balance = balance + 0.02*balance;
```

If we repeat this action until the value of `balance` reaches (or exceeds) `100.00` and we count the number of repetitions, then we will know the number of months it will take for the balance to reach `100.00`. To do this we need another variable to count

the number of times the balance is changed. Let us call this new variable `count`. The final body of our *`while`* loop will thus contain the following statements:

```
balance = balance + 0.02*balance;
count++;
```

In order to make this loop perform correctly, we must give appropriate values to the variables `balance` and `count` before the loop is executed. In this case, we can initialize the variables when they are declared. The complete program is shown in Display 2.14.

PITFALL Infinite Loops

A *`while`* loop or *`do-while`* loop does not terminate as long as the Boolean expression after the word *`while`* is true. This Boolean expression normally contains a variable that will be changed by the loop body, and usually the value of this variable eventually is changed in a way that makes the Boolean expression false and therefore terminates the loop. However, if you make a mistake and write your program so that the Boolean expression is always true, then the loop will run forever. A loop that runs forever is called an **infinite loop.**

infinite loop

First let's describe a loop that does terminate. The following C++ code will write out the positive even numbers less than 12. That is, it will output the numbers 2, 4, 6, 8, and 10, one per line, and then the loop will end.

```
x = 2;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
}
```

The value of `x` is increased by `2` on each loop iteration until it reaches `12`. At that point, the Boolean expression after the word *`while`* is no longer true, so the loop ends.

Now suppose you want to write out the odd numbers less than 12, rather than the even numbers. You might mistakenly think that all you need do is change the initializing statement to

```
x = 1;
```

but this mistake will create an infinite loop. Because the value of `x` goes from `11` to `13`, the value of `x` is never equal to `12`, so the loop will never terminate.

Display 2.14 Charge Card Program


```
#include <iostream>
using namespace std;
int main( )
{
    double balance = 50.00;
    int count = 0;

    cout << "This program tells you how long it takes\n"
         << "to accumulate a debt of $100, starting with\n"
         << "an initial balance of $50 owed.\n"
         << "The interest rate is 2% per month.\n";

    while (balance < 100.00)
    {
        balance = balance + 0.02 * balance;
        count++;
    }

    cout << "After " << count << " months,\n";
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "your balance due will be $" << balance << endl;

    return 0;
}
```

Sample Dialogue

```
This program tells you how long it takes
to accumulate a debt of $100, starting with
an initial balance of $50 owed.
The interest rate is 2% per month.
After 36 months,
your balance due will be $101.99
```

This sort of problem is common when loops are terminated by checking a numeric quantity using == or !=. When dealing with numbers, it is always safer to test for passing a value. For example, the following will work fine as the first line of our *while* loop:

```
while (x < 12)
```

With this change, x can be initialized to any number and the loop will still terminate.

A program that is in an infinite loop will run forever unless some external force stops it. Since you can now write programs that contain an infinite loop, it is a good idea to learn how to force a program to terminate. The method for forcing a program to stop varies from system to system. The keystrokes Control-C will terminate a program on many systems. (To type a Control-C hold down the Control key while pressing the C key.)

SELF-TEST EXERCISES

27 What is the output produced by the following (when embedded in a correct program with x declared to be of type *int*)?

```
x = 10;
while (x > 0)
{
    cout << x << endl;
    x = x - 3;
}
```

28 What output would be produced in the previous exercise if the > sign were replaced with <?

29 What is the output produced by the following (when embedded in a correct program with x declared to be of type *int*)?

```
x = 10;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

30 What is the output produced by the following (when embedded in a correct program with x declared to be of type *int*)?

```
x = -42;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

31 What is the most important difference between a *while* statement and a *do-while* statement?

32 What is the output produced by the following (when embedded in a correct program with x declared to be of type *int*)?

