# Chapter 1

*Programming is best regarded as*
*the process of creating works of literature,*
*which are meant to be read.*
—Donald E. Knuth

# Elementary C++ Programming

A *program* is a sequence of instructions that can be executed by a computer. Every program is written in some programming language. C++ (pronounced "see-plus-plus") is one of the most powerful programming languages available. It gives the programmer the power to write efficient, structured, object-oriented programs.

## 1.1 GETTING STARTED

To write and run C++ programs, you need to have a text editor and a C++ compiler installed on your computer. A *text editor* is a software system that allows you to create and edit text files on your computer. Programmers use text editors to write programs in a programming language such as C++. A *compiler* is a software system that translates programs into the machine language (called *binary code*) that the computer's operating system can then run. That translation process is called *compiling* the program. A *C++ compiler* compiles C++ programs into machine language.

If your computer is running a version of the Microsoft Windows operating system (*e.g.*, Windows 98 or Windows 2000), then it already has two text editors: WordPad and Notepad. These can be started from the Start key. In Windows 98, they are listed under Accessories.

Windows does not come with a built-in C++ compiler. So unless someone has installed a C++ compiler on the machine you are using, you will have to do that yourself. If you are using a Windows computer that is maintained by someone else (*e.g.*, an Information Services department at your school or company), you may find a C++ compiler already installed. Use the Start key to look under Programs for Borland C++Builder, Metrowerks CodeWarrior, Microsoft Visual C++, or any other program with "C++" in its name. If you have to buy your own C++ compiler, browse the Web for inexpensive versions of any of the compilers mentioned above. These are usually referred to as IDEs (*Integrated Development Environment*s) because they include their own specialized text editors and debuggers.

If your computer is running a proprietary version of the UNIX operating system on a workstation (*e.g.*, Sun Solaris on a SPARCstation), it may already have a C++ compiler installed. An easy way to find out is to create the program shown in Example 1.1 on page 2, name it `hello.C`, and then try to compile it with the command

```
CC hello
```

The Free Software Foundation has a suite of UNIX software, named "GNU" software that can be downloaded for free from

```
http://www.gnu.org/software/software.html
```

Use their GCC package which includes a C++ compiler and their Emacs editor. For DOS systems, use their DJGPP which includes a C++ compiler.

## 1.2 SOME SIMPLE PROGRAMS

Now you have a text editor for writing C++ programs and a C++ compiler for compiling them. If you are using an IDE such as Borland C++Builder on a PC, then you can compile and run your programs by clicking on the appropriate buttons. Other systems may require you to use the command line to run your programs. In that case, you do so by entering the file name as a command. For example, if your source code is in a file named `hello.cpp`, type

```
hello
```

at the command line to run the program after it has been compiled.

When writing C++ programs, remember that C++ is *case-sensitive*. That means that `main()` is different from `Main()`. The safest policy is to type everything in lower-case except when you have a compelling reason to capitalize something.

### EXAMPLE 1.1 The "Hello, World" Program

This program simply prints "Hello, World!":

```
#include <iostream>
int main()
{ std::cout << "Hello, World!\n";
}
```

The first line of this source code is a *preprocessor directive* that tells the C++ compiler where to find the definition of the `std::cout` object that is used on the third line. The identifier `iostream` is the name of a file in the *Standard C++ Library*. Every C++ program that has standard input and output must include this preprocessor directive. Note the required punctuation: the pound sign `#` is required to indicate that the word "`include`" is a preprocessor directive; the angle brackets `< >` are required to indicate that the word "`iostream`" (which stands for "input/output stream") is the name of a Standard C++ Library file. The expression `<iostream>` is called a *standard header*.

The second line is also required in every C++ program. It tells where the program begins. The identifier `main` is the name of a function, called *the main function* of the program. Every C++ program must have one and only one `main()` function. The required parentheses that follow the word "main" indicate that it is a function. The keyword `int` is the name of a data type in C++. It stands for "integer". It is used here to indicate the *return type* for the `main()` function. When the program has finished running, it can return an integer value to the operating system to signal some resulting status.

