Control, Functions, and Classes 4

If A equals success, then the formula is $A = X + Y + Z$.
X is work. Y is play. Z is keep your mouth shut.
Albert Einstein
quoted in SIGACT News, Vol. 25, No. 1, March 1994

Your "if" is the only peacemaker; much virtue in "if."
William Shakespeare
As You Like It, V, iv

Leave all else to the gods.
Horace
Odes, Book I, Ode ix

In the programs studied in Chapter 3, statements executed one after the other to produce output. This was true both when all statements were in `main` or when control was transferred from `main` to another function, as it was in `SlicePrice` in *pizza.cpp*, Program 3.5. However, code behind the scenes in *gfly.cpp*, Program 3.6, executed differently in response to the user's input and to a simulated wind-shear effect. Many programs require nonsequential control. For example, transactions made at automatic teller machines (ATMs) process an identification number and present you with a screen of choices. The program controlling the ATM executes different code depending on your choice—for example, either to deposit money or to get cash. This type of control is called **selection:** a different code segment is selected and executed based on interaction with the user or on the value of a program variable.

Another type of program control is **repetition:** the same sequence of C++ statements is repeated, usually with different values for some of the variables in the statements. For example, to print a yearly calendar your program could call a `PrintMonth` function twelve times:

```
PrintMonth("January", 31);
//...
PrintMonth("November",30);
PrintMonth("December",31);
```

Here the name of the month and the number of days in the month are arguments passed to `PrintMonth`. Alternatively, you could construct the `PrintMonth` function to determine the name of the month as well as the number of days in the month given the year and the number of the month. This could be done for the year 2000 by repeatedly executing the following statement and assigning values of 1, 2, ..., 12 to `month`:

```
PrintMonth(month, 2000);
```

In this chapter we'll study methods for controlling how the statements in a program are executed and how this control is used in constructing functions and classes. To do this we'll expand our study of arithmetic operators, introduced in the last chapter, to include operators for other kinds of data. We'll also study C++ statements that alter the flow of control within a program. Finally, we'll see how functions and classes can be used as a foundation on which we'll continue to build as we study how programs are used to solve problems. At the end of the chapter you'll be able to write the function `PrintMonth` but you'll also see a class that encapsulates the function so you don't have to write it.

4.1 The Assignment Operator

In the next three sections we'll use a program that makes change using U.S. coins to study relational and assignment statements and conditional execution. We'll use the same program as the basis for what could be a talking cash register.

A run of *change.cpp,* Program 4.1, shows how change is made using quarters, dimes, nickels, and pennies. The program shows how values can be stored in variables using the **assignment operator,** `=`. In previous programs the user entered values for variables, but values can also be stored using the assignment operator. The code below assigns values for the circumference and area of a circle according to the radius' value, then prints the values.[1]

```
double radius, area, circumference;
cout << "enter radius: ";
cin >> radius;
area = 3.14159*radius*radius;
circumference = 3.14159*2*radius;
cout << "area = " << area
     << " circumference = " << circumference << endl;
```

The assignment operator in Program 4.1 has two purposes, it assigns the number of each type of coin needed to the appropriate variable (e.g., `quarters` and `dimes`) and it resets the value of the variable `amount` so that change will be correctly calculated.

Syntax: assignment operator =

variable = *expression*

The assignment operator = stores values in variables. The expression on the right-hand side of the = is evaluated, and this value is stored in the memory location associated with the variable named on the left-hand side of the `=`. The use of the equal sign to assign values to variables can cause confusion, especially if you say "equals" when you read an expression like `quarters = amount/25`. Operationally, the value on the right is stored in `quarters`, and it would be better to write `quarters` $\leftarrow$ `amount / 25`. The assignment statement can be read as *"The memory location of the variable quarters is*

[1]The formula for the area of a circle is πr^2, the formula for circumference is $2\pi r$ where r is the radius.

assigned the value of amount/25," but that is cumbersome (at best). If you can bring yourself to say "gets" for =, you'll find it easier to distinguish between = and == (the boolean equality operator). Verbalizing the process by saying "*Quarters gets amount divided by twenty-five*" will help you understand what's happening when assignment statements are executed.

Program 4.1 change.cpp

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

// make change in U.S. coins
// Owen Astrachan, 03/17/99

int main()
{
    int amount;
    int quarters, dimes, nickels, pennies;

    // input phase of program

    cout << "make change in coins for what amount: ";
    cin >> amount;

    // calculate number of quarters, dimes, nickels, pennies

    quarters = amount/25;
    amount = amount − quarters∗25;

    dimes = amount/10;
    amount = amount − dimes∗10;

    nickels = amount/5;
    amount = amount − nickels∗5;

    pennies = amount;

    // output phase of program

    cout << "# quarters =\t" << quarters << endl;
    cout << "# dimes =\t"    << dimes    << endl;
    cout << "# nickels =\t"  << nickels  << endl;
    cout << "# pennies =\t"  << pennies  << endl;

    return 0;
}
```

change.cpp

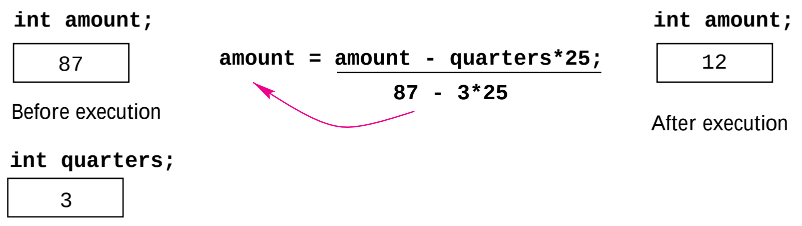


Figure 4.1 Updating a variable via assignment.

OUTPUT

```
prompt> change
make change in coins for what amount: 87
# quarters =    3
# dimes =       1
# nickels =     0
# pennies =     2
prompt> change
make change in coins for what amount: 42
# quarters =    1
# dimes =       1
# nickels =     1
# pennies =     2
```

The statement `amount = amount - quarters*25` updates the value of the variable `amount`. The right-hand side of the statement is evaluated first. The value of this expression, `amount - quarters*25`, is stored in the variable on the left-hand side of the assignment statement—that is, the variable `amount`. This process is diagrammed in Fig. 4.1 when `amount` is 87.

A sequence of assignments can be chained together in one statement:

```
x = y = z = 13;
```

This statement assigns the value 13 to the variables `x`, `y`, and `z`. The statement is interpreted as `x = (y = (z = 13))`. The value of the expression `(z = 13)` is 13, the value assigned to `z`. This value is assigned to `y`, and the result of the assignment to `y` is 13. This result of the expression `(y = 13)` is then assigned to `x`. Parentheses aren't needed in the statement `x = y = z = 13`, because the assignment operator = is **right-associative:** in the absence of parentheses the rightmost = is evaluated first.

Table 4.1 Escape sequences in C++

escape sequence	name	ASCII
`\n`	newline	NL (LF)
`\t`	horizontal tab	HT
`\v`	vertical tab	VT
`\b`	backspace	BS
`\r`	carriage return	CR
`\f`	form feed	FF
`\a`	alert (bell)	BEL
`\\`	backslash	\
`\?`	question mark	?
`\'`	single quote (apostrophe)	'
`\"`	double quote	"

In contrast, the subtraction operator is **left-associative,** so the expression `8 - 3 - 2` is equal to 3, because it is evaluated as `(8 - 3) - 2` rather than `8 - (3 - 2)`: here the leftmost subtraction is evaluated first. Most operators are left-associative; the associativity of all C++ operators is shown in Table A.4 in Howto A.

Escape Sequences. The output of *change.cpp* is aligned using a tab character `'\t'`. The tab character prints one tab position, ensuring that the amounts of each kind of coin line up. The backslash and `t` to print the tab character are an example of an **escape sequence.** Common escape sequences are given in Table 4.1. The table is repeated as Table A.5 in Howto A. Each escape sequence prints a single character. For example, the following statement prints the four-character string `"\'"`.

```
cout << "\"\\\'\"" << endl;
```

4.2 Choices and Conditional Execution

I shall set forth from somewhere, I shall make the reckless choice
Robert Frost
The Sound of the Trees

In this section we'll alter Program 4.1 so that it only prints the coins used in giving change. We'll also move the output part of the program to a separate function. By parameterizing the output and using a function, we make it simpler to incorporate modifications to the original program.

Program Tip 4.1: Avoid duplicating the same code in several places in the same program. Programs will be modified. If you need to make the same change in more than one place in your code it is very likely that you will leave some changes out, or make the changes inconsistently. In many programs more time is spent in **program maintenance** than in **program development**. Often, moving duplicated code to a function and calling the function several times helps avoid code duplication.

Program 4.2 change2.cpp

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

// make change in U.S. coins
// Owen Astrachan, 03/17/99

void Output(string coin, int amount)
{
    if (amount > 0)
    {   cout << "# " << coin << " =\t" << amount << endl;
    }
}

int main()
{
    int amount;
    int quarters, dimes, nickels, pennies;

    // input phase of program

    cout << "make change in coins for what amount: ";
    cin >> amount;

    // calculate number of quarters, dimes, nickels, pennies

    quarters = amount/25;
    amount = amount − quarters*25;

    dimes = amount/10;
    amount = amount − dimes*10;

    nickels = amount/5;
    amount = amount − nickels*5;

    pennies = amount;

    // output phase of program

    Output("quarters",quarters);
    Output("dimes",dimes);
    Output("nickels",nickels);
    Output("pennies",pennies);

    return 0;
}
```

change2.cpp

OUTPUT

```
prompt> change2
make change in coins for what amount: 87
# quarters =    3
# dimes =       1
# pennies =     2
```

In the function `Output` an `if` statement is used for **conditional execution**—that is, `if` makes the execution depend on the value of `amount`. In the C++ statement

```
if (amount > 0)
{    cout << "# " << coin << " =\t" << amount << endl;
}
```

the **test expression** `(amount > 0)` controls the `cout <<` statement so that output appears only if the value of the `int` variable `amount` is greater than zero.

4.2.1 The `if/else` Statement

An `if` statement contains a test expression and a **body:** a group of statements within curly braces { and }. These statements are executed *only* when the test expression, also called a **condition** or a **guard,** is true. The test *must* be enclosed by parentheses. In the next section we'll explore operators that can be used in tests, including <, <=, >, and >=. The body of the `if` statement can contain any number of statements. The curly braces that are used to delimit the body of the `if` statement aren't needed when there's only one statement in the body, but we'll always use them as part of a **defensive programming** strategy designed to ward off bugs before they appear.

Syntax: if statement

```
if ( test expression )
{
   statement list;
}
```

Program 4.3 shows that an `if` statement can have an `else` part, which also controls, or guards, a body of statements within curly braces { and } that is executed when the test expression is false. Any kind of statement can appear in the body of an `if/else` statement, including other `if/else` statements. We'll discuss formatting conventions for writing such code after we explore the other kinds of operators that can be used in the test expressions that are part of `if` statements. You may find yourself writing code with an **empty** `if` or `else` body: one with no statements. This can always be avoided by changing the test used with the `if` using rules of logic we'll discuss in Section 4.7.