```
x = 10;
while (x > 0)
{
    cout << x << endl;
    x = x + 3;
}
```

33 Write a complete C++ program that outputs the numbers 1 to 20, one per line. The program does nothing else.

2.5 Program Style

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

OSCAR WILDE, *THE IMPORTANCE OF BEING EARNEST*

All the variable names in our sample programs were chosen to suggest their use. Our sample programs were laid out in a particular format. For example, the declarations and statements were all indented the same amount. These and other matters of style are of more than aesthetic interest. A program that is written with careful attention to style is easier to read, easier to correct, and easier to change.

Indenting

A program should be laid out so that elements that are naturally considered a group are made to look like a group. One way to do this is to skip a line between parts that are logically considered separate. Indenting can also help to make the structure of the program clearer. A statement within a statement should be indented. In particular,

if-else statements, *while* loops, and *do-while* loops should be indented either as in our sample programs or as in some similar manner.

where to place braces { }

The braces { } determine a large part of the structure of a program. Placing each brace on a line by itself, as we have been doing, makes it easy to find the matching pairs. Notice that we have indented some pairs of braces. When one pair of braces is embedded in another pair, the embedded braces are indented more than the outer braces. Look back at the program in Display 2.14. The braces for the body of the *while* loop are indented more than the braces for the `main` part of the program.

There are at least two schools of thought on where you should place braces. The first, which we use in this book, is to reserve a separate line for each brace. This form is easiest to read. The second school of thought holds that the opening brace for a pair need not be on a line by itself. If used with care, this second method can be effective, and it does save space. The important point is to use a style that shows the structure of the program. The exact layout is not precisely dictated, but you should be consistent within any one program.

Comments

In order to make a program understandable, you should include some explanatory notes at key places in the program. Such notes are called **comments.** C++ and most other programming languages have provisions for including such comments within the text of a program. In C++ the symbols `//` are used to indicate the start of a comment. All of 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 `//` are two slashes (without a space between them).

In this book, comments will always be written in italic so they stand out from the program text. Some text editors indicate comments by showing them in a different color from the rest of the program text.

/*comments*/

There is another way to insert comments in a C++ program. Anything between the symbol pair `/*` and the symbol pair `*/` 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 only placed at the ends of lines or on separate lines by themselves.

There are differing opinions on which kind of comment is best to use. Either variety (the // kind or the /* */ kind) can be effective if used with care. We will use the // kind in this book.

when to comment

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

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

Notice the comment given at the start of the program in Display 2.15. All programs should begin with a comment similar to the one shown there. It gives all the essential information about the program: what file the program is in, who wrote the program, how to contact the person who wrote the program, what the program does, the date that the program was last modified, and any other particulars that are appropriate, such as the assignment number, if the program is a class assignment. Exactly what you include in this comment will depend on your particular situation. We will not include such long comments in the programs in the rest of this book, but you should always begin your programs with such a comment.

Naming Constants

There are two problems with numbers in a computer program. 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. In order 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, that four of the times it represents the number of branch offices, and that 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 that some of the 10's 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

Display 2.15 Comments and Named Constants


```
//File Name: health.cpp (Your system may require some suffix other than cpp.)
//Author: Your Name Goes Here.
//Email Address: you@yourmachine.bla.bla
//Assignment Number: 2
//Description: Program to determine if the user is ill.
//Last Changed: September 23, 2004

#include <iostream>
using namespace std;
int main( )
{
    const double NORMAL = 98.6;//degrees Fahrenheit
    double temperature;

    cout << "Enter your temperature: ";
    cin >> temperature;

    if (temperature > NORMAL)
    {
        cout << "You have a fever.\n";
        cout << "Drink lots of liquids and get to bed.\n";
    }
    else
    {
        cout << "You don't have a fever.\n";
        cout << "Go study.\n";
    }

    return 0;
}
```

Your programs should always begin with a comment similar to this one.

Sample Dialogue

```
Enter your temperature: 98.6
You don't have a fever.
Go study.
```

of 10, but when the bank opens a new branch, all you need do in order to update the program is to 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.

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 constants

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.

Although unnamed numeric constants are allowed in a program, you should seldom use them. It often makes sense to use unnamed number constants for well-known, easily recognizable, and unchangeable quantities, such as 100 for the number of centimeters in a meter. However, all other numeric constants should be given names in the fashion we just described. This will make your programs easier to read and easier to change.

Display 2.15 contains a simple program that illustrates the use of the declaration modifier *const*.

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;
```

SELF-TEST EXERCISES

34 The following *if-else* statement will compile and run without any problems. However, it is not laid out in a way that is consistent with the other *if-else* statements we have used in our programs. Rewrite it so that the layout (indenting and line breaks) matches the style we used in this chapter.

```
if (x < 0) {x = 7; cout << "x is now positive.";} else
{x = -7; cout << "x is now negative.";}
```

35 What output would be produced by the following two lines (when embedded in a complete and correct program)?

```
//cout << "Hello from";
cout << "Self-Test Exercise";
```

36 Write a complete C++ program that asks the user for a number of gallons and then outputs the equivalent number of liters. There are 3.78533 liters in a gallon. Use a declared constant. Since this is just an exercise, you need not have any comments in your program.