The last two lines constitute the actual body of the program. A *program body* is a sequence of program statements enclosed in braces `{ }`. In this example there is only one statement:

```
std::cout << "Hello, World!\n";
```

It says to send the string `"Hello, World!\n"` to the *standard output stream* object `std::cout`. The single symbol `<<` represents the *C++ output operator*. When this statement executes, the characters enclosed in quotation marks `" "` are sent to the *standard output device* which is usually the computer screen. The last two characters `\n` represent the *newline character*. When the output device encounters that character, it advances to the beginning of the next line of text on the screen. Finally, note that every program statement must end with a semicolon (`;`).

Notice how the program in Example 1.1 is formatted in four lines of source code. That formatting makes the code easier for humans to read. The C++ compiler ignores such formatting. It

reads the program the same as if it were written all on one line, like this:

```
#include <iostream>
int main(){std::cout<<"Hello, World!\n";}
```

Blank spaces are ignored by the compiler except where needed to separate identifiers, as in

```
int main
```

Note that the preprocessor directive must precede the program on a separate line.

### EXAMPLE 1.2 Another "Hello, World" Program

This program has the same output as that in Example 1.1:

```
#include <iostream>
using namespace std;
int main()
{ // prints "Hello, World!":
  cout << "Hello, World!\n";
  return 0;
}
```

The second line

```
using namespace std;
```

tells the C++ compiler to apply the prefix `std::` to resolve names that need prefixes. It allows us to use `cout` in place of `std::cout`. This makes larger programs easier to read.

The fourth line

```
{ // prints "Hello, World!"
```

includes the comment "`prints "Hello, World!"`". A *comment* in a program is a string of characters that the preprocessor removes before the compiler compiles the programs. It is included to add explanations for human readers. In C++, any text that follows the double slash symbol `//`, up to the end of the line, is a comment. You can also use C style comments, like this:

```
{ /*  prints "Hello, World!"  */
```

A *C style comment* (introduced by the programming language named "C") is any string of characters between the symbol `/*` and the symbol `*/`. These comments can run over several lines.

The sixth line

```
return 0;
```

is optional for the `main()` function in Standard C++. We include it here only because some compilers expect it to be included as the last line of the `main()` function.

A *namespace* is a named group of definitions. When objects that are defined within a namespace are used outside of that namespace, either their names must be prefixed with the name of the namespace or they must be in a block that is preceded by a `using namespace` statement. Namespaces make it possible for a program to use different objects with the same name, just as different people can have the same name. The `cout` object is defined within a namespace named `std` (for "standard") in the `<iostream>` header file.

Throughout the rest of this book, every program is assumed to begin with the two lines

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

These two required lines will be omitted in the examples. We will also omit the line

```
  return 0;
```

from the `main()` function. Be sure also to include this line if you are using a compiler (such as Microsoft Visual C++) that expects it.

## 1.3 THE OUTPUT OPERATOR

The symbol `<<` is called the *output operator* in C++. (It is also called the *put operator* or the *stream insertion operator*.) It inserts values into the output stream that is named on its left. We usually use the `cout` output stream, which ordinarily refers to the computer screen. So the statement

```
cout << 66;
```

would display the number 66 on the screen.

An *operator* is something that performs an action on one or more objects. The output operator `<<` performs the action of sending the value of the expression listed on its right to the output stream listed on its left. Since the direction of this action appears to be from right to left, the symbol `<<` was chosen to represent it. It should remind you of an arrow pointing to the left.

The `cout` object is called a "stream" because output sent to it flows like a stream. If several things are inserted into the `cout` stream, they fall in line, one after the other as they are dropped into the stream, like leaves falling from a tree into a natural stream of water. The values that are inserted into the `cout` stream are displayed on the screen in that order.

### EXAMPLE 1.3 Yet Another "Hello, World" Program

This program has the same output as that in Example 1.1:

```
int main()
{ // prints "Hello, World!":
  cout << "Hel" << "lo, Wo" << "rld!" << endl;
}
```

The output operator is used four times here, dropping the four objects `"Hel"`, `"lo, Wo"`, `"rld!"`, and `endl` into the output stream. The first three are strings that are concatenated together (*i.e.*, strung end-to-end) to form the single string `"Hello, World!"`. The fourth object is the *stream manipulator* object `endl` (meaning "end of line"). It does the same as appending the *endline character* `'\n'` to the string itself: it sends the print cursor to the beginning of the next line. It also "flushes" the output buffer.

## 1.4 CHARACTERS AND LITERALS

The three objects `"Hel"`, `"lo, Wo"`, and `"rld!"` in Example 1.3 are called *string literals*. Each literal consists of a sequence of characters delimited by quotation marks.