Syntax: if/else statement

```
if ( test expression )
{
   statement list;
}
else
{
   statement list;
}
```

In Program 4.3, if the value of `response` is something other than `"yes"`, then the `cout <<` statements associated with the `if` section are not executed, and the statements in the `else` section of the program are executed instead. In particular, if the user enters `"yeah"` or `"yup"`, then the program takes the same action as when the user enters `"no"`. Furthermore, the answer `"Yes"` is also treated like the answer `"no"` rather than `"yes"`, because a capital letter is different from the equivalent lower-case letter. As we saw in Program 4.1, *change.cpp*, the rules of C++ do *not* require an `else` section for every `if`.

Program 4.3 broccoli.cpp

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

// illustrates use of if-else statement

int main()
{
    string response;
    cout << "Do you like broccoli [yes/no]> ";
    cin >> response;
    if ("yes" == response)
    {   cout << "Green vegetables are good for you" << endl;
        cout << "Broccoli is good in stir-fry as well" << endl;
    }
    else
    {   cout << "De gustibus non disputandum" << endl;
        cout << "(There is no accounting for taste)" << endl;
    }
    return 0;
}
```

broccoli.cpp

OUTPUT

```
prompt> broccoli
Do you like broccoli [yes/no]> yes
Green vegetables are good for you
Broccoli is good in stir-fry as well

prompt> broccoli
Do you like broccoli [yes/no]> no
De gustibus non disputandum
(There is no accounting for taste)
```

The `else` section in Program 4.3 could be removed, leaving the following:

```
int main()
{
  string response;
  cout << "Do you like broccoli [yes/no]> ";
  cin >> response;
  if ("yes" == response)
  { cout << "Green vegetables are good for you" << endl;
    cout << "Broccoli is good in stir-fry as well" << endl;
  }
  return 0;
}
```

In this modified program, if the user enters any string other than `"yes"`, nothing is printed.

Program Tip 4.2: Use an `if` statement to guard a sequence of statements and an `if/else` statement to choose between two sequences. The `if` statement solves the problem of guarding the sequence of statements in the body of the `if` so that these statements are executed only when a certain condition holds. The `if/else` statement solves the problem of choosing between two different sequences. Later in the chapter we'll see how to choose between more than two sequences using cascaded `if/else` statements.

4.3 Operators

We've seen arithmetic operators such as `+`, `*`, `%`, the assignment operator `=`, and the `<` operator used in `if/else` statements. In this section we'll study the other operators available in C++. You'll use all these operators in constructing C++ programs.

Table 4.2 The relational operators

symbol	meaning	example
$==$	equal to	`if ("yes" == response)`
$>$	greater than	`if (salary > 30000)`
$<$	less than	`if (0 < salary)`
$!=$	not equal to	`if ("yes" != response)`
$>=$	greater than or equal to $\geq$	`if (salary >= 10000)`
$<=$	less than or equal to $\leq$	`if (20000 <= salary)`

4.3.1 Relational Operators

Comparisons are odious.
John Fortescue
De Laudibus Legum Angliae, 1471

The expressions that form the test of an `if` statement are built from different operators. In this section we'll study the **relational operators,** which are used to determine the relationships between different values. Relational operators are listed in Table 4.2.

The parenthesized expression that serves as the test of an `if` statement can use any of the relational operators shown in Table 4.2. The parenthesized expressions evaluate to true or false and are called **boolean** expressions, after the mathematician George Boole. Boolean expressions have one of two values: **true** or **false.** In C++ programs, any nonzero value is considered "true," and zero-valued expressions are considered "false." The C++ type **bool** is used for variables and expressions with one of two values: *true* and *false*. Although `bool` was first approved as part of C++ in 1994, some older compilers do not support it.[2] We'll use `true` and `false` as values rather than zero and one, but remember that zero is the value used for false in C++ .

The relational operators < and > behave as you might expect when used with `int` and `double` values. In the following statement the variable `salary` can be an `int` or a `double`. In either case the phrase about minimum wage is printed if the value of `salary` is less than 10.0.

[2]If you're using a compiler that doesn't support `bool` as a built-in type, you can use the header file `bool.h` supplied with the code from this book via `#include"bool.h"` to get access to a programmer-defined version of type `bool`.

```
if (salary < 10.0)
{   cout << "you make below minimum wage" << endl;
}
```

When `string` values are compared, the behavior of the inequality operators < and > is based on a dictionary order, sometimes called **lexicographical** order:

```
string word;
cout << "enter a word: ";
cin >> word;
if (word < "middle")
{   cout << word << " comes before middle" << endl;
}
```

In the foregoing code fragment, entering the word `"apple"` generates the following output.

O U T P U T

```
apple comes before middle
```

Entering the word `"zebra"` would cause the test `(word < "middle")` to evaluate to false, so nothing is printed. The comparison of strings is based on the order in which the strings would appear in a dictionary, so that `"A"` comes before `"Z"`. Sometimes the behavior of string comparisons is unexpected. Entering `"Zebra"`, for example, generates this output:

O U T P U T

```
Zebra comes before middle
```

This happens because capital letters come before lower-case letters in the ordering of characters used on most computers.[3]

To see how relational operators are evaluated, consider the output of these statements:

```
cout << (13 < 5) << endl;
cout << (5 + 1 < 6 * 2) << endl;
```

[3]We'll explore the ASCII character set, which is used to determine this ordering, in Chapter 9.

OUTPUT

```
0
1
```

The value of `13 < 5` is false, which is zero; and the value of `6 < 12` is true, which is one. (In Howto B, a standard method for printing bool values as `true` and `false`, rather than 1 and 0, is shown.) In the last output statement, the arithmetic operations are executed first, because they have higher precedence than relational operators. You've seen precedence used with arithmetic operators; for example, multiplication has higher precedence than addition, so that $3 + 4 \times 2 = 11$. You can use parentheses to bypass the normal precedence rules. The expression $(3 + 4) \times 2$ evaluates to 14 rather than 11. A table showing the relative precedence of all C++ operators can be found in Table A.4 in Howto A.

Program Tip 4.3: When you write expressions in C++ programs, use parentheses liberally. Trying to uncover precedence errors in a complex expression can be very frustrating. Looking for precedence errors is often the last place you'll look when trying to debug a program. As part of defensive programming, use parentheses rather than relying exclusively on operator precedence.

Because execution of an `if` statement depends only on whether the test is true or false (nonzero or zero), the following code is legal in C++ :

```
if (6 + 3 - 9)
{   cout << "great minds think alike" << endl;
}
else
{   cout << "fools seldom differ" << endl;
}
```

These statements cause the string "fools seldom differ" to be output, because the expression $(6 + 3 - 9)$ evaluates to 0, which is false in C++ . Although this code is legal, it is not necessarily good code. It is often better to make the comparison explicit, as in

```
if (x != 0)
{   DoSomething();
}
```

rather than relying on the equivalence of "true" and any nonzero value:

```
if (x)
{   DoSomething();
}
```

which is equivalent in effect, but not in clarity. There are situations, however, in which the second style of programming is clearer. When such a situation arises, I'll point it out.

4.3.2 Logical Operators

As we will see (for example, in *usemath.cpp*, Program 4.6, it can be necessary to check that a value you enter is in a certain range (e.g., not negative). In *change.cpp*, Program 4.1, the program should check to ensure that the user's input is valid (e.g., between 0 and 99). The following code implements this kind of check.

```
if (choice < 0)
{   cout << "illegal choice" << endl;
}
else if (choice > 99)
{   cout << "illegal choice" << endl;
}
else
{   // choice ok, continue
}
```

This code has the drawback of duplicating the code that's executed when the user enters an illegal choice. Suppose a future version of the program will require the user to reenter the choice. Modifying this code fragment would require adding new code in two places, making the likelihood of introducing an error larger. In addition, when code is duplicated, it is often difficult to make the same modifications everywhere the code appears. Logical operators allow boolean expressions to be combined, as follows:

```
if (choice < 0 || choice > 99)
{   cout << "illegal choice" << endl;
}
else
{   // choice ok, continue
}
```

The test now reads, "If choice is less than zero or choice is greater than ninety-nine." The test is true (nonzero) when either `choice < 0` or `choice > 99`. The operator `||` is the **logical or** operator. It evaluates to true when either or both of its boolean arguments is true. The **logical and** operator && operates on two boolean expressions and returns true only when both are true.

The preceding test for valid input can be rewritten using logical and as follows:

```
if (0 <= choice && choice <= 99)
{   // choice ok, continue
}
else
{   cout << "illegal choice" << endl;
}
```

Table 4.3 Truth table for logical operators

A	B	A || B	A && B	!A
false	false	false	false	true
false	true	true	false	true
true	false	true	false	false
true	true	true	true	false

Be careful when translating English or mathematics into C++ code. The phrase "choice is between 0 and 99" is often written in mathematics as $0 \le \text{choice} \le 99$. In C++, relational operators are left-associative, so the following `if` test, coded as it would be in mathematics, will evaluate to true for *every* value of `choice`.

```
if (0 <= choice <= 99)
{   // choice ok, continue
}
```

Since the leftmost `<=` is evaluated first (the relational operators, like all binary operators, are left associative), the test is equivalent to `( (0 <= choice) <= 99 )` and the value of the expression `(0 <= choice)` is either false (0) or true (1), both of which are less than or equal to 99, thus satisfying the second test.

There is also a unary operator `!` that works with boolean expressions. This is the **logical not operator.** The value of `!expression` is false if the value of `expression` is true, and true when the value of `expression` is false. The two expressions below are equivalent.

```
          x != y          !(x == y)
```

Because `!` has a very high precedence, the parentheses in the expression on the right are necessary (see Table A.4).

4.3.3 Short-Circuit Evaluation

The following statement is designed to print a message when a grade-point average is higher than 90%:

```
if (scoreTotal/numScores > 0.90)
{    cout << "excellent!  very good work" << endl;
}
```

This code segment might cause a program to exit abnormally[4] if the value of `numScores` is zero, because the result of division by zero is not defined. The abnormal exit can be avoided by using another **nested** `if` statement (the approach required in languages such as Pascal):

[4]The common phrase for such an occurrence is **bomb,** as in "The program bombed." If you follow good defensive programming practices, your programs should not bomb.