CHAPTER SUMMARY

- Use meaningful names for variables.
- Be sure to check that variables are declared to be of the correct data type.

- 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.
- Use enough parentheses in arithmetic expressions to make the order of operations clear.
- Always include a prompt line in a program whenever the user is expected to enter data from the keyboard, and always echo the user's input.
- An *if-else* statement allows your program to choose one of two alternative actions. An *if* statement allows your program to decide whether or not to perform some one particular action.
- A *do-while* loop always executes its loop body at least once. In some situations a *while* loop might not execute the body of the loop at all.
- 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 an indenting, spacing, and line break pattern similar to the sample programs.
- Insert comments to explain major subsections or any unclear part of a program.

Answers to Self-Test Exercises

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

2
```
int count = 0;
double distance = 1.5;
```

Alternatively, you could use

```
int count(0);
double distance(1.5);
```

3
```
sum = n1 + n2;
```

4
```
length = length + 8.3;
```

5
```
product = product*n;
```

6 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 << endl;
  return 0;
}
```

7 There is no unique right answer for this one. Below are possible answers:

a. speed
b. pay_rate
c. highest or max_score

8
```
cout << "The answer to the question of\n"
     << "Life, the Universe, and Everything is 42.\n";
```

9
```
cout << "Enter a whole number and press return: ";
cin >> the_number;
```

10
```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(3);
```

11
```
#include <iostream>
using namespace std;
int main( )
{
    cout << "Hello world\n";
    return 0;
}
```

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

int main( )
{
    int n1, n2, sum;
    cout << "Enter two whole numbers\n";
    cin >> n1 >> n2;
    sum = n1 + n2;
```

```
    cout << "The sum of " << n1 << " and "
         << n2 << " is " << sum << endl;
    return 0;
}
```

13
```
cout << endl << "\t";
```

14
```
#include <iostream>
using namespace std;

int main( )
{
  double one(1.0), two(1.414), three(1.732), four(2.0),
         five(2.236);
  cout << "\tN\tSquare Root\n";
  cout << "\t1\t" << one << endl
       << "\t2\t" << two << endl
       << "\t3\t" << three << endl
       << "\t4\t" << four << endl
       << "\t5\t" << five << endl;
  return 0;
}
```

15
```
3*x
3*x + y
(x + y)/7
(3*x + y)/(z + 2)
```
Note that `x + y/7` is not correct.

16
```
bcbc
```

17
```
(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`.

18
```
#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) << endl
         << "with a remainder of " << (number1%number2)
         << endl;
    return 0;
}
```

19 a. `52.0`

b. `9/5` has `int` value `1`, since numerator and denominator are both `int`, integer division is done; the fractional part is discarded.

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

20
```
if (score > 100)
    cout << "High";
else
    cout << "Low";
```

You may want to add `\n` to the end of the above quoted strings depending on the other details of the program.

21
```
if (savings >= expenses)
{
    savings = savings - expenses;
    expenses = 0;
    cout << "Solvent";
}
else
{
    cout << "Bankrupt";
}
```

You may want to add `\n` to the end of the above quoted strings depending on the other details of the program.

22
```
if ( (exam >= 60) && (programs_done >= 10) )
    cout << "Passed";
else
    cout << "Failed";
```

You may want to add \n to the end of the above quoted strings depending on the other details of the program.

23
```
if ( (temperature >= 100) || (pressure >= 200) )
    cout << "Warning";
else
    cout << "OK";
```

You may want to add \n to the end of the above quoted strings depending on the other details of the program.

24 `(x < -1) || ( x > 2)`

25 `(1 < x) && (x < 3)`

26 a. 0 is *false*. In the section on type compatibility, it is noted that the *int* value 0 converts to *false*.

b. 1 is *true*. In the section on type compatibility, it is noted that a nonzero *int* value converts to *true*.

c. -1 is *true*. In the section on type compatibility, it is noted that a nonzero *int* value converts to *true*.

27
```
10
7
4
1
```

28 There would be no output, since the Boolean expression `(x < 0)` is not satisfied and so the *while* statement ends without executing the loop body.

29 The output is exactly the same as it was for Self-Test Exercise 27.

30 The body of the loop is executed before the Boolean expression is checked, the Boolean expression is false, and so the output is