```
if (numScores != 0)
{
   if (scoreTotal/numScores > 0.90)
   {    cout << "excellent!  very good work" << endl;
   }
}
```

However, in languages like C, C++, and Java another approach is possible:

```
if (numScores != 0 && scoreTotal/numScores > 0.90)
{    cout << "excellent!  very good work" << endl;
}
```

The subexpressions in an expression formed by the logical operators && and || are evaluated from left to right. Furthermore, the evaluation automatically stops as soon as the value of the entire test expression can be determined. In the present example, if the expression `numScores != 0` is false (so that `numScores` is equal to 0), the entire expression must be false, because when && is used to combine two boolean subexpressions, both subexpressions must be true (nonzero) for the entire expression to be true (see Table 4.3). When `numScores == 0`, the expression `scoreTotal/numScores > 0.90` will *not* be evaluated, avoiding the potential division by zero.

Similarly, when || is used, the second subexpression will not be evaluated if the first is true, because in this case the entire expression must be true—only one subexpression needs to be true for an entire expression to be true with ||. For example, in the code

```
if (choice < 1 || choice > 3)
{    cout << "illegal choice" << endl;
}
```

the expression `choice > 3` is not evaluated when `choice` is 0. In this case, `choice < 1` is true, so the entire expression must be true.

The term **short-circuit evaluation** describes this method of evaluating boolean expressions. The short circuit occurs when some subexpression is not evaluated because the value of the entire expression is already determined. We'll make extensive use of short-circuit evaluation (also called "lazy evaluation") in writing C++ programs.

4.3.4 Arithmetic Assignment Operators

C++ has several operators that serve as "contractions," in the grammatical sense that "I've" is a contraction of "I have." These operators aren't necessary, but they can simplify and shorten code that changes the value of a variable. For example, several statements in *change.cpp,* Program 4.1, alter the value of `amount`; these statements are similar to the following:

```
amount = amount - quarters*25;
```

This statement can be rewritten using the operator `-=`.

```
amount -= quarters*25;
```

Table 4.4 Arithmetic assignment operators.

symbol	example	equivalent
`+=`	`x += 1;`	`x = x + 1;`
`*=`	`doub *= 2;`	`doub = doub * 2;`
`-=`	`n -= 5;`	`n = n - 5;`
`/=`	`third /= 3;`	`third = third / 3;`
`%=`	`odd %= 2;`	`odd = odd % 2;`

Similarly, the statement `number = number + 1`, which increments the value of `number` by one, can be abbreviated using the += operator: `number += 1;`. In general, the statement *variable* = *variable* + *expression*`;` has exactly the same effect as the statement *variable* += *expression*`;`

Using such assignment operators can make programs easier to read. Often a long variable name appearing on both sides of an assignment operator = will cause a lengthy expression to wrap to the next line and be difficult to read. The arithmetic assignment operators summarized in Table 4.4 can alleviate this problem.

It's not always possible to use an arithmetic assignment operator as a contraction when a variable appears on both the left and right sides of an assignment statement. The variable must occur as the *first* subexpression on the right side. For example, if `x` has the value zero or one, the statement `x = 1 - x` changes the value from one to zero and vice versa. This statement cannot be abbreviated using the arithmetic assignment operators.

4.4 Block Statements and Defensive Programming

Following certain programming conventions can lead to programs that are more understandable (for you and other people reading your code) and more easily developed.

In this book we follow the convention of using **block delimiters,** { and }, for each part of an `if`/`else` statement. This is shown in *change2.cpp*, Program 4.2. It is possible to write the `if` statement in Program 4.2 without block delimiters:

```
if (amount > 0)
    cout << "# " << coin << " =\t" << amount << endl;
```

The test of the `if` statement controls the output statement so that it is executed only when `amount` is greater than zero.

As we've seen, it is useful to group several statements together so that all are executed precisely when the test of an `if`/`else` statement is true. To do this, a **block** (or **compound**) **statement** is used. A block statement is a sequence of one or more statements enclosed by curly braces, as shown in Program 4.2. If no braces are used, a program may compile and run, but its behavior might be other than expected. Consider the following program fragment:

```
int salary;
cout << "enter salary ";
cin >> salary;
if (salary > 30000)
    cout << salary << " is a lot to earn " << endl;
    cout << salary*0.55 << " is a lot of taxes << endl;
cout << "enter \# of hours worked ";
...
```

Two sample runs of this fragment follow:

OUTPUT

```
enter salary 31000
31000 is a lot to earn
17050.00 is a lot of taxes
enter # of hours worked
...
enter salary 15000
8250.00 is a lot of taxes
enter # of hours worked
...
```

Stumbling Block

Note that the indentation of the program fragment might suggest to someone reading the program (but not to the compiler!) that the “lot of taxes” message should be printed only when the salary is greater than 30,000. However, the taxation message is *always* printed. The compiler interprets the code fragment as though it were written this way:

```
int salary;
cout << "enter salary ";
cin >> salary;
if (salary > 30000)
{    cout << salary << " is a lot to earn " << endl;
}
cout << salary*0.55 << " is a lot of taxes " << endl;
cout << "enter # of hours worked ";
```

When 15000 is entered, the test `salary > 30000` evaluates to false, and the statement about “a lot to earn” is not printed. The statement about a “lot of taxes”, however, is printed, because it is not controlled by the test.

Indentation and spacing are ignored by the compiler, but they are important for people reading and developing programs. For this reason, we will always employ braces {} and a block statement when using `if`/`else` statements, even if the block statement consists of only a single statement.

June 7, 1999 10:10 owltex Sheet number 34 Page number 116 magenta black

4.4.1 Defensive Programming Conventions

This convention of always using braces is an example of a **defensive programming** strategy: writing code to minimize the potential for causing errors. Suppose you decide to add another statement to be controlled by the test of an `if` statement that is initially written without curly braces. When the new statement is added, it will be necessary to include the curly braces that delimit a block statement, but that is easy to forget to do. Since the missing braces can cause a hard-to-detect error, we adopt the policy of including them even when there is only a single statement controlled by the test.

Program Tip 4.4: Adopt coding conventions that make it easier to modify programs. You'll rarely get a program right the first time. Once a program works correctly, it's very likely that you'll need to make modifications so that the program works in contexts unanticipated when first designed and written. More time is spent modifying programs than writing them first, so strive to make modification as simple as possible.

Program 4.4 noindent.cpp

```
#include <iostream>
#include <string>
using namespace std;
int main() { string response; cout
<< "Do you like C++ programming [yes/no]> "; cin >> response;
        if ("yes" == response) { cout <<
"It's more than an adventure, it can be a job"
                     << endl; } else {  cout
<< "Perhaps in time you will" << endl; } return     0;}
```

noindent.cpp

In this book the left curly brace { always follows an `if`/`else` on the next line after the line on which the `if` or `else` occurs. The right curly brace } is indented the same level as the `if`/`else` to which it corresponds. Other indentation schemes are possible; one common convention follows, this is called *K&R* style after the originators of C, Kernighan and Ritchie.

```
 if ("yes" == response) {
     cout << "Green vegetables are good for you" << endl;
     cout << "Broccoli is good in stir-fry as well" << endl;
}
```

You can adopt either convention, but your boss (professor, instructor, etc.) may require a certain style. If you're consistent, the particular style isn't that important, although it's often the cause of many arguments between supporters of different indenting styles.

In this book we usually include the first statement between curly braces on the same line as the first (left) brace. If you use this style of indenting, you will not press return after you type the left curly brace `{`. However, we sometimes do press return, which usually makes programs easier to read because of the extra white space[5].

To see that indentation makes a difference, note that *noindent.cpp*, Program 4.4, compiles and executes without error but that no consistent indentation style is used. Notice that the program is much harder for people to read, although the computer "reads" it with no trouble.

Stumbling Block

Problems with = and ==. Typing = when you mean to type == can lead to hard-to-locate bugs in a program. A coding convention outlined here can help to alleviate these bugs, but you must keep the distinction between = and == in mind when writing code. Some compilers are helpful in this regard and issue warnings about "potentially unintended assignments."

The following program fragment is intended to print a message depending on a person's age:

```
string age;
cout << "are you young or old [young/old]: ";
cin >> age;
if (age = "young")
{ cout << "not for long, time flies when you're having fun";
}
else
{ cout << "hopefully you're young at heart";
}
```

If the user enters `old`, the message beginning "not for long…" is printed. Can you see why this is the case? The test of the `if`/`else` statement should be read as "if age gets young." The string literal `"young"` is assigned to `age`, and the result of the assignment is nonzero (it is `"young"`, the value assigned to `age`). Because anything nonzero is regarded as true, the statement within the scope of the `if` test is executed.

You can often prevent such errors by putting constants on the left of comparisons as follows:

```
if ("young" == age)
  // do something
```

If the assignment operator is used by mistake, as in `if ("young" = age)`, the compiler will generate an error.[6] It is much better to have the compiler generate an error message than to have a program with a bug in it.

Putting constants on the left in tests is a good defensive programming style that can help to trap potential bugs and eliminate them before they creep into your programs.

[5]In a book, space is more of a premium than it is on disk—hence the style of indenting that does not use the return. You should make sure you follow the indenting style used by your boss, supervisor, or programming role model.

[6]On one compiler the error message "error assignment to constant" is generated. On another, the less clear message "sorry, not implemented: initialization of array from dissimilar array type" is generated.

4.4.2 Cascaded if/else Statements

Sometimes a sequence of `if`/`else` statements is used to differentiate among several possible values of a single expression. Such a sequence is called **cascaded.** An example is shown in *monthdays.cpp*, Program 4.5.

Program 4.5 monthdays.cpp

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

// illustrates cascaded if/else statements

int main()
{
    string month;
    int days = 31;             // default value of 31 days/month

    cout << "enter a month (lowercase letters): ";
    cin >> month;

    // 30 days hath september, april, june, and november

    if ("september" == month)
    {    days = 30;
    }
    else if ("april" == month)
    {    days = 30;
    }
    else if ("june" == month)
    {    days = 30;
    }
    else if ("november" == month)
    {    days = 30;
    }
    else if ("february" == month)
    {    days = 28;
    }
    cout << month << " has " << days << " days" << endl;

    return 0;
}
```

monthdays.cpp

It's possible to write the code in *monthdays.cpp* using **nested** `if`/`else` statements as follows. This results in code that is much more difficult to read than code using cascaded `if`/ `else` statements. Whenever a sequence of `if`/`else` statements like this is used to test the value of one variable repeatedly, we'll use cascaded `if`/`else` statements. The rule of using a block statement after an `else` is not (strictly speaking) followed, but the code is much easier to read. Because a block statement follows the `if`, we're not violating the spirit of our coding convention.

```
if ("april" == month)
{    days = 30;
}
else
{
    if ("june" == month)
    {    days = 30;
    }
    else
    {
        if ("november" == month)
        {    days = 30;
        }
        else
        {
            if ("february" == month)
            {    days = 28;
            }
        }
    }
}
```

OUTPUT

```
prompt> days4
enter a month (lowercase letters): january
january has 31 days
prompt> days4
enter a month (lowercase letters): april
april has 30 days
prompt> days4
enter a month (lowercase letters): April
April has 31 days
```

Pause to Reflect

4.1 The statements altering `amount` in *change.cpp*, Program 4.1, can be written using the mod operator `%`. If `amount = 38`, then `amount/25 == 1`, and `amount % 25 == 13`, which is the same value as `38 - 25*1`. Rewrite the program using the mod operator. Try to use an arithmetic assignment operator.

4.2 Describe the output of Program 4.3 if the user enters the string `"Yes"`, the string `"yup"`, or the string `"none of your business"`.