```
-42
```

31 With a *do-while* statement the loop body is always executed at least once. With a *while* statement there can be conditions under which the loop body is not executed at all.

32 This is an infinite loop. The output would begin with the following and conceptually go on forever:

```
10
13
16
19
```

(Once the value of x becomes larger than the largest integer allowed on your computer, the program may stop or exhibit other strange behavior, but the loop is conceptually an infinite loop.)

33
```
#include <iostream>
using namespace std;

int main( )
{
    int n = 1;
    while (n <= 20)
    {
        cout << n << endl;
        n++;
    }
    return 0;
}
```

34
```
if (x < 0)
{
    x = 7;
    cout << "x is now positive.";
}
else
{
    x = -7;
    cout << "x is now negative.";
}
```

35 The first line is a comment and is not executed. So the entire output is just the following line:

```
Self-Test Exercise
```

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

int main( )
{
    const double LITERS_PER_GALLON = 3.78533;
    double gallons, liters;

    cout << "Enter the number of gallons:\n";
    cin >> gallons;

    liters = gallons*LITERS_PER_GALLON;
    cout << "There are " << liters << " in "
         << gallons << " gallons.\n";

    return 0;
}
```

Programming Projects

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. Your program should allow the user to repeat this calculation as often as the user wishes.

2 A government research lab has concluded that an artificial sweetener commonly used in diet soda pop will cause death in laboratory mice. A friend of yours is desperate to lose weight but cannot give up soda pop. Your friend wants to know how much diet soda pop 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 1/10th 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 percent as the `double` value `0.001`. Your program should allow the calculation to be repeated as often as the user wishes.

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. Your program should allow the calculation to be repeated as often as the user wishes.

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, to yield $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. Your program should allow the calculations to be repeated as often as the user wishes.

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. For a harder version write your program so that it allows the calculation to be repeated as often as the user wishes. If this is a class exercise, ask your instructor if you should do this harder version or not.

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 will then output the worker's gross pay, each withholding amount, and the net take-home pay for the week. For a harder version write your program so that it allows the calculation to be repeated as often as the user wishes. If this is a class exercise, ask your instructor if you should do this harder version or not.

7 It is difficult to make a budget that spans several years, because prices are not stable. If your company needs 200 pencils per year, you cannot simply use this year's price as the cost of pencils two years from now. Because of inflation the cost is likely to be higher than it is today. Write a program to gauge the expected cost of an item in a specified number of years. The program asks for the cost of the item, the number of years from now that the item will be purchased, and the rate of inflation. The program then outputs the estimated cost of the item after the specified period. Have the user enter the inflation rate as a percentage, like 5.6 (percent). Your program should then convert the percent to a fraction, like 0.056, and should use a loop to estimate the price adjusted for inflation. (*Hint:* This is similar to computing interest on a charge card account, which was discussed in this chapter.)

8 You have just purchased a stereo system that cost $1,000 on the following credit plan: no down payment, an interest rate of 18% per year (and hence 1.5% per month), and monthly payments of $50. The monthly payment of $50 is used to pay the interest and whatever is left is used to pay part of the remaining debt. Hence, the first month you pay 1.5% of $1,000 in interest. That is $15 in interest. So, the remaining $35 is deducted from your debt, which leaves you with a debt of $965.00. The next month you pay interest of 1.5% of $965.00, which is $14.48. Hence, you can deduct $35.52 (which is $50 – $14.48) from the amount you owe. Write a program that will tell you how many months it will take you to pay off the loan, as well as the total amount of interest paid over the life of the loan. Use a loop to calculate the amount of interest and the size of the debt after each month. (Your final program need not output the monthly amount of interest paid and remaining debt, but you may want to write a preliminary version of the program that does output these values.) Use a variable to count the number of loop iterations and hence the number of months until the debt is zero. You may want to use other variables as well. The last payment may be less than $50 if the debt is small, but do not forget the interest. If you owe $50, then your monthly payment of $50 will not pay off your debt, although it will come close. One month's interest on $50 is only 75 cents.

9 Write a program that reads in ten whole numbers and that outputs the sum of all the numbers greater than zero, the sum of all the numbers less than zero

(which will be a negative number or zero), and the sum of all the numbers, whether positive, negative, or zero. The user enters the ten numbers just once each and the user can enter them in any order. Your program should not ask the user to enter the positive numbers and the negative numbers separately.