4.3 Why is `days` given a “default” value of 31 in *monthdays.cpp*, Program 4.5?

4.4 How can *monthdays.cpp*, Program 4.5, be modified to take leap years into account?

4.5 Modify *broccoli.cpp*, Program 4.3, to include an `if` statement in the `else` clause so that the "taste" lines are printed only if the user enters the string `"no"`. Thus you might have lines such as

```
if ("yes" == response)
{
}
else if ("no" == response)
{
}
```

4.6 Using the previous modification, add a final `else` clause (with no `if` statement) so that the output might be as follows:

O U T P U T

```
prompt> broccoli
Do you like broccoli [yes/no]> no
De gustibus non disputandum
(There is no accounting for good taste)
prompt> broccoli
Do you like broccoli [yes/no]> nope
Sorry, only responses of yes and no are recognized
```

4.7 Write a sequence of `if`/`else` statements using > and, perhaps, < that prints a message according to a grade between 0 and 100, entered by the user. For example, high grades might get one message and low grades might get another message.

4.8 Explain why the output of the first statement below is 0, but the output of the second is 45:

```
cout << (9 * 3 < 4 * 5) << endl;
cout << (9 * (3 < 4) * 5) << endl;
```

Why are the parentheses needed?

4.9 What is output by each of the following statements (why?)

```
cout << (9 * 5 < 45)     << endl;
cout << (9*5 < 45 < 30) << endl;
```

4.10 Write a code fragment in which a `string` variable `grade` is assigned one of three states: `"High Pass"`, `"Pass"`, and `"Fail"` according to whether an input integer grade is between 80 and 100, between 60 and 80, or below 60, respectively. It may be useful to write the fragment so that a message is printed and then modify it so that a `string` variable is assigned a value.

Stumbling Block

The Dangling Else Problem. Using the block delimiters { and } in all cases when writing `if/else` statements can prevent errors that are very difficult to find because the indentation, which conveys meaning to a reader of the program, is ignored by the compiler when code is generated. Using block delimiters also helps in avoiding a problem that results from a potential ambiguity in computer languages such as C++ that use `if/else` statements (C and Pascal have the same ambiguity, for example).

The following code fragment attempts to differentiate odd numbers less than zero from other numbers. The indentation of the code conveys this meaning, but the code doesn't execute as intended:

```
if (x % 2 == 1)
   if (x < 0)
      cout << " number is odd and less than zero" << endl;
else
   cout << " number is even " << endl;
```

What happens if the `int` object `x` has the value 13? The indentation seems to hint that nothing will be printed. In fact, the string literal `"number is even"` will be printed if this code segment is executed when `x` is 13. The segment is read by the compiler as though it is indented as follows:

```
if (x % 2 == 1)
   if (x < 0)
      cout << " number is odd and less than zero" << endl;
   else
      cout << " number is even " << endl;
```

The use of braces makes the intended use correspond to what happens. Nothing is printed when `x` has the value 13 in

```
if (x % 2 == 1)
{   if (x < 0)
      cout << " number is odd and less than zero" << endl;
}
else
{   cout << " number is even " << endl;
}
```

As we have noted before, the indentation used in a program is to assist the human reader. The computer doesn't require a consistent or meaningful indentation scheme. Misleading indentation can lead to hard-to-find bugs where the human sees what is intended rather than what exists.

One rule to remember from this example is that an `else` always corresponds to the most recent `if`. Without this rule there is ambiguity as to which `if` the `else` belongs; this is known as the **dangling-else** problem. Always employ curly braces { and } when using block statements with `if`/`else` statements (and later with looping constructs). If braces are always used, there is no ambiguity, because the braces serve to delimit the scope of an `if` test.

Claude Shannon *(b. 1916)*

Claude Shannon founded **information theory**—a subfield of computer science that is used today in developing methods for encrypting information. Encryption is used to store data in a secure manner so that the information can be read only by designated people.

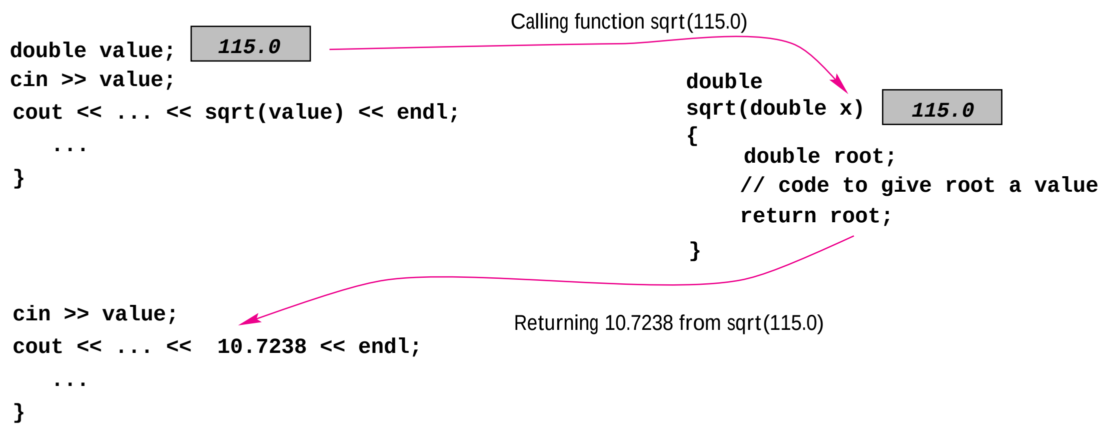
In his 1937 master's thesis, Shannon laid the foundation on which modern computers are built by equating Boolean logic with electronic switches. This work enabled hardware designers to design, build, and test circuits that could perform logical as well as arithmetic operations. In an interview in [Hor92], Shannon responds to the comment that his thesis is "possibly the most important master's thesis in the century" with "It just happened that no one else was familiar with both those fields at the same time." He then adds a wonderful non sequitur: "I've always loved that word 'Boolean.' " Shannon is fond of juggling and riding unicycles. Among his inventions are a juggling "dummy" that looks like W.C. Fields and a computer THROBAC: Thrifty Roman Numeral Backward Computer.

Although much of Shannon's work has led to significant advances in the theory of communication, he says:

> *I've always pursued my interests without much regard for financial value or the value to the world; I've spent lots of time on totally useless things.*

Shannon's favorite food is vanilla ice cream with chocolate sauce.

Shannon received the National Medal of Science in 1966. For more information see [Sla87, Hor92].



Calling function sqrt(115.0)
115.0
115.0
Returning 10.7238 from sqrt(115.0)

```
double value;
cin >> value;
cout << ... << sqrt(value) << endl;
   ...
}
```

```
double
sqrt(double x)
{
    double root;
    // code to give root a value
    return root;
}
```

```
cin >> value;
cout << ... <<  10.7238 << endl;
   ...
}
```

Figure 4.2 Evaluating the function call `sqrt115.0`

4.5 Functions That Return Values

Civilization advances by extending the number
of important operations which we can perform without thinking about them.
Alfred North Whitehead
An Introduction to Mathematics

In Chapter 3 we studied programmer-defined functions, such as `SlicePrice` in *pizza.cpp*, Program 3.5, whose prototype is

```
void SlicePrice(int radius, double price)
```

The return type of `SlicePrice` is `void`. Many programs require functions that have other return types. You've probably seen mathematical functions on hand-held calculators such as $\sin(x)$ or $\sqrt{x}$. These functions are different from the function `SlicePrice` in that they return a value. For example, when you use a calculator, you might enter the number 115, then press the square-root key. This displays the value of $\sqrt{115}$ or 10.7238. The number 115 is an **argument** to the square root function. The value returned by the function is the number 10.7238. Program 4.6 is a C++ program that processes information in the same way: users enter a number, and the square root of the number is displayed.

Control flow from *usemath.cpp* is shown in Fig. 4.2. The value 115, entered by the user and stored in the variable `value`, is copied into a memory location associated with the parameter `x` in the function `sqrt`. The square root of 115 is calculated, and a **return** statement in the function `sqrt` returns this square root, which is used in place of the expression `sqrt(value)` in the `cout` statement. As shown in Fig. 4.2, the value 10.7238 is displayed as a result.

The function `sqrt` is accessible by including the header file `<cmath>`. Table 4.5 lists some of the functions accessible from this header file. A more complete table of functions is given as Table F.1 in Howto F. In the sample output of *usemath.cpp*, Program 4.6, the square roots of floating-point numbers aren't always exact. For example, $\sqrt{100.001} = 10.0000499998$, but the value displayed is 10. Floating-point values cannot always be exactly determined. Because of inherent limits in the way these values are stored in the computer, the values are rounded off to the most precise values that can be represented in the computer. The resulting **roundoff error** illustrates the theme of *conceptual* and *formal* models introduced in Chapter 1. Conceptually, the square root of 100.001 can be calculated with as many decimal digits as we have time or inclination to write down. In the formal model of floating-point numbers implemented on computers, the precision of the calculation is limited.

Program 4.6 usemath.cpp

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

// illustrates use of math function returning a value

int main()
{
    double value;
    cout << "enter a positive number ";
    cin >> value;
    cout << "square root of " << value << " = " << sqrt(value) << endl;

    return 0;
}
```

usemath.cpp

O U T P U T

```
prompt> usemath
enter a positive number 115
square root of 115 = 10.7238
prompt> usemath
enter a positive number 100.001
square root of 100.001 = 10
prompt> usemath
enter a positive number -16
square root of -16 = nan
```

Table 4.5 Some functions in `<cmath>`

function name	prototype	returns
`double fabs`	`(double x)`	absolute value of x
`double log`	`(double x)`	natural log of x
`double log10`	`(double x)`	base-ten log of x
`double sin`	`(double x)`	sine of x (x in radians)
`double cos`	`(double x)`	cosine of x (x in radians)
`double tan`	`(double x)`	tangent of x (x in radians)
`double asin`	`(double x)`	arc sine of x $[-\pi/2, \pi/2]$
`double acos`	`(double x)`	arc cosine of x $[0, \pi]$
`double atan`	`(double x)`	arc tangent of x $[-\pi/2, \pi/2]$
`double pow`	`(double x, double y)`	x^y
`double sqrt`	`(double x)`	$\sqrt{x}$, square root of x
`double floor`	`(double x)`	largest integer value $\leq$ x
`double ceil`	`(double x)`	smallest integer value $\geq$ x

Finally, although the program prompts for positive numbers, there is no check to ensure that the user has entered a positive number. In the output shown, the symbol `nan` stands for "not a number."[7] Not all compilers will display this value. In particular, on some computers, trying to take the square root of a negative number may cause the machine to lock up. It would be best to guard the call `sqrt(value)` using an `if` statement such as the following one:

```
if (0 <= value)
{   cout << "square root of " << value << " = "
         << sqrt(value) << endl;
}
else
{   cout << "nonpositive number " << value
         << " entered" << endl;
}
```

Alternatively, we could **compose** the function `sqrt` with the function `fabs`, which computes absolute values.

```
cout << "square root of " << value << " = "
     << sqrt(fabs(value)) << endl;
```

The result returned by the function `fabs` is used as an argument to `sqrt`. Since the return type of `fabs` is `double` (see Table 4.5), the argument of `sqrt` has the right type.

[7]Some compilers print `NaN`, others crash rather than printing an error value.

4.5.1 The Math Library `<cmath>`

In C and C++ several mathematical functions are available by accessing a math library using `#include <cmath>`.[8] Prototypes for some of these functions are listed in Table 4.5, and a complete list is given as Table F.1 in Howto F.

All of these functions return `double` values and have `double` parameters. Integer values can be converted to `double`s, so the expression `sqrt(125)` is legal (and evaluates to 11.18033). The function `pow` is particularly useful, because there is no built-in exponentiation operator in C++. For example, the statement

```
cout << pow(3,13) << endl}
```

outputs the value of 3^{13}: three to the thirteenth.

The functions declared in `<cmath>` are tools that can be used in any program. As programmers, we'll want to develop functions that can be used in the same way. On occasion, we'll develop functions that aren't useful as general-purpose tools but make the development of one program simpler. For example, in *pizza.cpp*, Program 3.5, the price per square inch of pizza is calculated and printed by the function `SlicePrice`. If the value were returned by the function rather than printed, it could be used to determine which of several pizzas was the best buy. This is shown in *pizza2.cpp*, Program 4.7. Encapsulating the calculation of the price per square inch in a function, as opposed to using the expression `smallPrice/(3.14159 * smallRadius * smallRadius)`, avoids errors that might occur in copying or retyping the expression for a large pizza. Using a function also makes it easier to include other sizes of pizza in the same program. If it develops that we've made a mistake in calculating the price per square inch, isolating the mistake in one function makes it easier to change than finding all occurrences of the calculation and changing each one.

Program 4.7 pizza2.cpp

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

// find the price per square inch of pizza
// to compare large and small sizes for the best value
//
// Owen Astrachan
// March 29, 1999
//

double Cost(double radius, double price)
// postcondition: returns the price per sq. inch
{
    return price/(3.14159∗radius∗radius);
}
```

[8]The name `cmath` is the C++ math library, but with many older compilers you will need to use `math.h` rather than `cmath`.

```
int main()
{
    double smallRadius, largeRadius;
    double smallPrice, largePrice;
    double smallCost,largeCost;

    // input phase of computation

    cout << "enter radius and price of small pizza ";
    cin >> smallRadius >> smallPrice;

    cout << "enter radius and price of large pizza ";
    cin >> largeRadius >> largePrice;


    // process phase of computation

    smallCost = Cost(smallRadius,smallPrice);
    largeCost = Cost(largeRadius,largePrice);

    // output phase of computation

    cout << "cost of small pizza = " << smallCost << " per sq.inch" << endl;
    cout << "cost of large pizza = " << largeCost << " per sq.inch" << endl;

    if (smallCost < largeCost)
    {   cout << "SMALL is the best value " << endl;
    }
    else
    {   cout << "LARGE is the best value " << endl;
    }

    return 0;
}
```

pizza2.cpp

OUTPUT

```
prompt> pizza2
enter radius and price of small pizza 6 6.99
enter radius and price of large pizza 8 10.99
cost of small pizza = 0.0618052 per sq.inch
cost of large pizza = 0.0546598 per sq.inch
LARGE is the best value
```

From the user's point of view, Program 3.5 and Program 4.7 exhibit similar, though not identical, behavior. When two programs exhibit identical behavior, we describe this sameness by saying that the programs are identical as **black boxes.** We cannot see the inside of a black box; the behavior of the box is discernible only by putting values into the box (running the program) and noting what values come out (are printed by the program). A black box specifies input and output, but not how the processing step takes place. The `balloon` class and the math function `sqrt` are black boxes; we don't know how they are implemented, but we can use them in programs by understanding their input and output behavior.

4.5.2 Pre- and Post-conditions

In the function `SlicePrice` of *pizza2.cpp*, Program 4.7, a comment is given in the form of a **postcondition.** A postcondition of a function is a statement that is true when the function finishes executing. Each function in this book will include a postcondition that describes what the function does. Some functions have **preconditions**. A precondition states what parameter values can be passed to the function. Together a function's precondition and postcondition provide a contract for programmers who call the function: if the precondition is true the postcondition will be true. For example, a precondition of the function `sqrt` might be that the function's parameter is non-negative.

Program Tip 4.5: When calling functions, read postconditions carefully. When writing functions, provide postconditions. When possible, provide a precondition as well as a postcondition since preconditions provide programmers with information about what range of values can be passed to each parameter of a function.

In the `main` function of *pizza2.cpp*, the extraction operator `>>` extracts two values in a single statement. Just as the insertion operator `<<` can be used to put several items on the output stream `cout`, the input stream `cin` continues to flow so that more than one item can be extracted.

Pause to Reflect

4.11 Write program fragments or complete programs that convert degrees Celsius to degrees Fahrenheit, British thermal units (Btu) to joules (J), and knots to miles per hour. Note that x degrees Celsius equals $(9/5)x + 32$ degrees Fahrenheit; that x J equals $9.48 \times 10^{-4}(x)$Btu; and that 1 knot $= 101.269$ ft/min (and that 5,280 ft $=$ 1 mile). At first do this *without* using assignment statements, by incorporating the appropriate expressions in output statements. Then define variables and use assignment statements as appropriate. Finally, write functions for each of the conversions.

4.12 Modify *pizza2.cpp*, Program 4.7, to use the function `pow` to square `radius` in the function `Cost`.

4.13 If a negative argument to the function `sqrt` causes an error, for what values of `x` does the following code fragment generate an error?

```
if (x >= 0 && sqrt(x) > 100)
    cout << "big number" << endl;
```

4.14 Heron's formula gives the area of a triangle in terms of the lengths of the sides of the triangle: `a`, `b`, and `c`.

$$\text{area} = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)} \qquad (4.1)$$

where s is the semiperimeter, or half the perimeter $a+b+c$ of the triangle. Write a function `TriangleArea` that returns the area of a triangle. The sides of the triangle should be parameters to `TriangleArea`.

4.15 The law of cosines gives the length of one side of a triangle, c, in terms of the other sides a and b and the angle C formed by the sides a and b:

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cos(C)$$

Write a function `SideLength` that computes the length of one side of a triangle, given the other two sides and the angle (in radians) between the sides as parameters to `SideLength`.

4.16 The following code fragment allows a user to enter three integers:

```
int a,b,c;
cout << "enter three integers ";
cin >> a >> b >> c;
```

Add code that prints the average of the three values read. Does it make a difference if the type is changed from `int` to `double`? Do you think that `>>` has the same kind of associativity as `=`, the assignment operator?

4.5.3 Function Return Types

The functions `sqrt` and `SlicePrice` used in previous examples both returned `double` values. In this section we'll see that other types can be returned.

Determining Leap Years. Leap years have an extra day (February 29) not present in nonleap years. We use arithmetic and logical operators to determine whether a year is a leap year. Although it's common to think that leap years occur every four years, the rules for determining leap years are somewhat more complicated, because the period of the Earth's rotation around the Sun is not exactly 365.25 days but approximately 365.2422 days.

- If a year is evenly divisible by 400, then it is a leap year.
- Otherwise, if a year is divisible by 100, then it is *not* a leap year.
- The only other leap years are evenly divisible by 4.[9]

[9]These rules correspond to a year length of 365.2425 days. In the *New York Times* of January 2, 1996 (page B7, out-of-town edition), a correction to the rules used here is given. The year 4000 is *not* a leap year, nor will any year that's a multiple of 4000 be a leap year. Apparently this rule, corresponding to a year length of 365.24225 days, will have to be modified too, but we probably don't need to worry that our program will be used beyond the year 4000.

For example, 1992 is a leap year (it is divisible by 4), but 1900 is not a leap year (it is divisible by 100), yet 2000 is a leap year, because, although it is divisible by 100, it is also divisible by 400.

The boolean-valued function `IsLeapYear` in Program 4.8 uses multiple `return` statements to implement this logic.

Recall that in the expression `(a % b)` the modulus operator `%` evaluates to the remainder when `a` is divided by `b`. Thus, `2000 % 400 == 0`, since there is no remainder when 2000 is divided by 400.

The sequence of cascaded `if` statements in `IsLeapYear` tests the value of the parameter `year` to determine whether it is a leap year. Consider the first run shown, when `year` has the value 1996. The first test, `year % 400 == 0`, evaluates to false, because 1996 is not divisible by 400. The second test evaluates to false, because 1996 is not divisible by 100. Since $1996 = 4 \times 499$, the third test, `(year % 4 == 0)`, is true, so the value `true` is returned from the function `IsLeapYear`. This makes the expression `IsLeapYear(1996)` in `main` true, so the message is printed indicating that 1996 is a leap year. You may be tempted to write

```
if (IsLeapYear(year) == true)
```

rather than using the form shown in *isleap.cpp*. This works, but the `true` is redundant, because the function `IsLeapYear` is boolean-valued: it is either true or false.

The comments for the function `IsLeapYear` are given in the form of a **precondition** and a postcondition. For our purposes, a precondition is what must be satisfied for the function to work as intended. The "as intended" part is what is specified in the postcondition. These conditions are a *contract* for the caller of the function to read: if the precondition is satisfied, the postcondition will be satisfied. In the case of `IsLeapYear` the precondition states that the function works for any year greater than 0. The function is *not* guaranteed to work for the year 0 or if a negative year such as -10 is used to indicate the year 10 B.C.

It is often possible to implement a function in many ways so that its postcondition is satisfied. Program 4.9 shows an alternative method for writing `IsLeapYear`. Using a black-box test, this version is indistinguishable from the `IsLeapYear` used in Program 4.8.

Program 4.8 isleap.cpp

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

// illustrates user-defined function for determining leap years

bool IsLeapYear(int year)
// precondition: year > 0
// postcondition: returns true if year is a leap year, else returns false
{
    if (year % 400 == 0)         // divisible by 400
    {   return true;
    }
    else if (year % 100 == 0)    // divisible by 100
    {   return false;
    }
    else if (year % 4 == 0)      // divisible by 4
    {   return true;
    }
    return false;
}

int main()
{
    int year;
    cout << "enter a year ";
    cin >> year;
    if (IsLeapYear(year))
    {   cout << year << " has 366 days, it is a leap year" << endl;
    }
    else
    {   cout << year << " has 365 days, it is NOT a leap year" << endl;
    }
    return 0;
}
```

isleap.cpp

OUTPUT

```
prompt> isleap
enter a year 1996
1996 has 366 days, it is a leap year

prompt> isleap
enter a year 1900
1900 has 365 days, it is NOT a leap year
```

Program 4.9 isleap2.cpp

```
bool IsLeapYear(int year)
// precondition: year > 0
// postcondition: returns true  if year is a leap year, else false
{
    return (year % 400 == 0) || ( year % 4 == 0 && year % 100 != 0 );
}
```

isleap2.cpp

A boolean value is returned from `IsLeapYear` because the logical operators && and `||` return boolean values. For example, the expression `IsLeapYear(1974)` causes the following expression to be evaluated by substituting 1974 for `year`:

```
(1974 % 400 == 0) || ( 1974 % 4 == 0 && 1974 % 100 != 0 );
```

Since the logical operators are evaluated left to right to support short-circuit evaluation, the subexpression `1974 % 400 == 0` is evaluated first. This subexpression is false, because `1974 % 400` is 374. The rightmost parenthesized expression is then evaluated, and its subexpression `1974 % 4 == 0` is evaluated first. Since this subexpression is false, the entire && expression must be false (why?), and the expression `1974 % 100 != 0` is not evaluated. Since both subexpressions of `||` are false, the entire expression is false, and `false` is returned.

Boolean-valued functions such as `IsLeapYear` are often called **predicates.** Predicate functions often begin with the prefix `Is`. For example, the function `IsEven` might be used to determine whether a number is even; the function `IsPrime` might be used to determine whether a number is prime (divisible by only 1 and itself, e.g., 3, 17); and the function `IsPalindrome` might be used to determine whether a word is a palindrome (reads the same backward as forward, e.g., mom, racecar).

Program Tip 4.6: Follow conventions when writing programs. Conventions make it easier for other people to read and use your programs and for you to read them long after you write them. One common convention is using the prefix `Is` for predicate/boolean-valued functions.

Converting Numbers to English. We'll explore a program that converts some integers to their English equivalent. For example, 57 is "fifty-seven" and 14 is "fourteen." Such a program might be the basis for a program that works as a talking cash register, speaking the proper coins to give as change. With speech synthesis becoming cheaper on computers, it's fairly common to encounter a computer that "speaks." The number you hear after dialing directory assistance is often spoken by a computer. There are many home finance programs that print checks; these programs employ a method of converting numbers to English to print the checks. In addition to using arithmetic operators, the program shows that functions can return strings as well as numeric and boolean types, and it emphasizes the importance of pre- and postconditions.

Program 4.10 numtoeng.cpp

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

// converts two digit numbers to English equivalent
// Owen Astrachan, 3/30/99

string DigitToString(int num)
// precondition: 0 <= num < 10
// postcondition: returns english equivalent, e.g., 1->one,...9->nine
```

```
{
    if (0 == num)       return "zero";
    else if (1 == num)  return "one";
    else if (2 == num)  return "two";
    else if (3 == num)  return "three";
    else if (4 == num)  return "four";
    else if (5 == num)  return "five";
    else if (6 == num)  return "six";
    else if (7 == num)  return "seven";
    else if (8 == num)  return "eight";
    else if (9 == num)  return "nine";
    else return "?";
}

string TensPrefix(int num)
// precondition: 10 <= num <= 99 and num % 10 == 0
// postcondition: returns ten, twenty, thirty, forty, etc.
//                corresponding to num, e.g., 50->fifty
{
    if (10 == num) return "ten";
    else if (20 == num) return "twenty";
    else if (30 == num) return "thirty";
    else if (40 == num) return "forty";
    else if (50 == num) return "fifty";
    else if (60 == num) return "sixty";
    else if (70 == num) return "seventy";
    else if (80 == num) return "eighty";
    else if (90 == num) return "ninety";
    else return "?";
}

string TeensToString(int num)
// precondition: 11 <= num <= 19
// postcondition: returns eleven, twelve, thirteen, fourteen, etc.
//                corresponding to num, e.g., 15 -> fifteen
{
    if (11 == num) return "eleven";
    else if (12 == num) return "twelve";
    else if (13 == num) return "thirteen";
    else if (14 == num) return "fourteen";
    else if (15 == num) return "fifteen";
    else if (16 == num) return "sixteen";
    else if (17 == num) return "seventeen";
    else if (18 == num) return "eighteen";
    else if (19 == num) return "nineteen";
    else return "?";
}

string NumToString(int num)
// precondition: 0 <= num <= 99
// postcondition: returns english equivalent, e.g., 1->one, 13->thirteen
{
    if (0 <= num && num < 10)
    {   return DigitToString(num);
    }
    else if (10 < num && num < 20)
    {   return TeensToString(num);
    }
    else if (num % 10 == 0)
    {   return TensPrefix(num);
    }
    else
    {   // concatenate ten's digit with one's digit
        return TensPrefix(10 * (num/10)) + "-" + DigitToString(num % 10);
    }
}

int main()
{
    int number;
    cout << "enter number between 0 and 99: ";
    cin >> number;
    cout << number  << " = " << NumToString(number) << endl;
    return 0;
}
```

numtoeng.cpp

OUTPUT

```
prompt> numtoeng
enter number between 0 and 99: 22
22 = twenty-two

prompt> numtoeng
enter number between 0 and 99: 17
17 = seventeen

prompt> numtoeng
enter number between 0 and 99: 103
103 = ?-three
```

The code in the `DigitToString` function does not adhere to the rule of using block statements in every `if`/`else` statement. In this case, using {} delimiters would make the program unnecessarily long. It is unlikely that statements will be added (necessitating the use of a block statement), and the form used here is clear.

Program Tip 4.7: White space usually makes a program easier to read and clearer. Block statements used with if/else statements usually make a program more robust and easier to change. However, there are occasions when these rules are not followed. As you become a more practiced programmer, you'll develop your own aesthetic sense of how to make programs more readable.

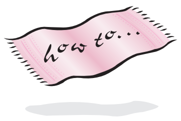
A new use of the operator + is shown in function NumToString. In the final else statement, three strings are joined together using the + operator:

```
return TensPrefix(10*(num/10))+ "-" + DigitToString(num%10);
```

When used with string values, the + operator joins or **concatenates** (sometimes "catenates") the string subexpressions into a new string. For example, the value of "apple" + "sauce" is a new string, "applesauce". This is another example of operator overloading; the + operator has different behavior for string, double, and int values.

Robust Programs. In the sample runs shown, the final input of 103 does not result in the display of one hundred three. The value of 103 violates the precondition of NumToString, so there is no guarantee that the postcondition will be satisfied. **Robust** programs and functions do not bomb in this case, but either return some value that indicates an error or print some kind of message telling the user that input values aren't valid. The problem occurs in this program because "?" is returned by the function call TensPrefix(10 * (num/10)). The value of the argument to TensPrefix is $10 \times (103/10) == 10 \times 10 == 100$. This value violates the precondition of TensPrefix. If no final else were included to return a question mark, then nothing would be returned from the function TensPrefix when it was called with 103 as an argument. This situation makes the concatenation of "nothing" with the hyphen and the value returned by DigitToString(num % 10) problematic, and the program would terminate, because there is no string to join with the hyphen.

Many programs like *numtoeng.cpp* prompt for an input value within a range. A function that ensures that input is in a specific range by reprompting would be very useful. A library of three related functions is specified in *prompt.h*. We'll study these functions in the next chapter, and you can find information about them in Howto G. Here is a modified version of main that uses PromptRange:



```
int main()
{
   int number = PromptRange("enter a number",0,99);
   cout << number << " = " << NumToString(number) << endl;

   return 0;
}
```

O U T P U T

```
prompt> numtoeng
enter number between 0 and 99: 103
enter a number between 0 and 99: 100
enter a number between 0 and 99: -1
enter a number between 0 and 99: 99
99 = ninety-nine
```

You don't have enough programming tools to know how to write `PromptRange` (you need loops, studied in the next chapter), but the specifications of each function make it clear how the functions are called. You can treat the functions as black boxes, just as you treat the square-root function *sqrt* in `<cmath>` as a black box.

Pause to Reflect

4.17 Write a function `DaysInMonth` that returns the number of days in a month encoded as an integer with 1 = January, 2 = February,…, 12 = December. The year is needed, because the number of days in February depends on whether the year is a leap year. In writing the function, you can call `IsLeapYear`. The specification for the function is

```
int DaysInMonth(int month,int year)
// pre: month coded as: 1 = january, ..., 12 = december
// post: returns # of days in month in year
```

4.18 Why are parentheses needed in the expression `TensPrefix(10*(num/10))`? For example, if `TensPrefix(10*num/10)` is used, the program generates a non-number when the user enters 22.

4.19 Write a predicate function `IsEven` that evaluates to true if its `int` parameter is an even number. The function should work for positive and negative integers. Try to write the function using only one statement: `return` *expression*.

4.20 Write a function `DayName` whose header is

```
string DayName(int day)
// pre: 0 <= day <= 6
// post: returns string representing day, with
//       0 = "Sunday", 1 = "Monday", ..., 6 = "Saturday"
```

so that the statement `cout << DayName(3) << endl;` prints `Wednesday`.

4.21 Describe how to modify the function `NumToString` in *numtoeng.cpp*, Program 4.10, so that it works with three-digit numbers.

4.22 An Islamic year y is a leap year if the remainder, when $11y + 14$ is divided by 30, is less than 11. In particular, the 2nd, 5th, 7th, 10th, 13th, 16th, 18th, 21st, 24th, 26th, and 29th years of a 30-year cycle are leap years. Write a function `IsIslamicLeapYear` that works with this definition of leap year.

4.23 In the Islamic calendar [DR90] there are also 12 months, which strictly alternate between 30 days (odd-numbered months) and 29 days (even-numbered months), except for the twelfth month, *Dhu al-Hijjah*, which in leap years has 30 days. Write a function `DaysInIslamicMonth` for the Islamic calendar that uses only three `if` statements.

4.6 Class Member Functions

Section 3.4 discusses a class named `Balloon` for simulating hot-air balloons. We've used the class `string` extensively in our examples, but we haven't used all of the functionality provided by strings themselves. Recall from Section 3.4 that functions provided by a class are called **member functions**. In this section we'll study three `string` member functions, and many more are explained in Howto C. We'll also have a sneak preview at the class `Date` which is covered more extensively in the next chapter. We'll show just one example program using the class, but the program provides a glimpse of the power that classes bring to programming.

4.6.1 `string` Member Functions

The functions we've studied so far, like those in `<cmath>`, are called **free functions** because they do not belong to a class. Member functions are part of a class and are invoked by applying a function to an object with the dot operator. Program 4.11 shows the `string` member functions `length` and `substr`. The function `length` returns the number of characters in a string, the function `substr` returns a *substring* of a string given a starting position and a number of characters.

Program 4.11 strdemo.cpp

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

// illustrates string member functions length() and substr()

int main()
{
    string s;
    cout << "enter string: ";
    cin >> s;
    int len = s.length();
    cout << s << " has " << len << " characters" << endl;
    cout << "first char is " << s.substr(0, 1)   << endl;
    cout << "last char is  " << s.substr(s.length()-1, 1) << endl;
    cout << endl << "all but first is " << s.substr(1,s.length()) << endl;
    return 0;
}
```

strdemo.cpp

OUTPUT

```
prompt> strdemo
enter string: theater
theater has 7 characters
first char is t
last char is r
all but first is heater
prompt> strdemo
enter string: slaughter
theater has 9 characters
first char is s
last char is r
all but first is laughter
```

The first position or **index** of a character in a string is zero, so the last index in a string of 11 characters is 10. The prototypes for these functions are given in Table 4.6.

Each `string` member function used in Program 4.11 is invoked using an object and the dot operator. For example, `s.length()` returns the length of `s`. When I read code, I read this as "s dot length", and think of the length function as applied to the object s, returning the number of characters in s.

Table 4.6 Three `string` member functions

function prototype and description
`int length()` *postcondition: returns the number of characters in the string*
`string substr(int pos, int len)` *precondition: 0 <= pos < length()* *postcondition: returns substring of* `len` *characters beginning at position* `pos` (as many characters as possible if len too large, but error if pos is out of range)
`int find(string s)` *postcondition: returns first position/index at which string s begins* (returns `string::npos` if s does not occur)

Program Tip 4.8: Ask not what you can do to an object, ask what an object can do to itself. When you think about objects, you'll begin to think about what an object can tell you about itself rather than what you can tell an object to do.

In the last use of `substr` in Program 4.11 more characters are requested than can be supplied by the arguments in the call `s.substr(1, s.length())`. Starting at index 1, there are only `s.length()-1` characters in `s`. However, the function `substr` "does the right thing" when asked for more characters than there are, and gives as many as it can without generating an error. For a full description of this and other `string` functions see Howto C. Although the string returned by `substr` is printed in *strdemo.cpp*, the returned value could be stored in a string variable as follows:

```
string allbutfirst = s.substr(1,s.length());
```

The `string` Member Function `find`. The member function `find` returns the index in a string at which another string occurs. For example, `"plant"` occurs at index three in the string `"supplant"`, at index five in `"transplant"`, and does not occur in `"vegetable"`. Program 4.12, *strfind.cpp* shows how `find` works. The return value `string::npos` indicates that a substring does not occur. Your code should not depend on `string::npos` having any particular value[10].

Program 4.12 strfind.cpp

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

int main()
{
    string target = "programming is a creative process";
    string s;
    cout << "target string: " << target << endl;
    cout << "search for what substring: ";
    cin >> s;
    int index = target.find(s);
    if (index != string::npos)
    {   cout << "found at " << index << endl;
    }
    else
    {   cout << "not found" << endl;
    }
    return 0;
}
```

strfind.cpp

[10]Actually, the value of `string::npos` is the largest positive index, see Howto C.

OUTPUT

```
prompt> strfind
target string: programming is a creative process
search for what substring: pro
found at 0
prompt> strfind
target string: programming is a creative process
search for what substring: gram
found at 3
prompt> strfind
target string: programming is a creative process
search for what substring: create
not found
```

The double colon `::` used in `string::npos` separates the value, in this case `npos`, from the class in which the value occurs, in this case `string`. The `::` is called the **scope resolution operator**, we'll study it in more detail in the next chapter.

4.6.2 Calling and Writing Functions

When you first begin to use functions that return values, you may forget to process the return value. All non-void functions return a value that should be used in a C++ expression (e.g., printed, stored in a variable, used in an arithmetic expression). The following C++ statements show how three of the functions studied in this chapter (`sqrt`, `NumToString`, and `IsLeap`) are used in expressions so that the values returned by the functions aren't ignored.

```
double hypotenuse = sqrt{side1*side1 + side2*side2);
cout << NumToString(47) << endl;
bool millenniumLeaps = IsLeap(2000) || IsLeap(3000);
```

It doesn't make sense, for example, to write the following statements in which the value returned by `sqrt` is ignored.

```
double s1, s2;
cout << "enter sides: ";
cin >> s1 >> s2;
double root;
sqrt(s1*s1 + s2*s2);
```

The programmer may have meant to store the value returned by the function call to `sqrt` in the variable `root`, but the return value from the function call in the last statement is ignored.

Whenever you call a function, think carefully about the function's prototype and its postcondition. Be sure that if the function returns a value that you use the value.[11]

Program Tip 4.9: Do not ignore the value returned by non-void functions. Think carefully about each function call you make when writing programs and do something with the return value that makes sense in the context of your program and that is consistent with the type and value returned.

Write Lots of Functions. When do you write a function? You may be writing a program like *pizza2.cpp*, Program 4.7, where the function `Cost` is used to calculate how much a square inch of pizza costs. The function is reproduced here.

```
double Cost(double radius, double price)
// postcondition: returns the price per sq. inch
{
   return price/(3.14159*radius*radius);
}
```

Is it worth writing another function called `CircleArea` like this?

```
double CircleArea(double radius)
// postcondition: return area of circle with given radius
{
   return radius*radius*3.14159;
}
```

In general, when should you write a function to encapsulate a calculation or sequence of statements? There is no simple answer to this question, but there are a few guidelines.

Program Tip 4.10: Functions encapsulate abstractions and lead to code that's often easier to read and modify. Do not worry about so-called execution time "overhead" in the time it takes a program to execute a function call. Make your programs correct and modifiable before worrying about making them fast.

As an example, it's often easier to write a complex boolean expression as a function that might include `if/else` statements, and then call the function, than to determine what the correct boolean expression is. In the next section we'll study a tool from logic that helps in writing correct boolean expressions, but writing functions are useful when you're trying to develop appropriate loop tests. For example, if you need to determine

[11] Some functions return a value but are called because they cause some change in program state separate from the value returned. Such functions are said to have **side-effects** since they cause an effect "on the side," or in addition to the value returned by the function. In some cases the returned value of a function with side-effects is ignored.

if a one-character string represents a consonant, it's probably easier to write a function `IsVowel` and use that function to write `IsConsonant`, or to use `!IsVowel()` when you need to determine if a string is a consonant.

```
bool IsVowel(string s)
// pre: s is a one-character string
// post: returns true if s is a vowel, return false
{
    if (s.length() != 1)
    {   return false;
    }
    return s == "a" || s == "e" || s == "i" ||
           s == "o" || s == "u";
}
```

The `return` Statement. In the function `IsVowel()` there are two `return` statements and an `if` without an `else`. When a `return` statement executes, the function being returned from immediately exits. In `IsVowel()`, if the string parameter `s` has more than one character, the function immediately returns `false`. Since the function exits, there is no need for an `else` body, though some programmers prefer to use an `else`. Some programmers prefer to have a single `return` statement in every function. To do this requires introducing a local variable and using an `else` body as follows.

```
bool IsVowel(string s)
// pre: s is a one-character string
// post: returns true if s is a vowel, else return false
{
    bool retval = false;   // assume false
    if (s.length() == 1)
    {   retval = (s == "a" || s == "e" || s == "i" ||
                  s == "o" || s == "u");
    }
    return retval;
}
```

You should try to get comfortable with the assignment to `retval` inside the `if` statement. It's often easier to think of the assignment using this code.

```
if (s == "a"|| s == "e"|| s == "i"|| s == "o"|| s == "u")
{   retval = true;
}
else
{   retval = false;
}
```

This style of programming uses more code. It's just as efficient, however, and it's ok to use it though the single assignment to `retval` is more terse and, to many, more elegant.

4.6.3 The `Date` class

At the beginning of this chapter we discussed writing a function `PrintMonth` that prints a calendar for a month specified by a number and a year. As described, printing a calendar for January of the year 2001 could be done with the call `PrintMonth(1,2001)`. You could write this function with the tools we've studied in this chapter though it would be cumbersome to make a complete calendar. However, using the class `Date` makes it much simpler to write programs that involve dates and calendars than writing your own functions like `IsLeap`. In general, it's easier to use classes that have been developed and debugged than to develop your own code to do the same thing, though it's not always possible to find classes that serve your purposes.

We won't discuss this class in detail until the next chapter, but you can see how variables of type `Date` are defined, and two of the `Date` member functions used in Program 4.13. More information about the class is accessible in Howto G. To use `Date` objects you'll need to add `#include"date.h"` to your programs.

how to...

Program 4.13 datedemo.cpp

```
#include <iostream>
using namespace std;
#include "date.h"

// simple preview of using the class Date

int main()
{
    int month, year;
    cout << "enter month (1-12) and year ";
    cin >> month >> year;

    Date d(month, 1, year);
    cout << "that day is " << d << ", it is a " << d.DayName() << endl;
    cout << "the month has " << d.DaysIn() << " days in it " << endl;

    return 0;
}
```

datedemo.cpp

After examining the program and the output on the next page, you should be think about how you would use the class `Date` to solve the following problems, each can be solved with just a few lines of code.

Pause to Reflect

4.24 Determine if a year the user enters is a leap year.

4.25 Determine the day of the week of any date (month, day, year) the user enters.

4.26 Determine the day of the week your birthday falls on in the year 2002.

OUTPUT

```
prompt> datedemo
enter month (1-12) and year 9 1999
that day is September 1 1999, it is a Wednesday
the month has 30 days in it
prompt> datedemo
enter month (1-12) and year 2 2000
that day is February 1 2000, it is a Tuesday
the month has 29 days in it
```

4.7 Using Boolean Operators: De Morgan's Law

Many people new to the study of programming have trouble developing correct expressions used for the guard of an *if* statement. For example, suppose you need to print an error message if the value of an `int` variable is either 7 or 11.

```
if (value == 7 || value == 11)
{   cout << "**error** illegal value: " << value << endl;
}
```

The statement above prints an error message for the illegal values of 7 and 11 only and not for other, presumably legal, values. On the other hand, suppose you need to print an error message if the value is anything other than 7 or 11 (i.e., 7 and 11 are the only legal values). What do you do then? Some beginning programmers recognize the similarity between this and the previous problem and write code like the following.

```
if (value == 7 || value == 11)
{   // do nothing, value ok
}
else
{   cout << "**error** illegal value: " << value << endl;
}
```

This code works correctly, but the empty block guarded by the `if` statement is not the best programming style. One simple way to avoid the empty block is to use the logical negation operator. In the code below the operator `!` negates the expression that follows so that an error message is printed when the value is anything other than 7 or 11.

```
if ( ! (value == 7 || value == 11) )
{   cout << "**error** illegal value: " << value << endl;
}
```

Table 4.7 De Morgan's Laws for logical operators

expression	logical equivalent by De Morgan's law
`! (a && b)`	`(!a) || (!b)`
`! (a || b)`	`(!a) && (!b)`

Alternatively, we can use De Morgan's law[12] to find the logical negation, or opposite, of an expression formed with the logical operators && and `||`. De Morgan's laws are summarized in Table 4.7.

The negation of an && expression is an `||` expression, and vice versa. We can use De Morgan's law to develop an expression for printing an error message for any value other than 7 or 11 by using the logical equivalent of the guard in the `if` statement above.

```
if ( (value != 7 && (value != 11) )
{   cout << "**error** illegal value: " << value << endl;
}
```

De Morgan's law can be used to reason effectively about guards when you read code. For example, if the code below prints an error message for illegal values, what are the legal values?

```
if (s != "rock" && s != "paper" && s != "scissors")
{    cout << "** error** illegal value: " << s << endl;
}
```

By applying De Morgan's law twice, we find the logical negation of the guard which tells us the legal values (what would be an else block in the statement above.)

```
if (s == "rock" || s == "paper" || s == "scissors") //legal
```

This shows the legal values are "rock" or "paper" or "scissors" and all other strings represent illegal values.

[12]Augustus De Morgan (1806–1871), first professor of mathematics at University College, London, as well as teacher to Ada Lovelace (see Section 2.5.2.)

Richard Stallman *(b. 1953)*

Richard Stallman is hailed by many as "the world's best programmer." Before the term *hacker* became a pejorative, he used it to describe himself as "someone fascinated with how things work, [who would see a broken machine and try to fix it]."

Stallman believes that software should be free, that money should be made by adapting software and explaining it, but not by writing it. Of software he says, "I'm going to make it free even if I have to write it all myself." Stallman uses the analogy that for software he means "free as in free speech, not as in free beer." He is the founder of the GNU software project, which creates and distributes free software tools. The GNU `g++` compiler, used to develop the code in this book, is widely regarded as one of the best compilers in the world. The free operating system `Gnu/Linux` has become one of the most widely used operating systems in the world. In 1990 Stallman received a MacArthur "genius" award of $240,000 for his dedication and work. He continues this work today as part of the League for Programming Freedom, an organization that fights against software patents (among other things). In an interview after receiving the MacArthur award, Stallman had a few things to say about programming freedom:

> *I disapprove of the obsession with profit that tempts people to throw away their ideas of good citizenship.…businesspeople design software and make their profit by obstructing others' understanding. I made a decision not to do that. Everything I do, people are free to share. The only thing that makes developing a program worthwhile is the good it does.*

4.8 Chapter Review

In this chapter we discussed using and building functions. Changing the flow of control within functions is important in constructing programs. Encapsulating information in functions that return values is an important abstraction technique and a key concept in building large programs.

The `if/else` statement can be used to alter the flow of control in a program.

You can write programs that respond differently to different inputs by using `if/else` statements. The test in an `if` statement uses relational operators to yield a boolean value whose truth determines what statements are executed. In addition to relational operators, logical (boolean), arithmetic, and assignment operators were discussed and used in several different ways.

The following C++ and general programming features were covered in this chapter:

- The `if/else` statement is used for conditional execution of code. Cascaded `if` statements are formatted according to a convention that makes them more readable.
- A function's return type is the type of value returned by the function. For example, the function `sqrt` returns a `double`. Functions can return values of any type.
- The library whose interface is specified in `<cmath>` supplies many useful mathematical functions.
- Boolean expressions and tests have values of true or false and are used as the tests that guard the body of code in `if/else` statements. The type `bool` is a built-in type in C++ with values of `true` and `false`.
- A block (compound) statement is surrounded by { and } delimiters and is used to group several statements together.
- Relational operators are used to compare values. For example, `3 < 4` is a relational expression using the < operator. Relational operators include `==`, `!=`, <, >, `<=`, `>=`. Relational expressions have boolean values.
- Logical operators are used to combine boolean expressions. The logical operators are `||`, &&, `!`. Both `||` and && (logical or and logical and, respectively) are evaluated using **short-circuit** evaluation.
- Boolean operators in C++ use short-circuit evaluation so that only as much of an expression is evaluated from left-to-right as needed to determine whether the expression is true (or false).
- Defensive programming is a style of programming in which care is taken to prevent errors from occurring rather than trying to clean up when they do occur.
- Pre- and postconditions are a method of commenting functions; if the preconditions are true when a function is called, the postconditions will be true when the function has finished executing. These provide a kind of contractual arrangement between a function and the caller of a function.
- Several **member functions** of the `string` class can be used to determine the length of a string and to find substrings of a given string. Non-member functions are called **free functions**.
- Functions encapsulate abstractions like when a leap year occurs and in calculating a square root. Functions should be used with regularity in programs.
- Classes encapsulate related functions together. The class `string` encapsulates functions related to manipulating strings and the class `Date` encapsulates functions related to calendar dates.
- De Morgan's laws are useful in developing boolean expressions for use in `if` statements, and in reasoning about complex boolean expressions.

4.9 Exercises

4.1 Write a program that prompts the user for a person's first and last names (be careful; more than one `cin >>` statement may be necessary). The program should print a message that corresponds to the user's names. The program should recognize at least four different names. For example:

OUTPUT

```
enter your first name> Owen
enter your last name>  Astrachan
Hi Owen, your last name is interesting.
enter your first name> Dave
enter your last name>  Reed
Hi Dave, your last name rhymes with thneed.
```

4.2 Write a function whose specification is

```
string IntToRoman(int num)
// precondition: 0 <= num <= 10
// postcondition: returns Roman equivalent of num
```

so that `cout << IntToRoman(7) << endl;` would cause `"VII"` to be printed. Note the precondition. Write a program to test that the function works.

4.3 Write a function with prototype `int Min2(int,int)` that returns the minimum value of its parameters. Then use this function to write another function with prototype `int Min3(int,int,int)` that returns the minimum of its three parameters. `Min3` can be written with a single line:

```
int Min3(int x, int y, int z)
// post: returns minimum of x, y, and z
{
   return Min2(                         );
}
```

where the two-parameter function is called with appropriate actual parameters. Write a test program to test both functions.

You can then rewrite the minimum functions, naming them both `Min`. In C++, functions can have the same name, if their parameters differ (this is another example of overloading).

4.4 Write a program in which the user is prompted for a real number (of type `double`) and a positive integer and that prints the double raised to the integer power. Use the function `pow` from `<cmath>`. For example:

O U T P U T

```
enter real number 3.5
enter positive power 5
3.5 raised to the power 5 = 525.218
```

4.5 Write a program that is similar to *numtoeng.cpp*, Program 4.1, but that prints an English equivalent for any number less than one million. If you know a language other than English (e.g., French, Spanish, Arabic), use that language instead of English.

4.6 Use the function `sqrt` from the math library[13] to write a function `PrintRoots` that prints the roots of a quadratic equation whose coefficients are passed as parameters.

```
PrintRoots(1,-5,6);
```

might cause the following to be printed, but your output doesn't have to look exactly like this.

O U T P U T

```
roots of equation 1*xˆ2 - 5*x + 6 are 2.0 and 3.0
```

4.7 (from [Coo87]) The surface area of a person is given by the formula

$$7.184^{-3} \times \text{weight}^{0.452} \times \text{height}^{0.725} \tag{4.2}$$

where weight is in kilograms and height is in centimeters. Write a program that prompts for height and weight and then prints the surface area of a person. Use the function `pow` from `<cmath>` to raise a number to a power.

4.8 Write a program using the class `Date` that prints the day of the week on which your birthday occurs for the next seven years.

4.9 Write a program using ideas from the head-drawing program *parts.cpp*, Program 2.4, that could be used as a kind of police sketch program. A sample run could look like the following.

[13]On some systems you may need to link the math library to get access to the square root function.

O U T P U T

```
prompt> sketch
Choices of hair style follow

(1)  parted
(2)  brush cut
(3)  balding

enter choice: 1
Choices of eye style follow

(1)  beady-eyed
(2)  wide-eyed
(3)  wears glasses

enter choice: 3
Choices of mouth style follow

(1)  smiling
(2)  straightfaced
(3)  surprised

enter choice: 3

   ||||||||////////
   |              |
   |   ---  ---   |
   |---|o|--|o|---|
   |   ---  ---   |
  _|              |_
 |_                _|
   |       o      |
   |              |
```

4.10 Write a function that allows the user to design different styles of T-shirts. You should allow choices for the neck style, the sleeve style, and the phrase or logo printed on the T-shirt. For example,

OUTPUT

```
prompt> teedesign
Choices of neck style follow

(1) turtle neck
(2) scoop neck (rounded)
(3) vee neck

enter choice: 1
Choices of sleeve style follow

(1) short
(2) sleeveless
(3) long

enter choice: 2
Choices of logo follow

(1) logo once
(2) logo three times
(3) logo slanted

enter choice: 3
            +------+
            |      |
      -------      ------
     /                   \
    /                     \
  --                        --
 |                            |
 |                            |
 |                            |
  --                        --
    |    F                 |
    |         O            |
    |             O        |
    |                      |
    |                      |
```

4.11 (from[KR96]) The wind chill temperature is given according to a somewhat complex formula derived empirically. The formula converts a temperature (in degrees Fahrenheit) and a wind speed to an equivalent temperature (eqt) as follows:

$$\text{eqt} = \begin{cases} \text{temp} & \text{if wind} \leq 4 \\ a - (b + c \times \sqrt{\text{wind}} - d \times \text{wind}) \times (a - \text{temp})/e & \text{if temp} \leq 45 \\ 1.6 * \text{temp} - 55.0 & \text{otherwise} \end{cases} \tag{4.3}$$

Where $a = 91.4$, $b = 10.45$, $c = 6.69$, $d = 0.447$, $e = 22.0$. Write a program that prompts for a wind speed and a temperature and prints the corresponding wind chill temperature. Use a function `WindChill` with the following prototype:

```
double WindChill(double temperature, double windSpeed)
// pre: temperature in degrees Fahrenheit
// post: returns wind-chill index/
//       comparable temperature
```

4.12 (also from [KR96]) The U.S. CDC (Centers for Disease Control—this time, not Control Data Corporation) determine obesity according to a "body mass index," computed by

$$\text{index} = \frac{\text{weight in kilograms}}{(\text{height in meters})^2} \qquad (4.4)$$

An index of 27.8 or greater for men or 27.3 or greater for nonpregnant women is considered obese. Write a program that prompts for height, weight, and sex and that determines whether the user is obese. Write a function that returns the body mass index given the height and weight in inches and pounds, respectively. Note that one meter is 39.37 inches, one inch is 2.54 centimeters, one kilogram is 2.2 pounds, and one pound is 454 grams.

4.13 Write a program that converts a string to its Pig-Latin equivalent. To convert a string to Pig-Latin use the following algorithm:

1. If the string begins with a vowel, add `"way"` to the string. For example, Pig-Latin for "apple" is "appleway."
2. Otherwise, find the first occurrence of a vowel, move all the characters before the vowel to the end of the word, and add `"ay"`. For example, Pig-Latin for "strong" is "ongstray" since the characters "str" occur before the first vowel.

Assume that vowels are a, e, i, o, and u. You'll find it useful to write several functions to help in converting a string to its Pig-Latin equivalent. You'll need to use string member functions `substr`, `find`, and `length`. You'll also need to concatenate strings using `+`. Finally, to find the first vowel, you may find it useful to write a function that returns the minimum of two values. You'll need to be careful with the value `string::npos` returned by the string member function `find`. Sample output for the program follows.

OUTPUT

```
prompt> pigify
enter string: strength
strength = engthstray
prompt> pigify
enter string: alpha
alpha = alphaway
prompt> pigify
enter string: frzzl
frzzl = frzzlay
```