A *character* is an elementary symbol used collectively to form meaningful writing. English writers use the standard Latin alphabet of 26 lower case letters and 26 upper case letters along with the 10 Hindu-Arabic numerals and a collection of punctuation marks. Characters are stored in computers as integers. A *character set code* is a table that lists the integer value for each character in the set. The most common character set code in use at the end of the millennium is the *ASCII Code*, shown in Appendix A. The acronym (pronounced "as-key") stands for American Standard Code for Information Interchange.

The *newline character* `'\n'` is one of the nonprinting characters. It is a single character formed using the backslash `\` and the letter `n`. There are several other characters formed this way, including the *horizontal tab* character `'\t'` and the *alert character* `'\a'`. The backslash is also used to denote the two printing characters that could not otherwise be used within a string literal: the quote character `\"` and the backslash character itself `\\`.

Characters can be used in a program statement as part of a string literal, or as individual objects. When used individually, they must appear as character constants. A character constant is a character enclosed in single quotes. As individual objects, character constants can be output the same way string literals are.

### EXAMPLE 1.4 A Fourth Version of the "Hello, World" Program

This program has the same output as that in Example 1.1:

```
int main()
{ // prints "Hello, World!":
  cout << "Hello, W" << 'o' << "rld" << '!' << '\n';
}
```

This shows that the output operator can process characters as well as string literals. The three individual characters `'o'`, `'!'`, and `'\n'` are concatenated into the output the same was as the two string literals `"Hello, W"` and `"rld"`.

### EXAMPLE 1.5 Inserting Numeric Literals into the Standard Output Stream

```
int main()
{ // prints "The Millennium ends Dec 31 2000.":
  cout << "The Millennium ends Dec " << 3 << 1 << ' ' << 2000 << endl;
}
```

When numeric literals like `3` and `2000` are passed to the output stream they are automatically converted to string literals and concatenated the same way as characters. Note that the *blank character* (`' '`) must be passed explicitly to avoid having the digits run together.

## 1.5 VARIABLES AND THEIR DECLARATIONS

A *variable* is a symbol that represents a storage location in the computer's memory. The information that is stored in that location is called the *value* of the variable. One common way for a variable to obtain a value is by an *assignment*. This has the syntax

*variable* **=** *expression*;

First the *expression* is evaluated and then the resulting value is assigned to the *variable*. The equals sign "=" is the *assignment operator* in C++.

### EXAMPLE 1.6 Using Integer Variables

In this example, the integer `44` is assigned to the variable `m`, and the value of the expression `m + 33` is assigned to the variable `n`:

```
int main()
{ // prints "m = 44 and n = 77":
  int m, n;
  m = 44;  // assigns the value 44 to the variable m
  cout << "m = " << m;
  n = m + 33;  // assigns the value 77 to the variable n
  cout << " and n = " << n << endl;
}
```

The output from the program is shown in the shaded panel at the top of the next page.

```
m = 44 and n = 77
```

We can view the variables `m` and `n` like this:


The variable named `m` is like a mailbox. Its name `m` is like the address on a mailbox, its value `44` is like the contents of a mailbox, and its type `int` is like a legal classification of mailboxes that stipulates what may be placed inside it. The type `int` means that the variable holds only integer values.

Note in this example that both `m` and `n` are declared on the same line. Any number of variables can be declared together this way if they have the same type.

Every variable in a C++ program must be declared before it is used. The syntax is

*`specifier type name initializer;`*

where *`specifier`* is an optional keyword such as `const` (see Section 1.8), *`type`* is one of the C++ data types such as `int`, *`name`* is the name of the variable, and *`initializer`* is an optional initialization clause such as `=44` (see Section 1.7).

The purpose of a declaration is to introduce a name to the program; *i.e.*, to explain to the compiler what the name means. The *`type`* tells the compiler what range of values the variable may have and what operations can be performed on the variable.

The location of the declaration within the program determines the *scope* of the variable: the part of the program where the variable may be used. In general, the scope of a variable extends from its point of declaration to the end of the immediate block in which it is declared or which it controls.

## 1.6 PROGRAM TOKENS

A computer program is a sequence of elements called *tokens*. These tokens include keywords such as `int`, identifiers such as `main`, punctuation symbols such as `{`, and operators such as `<<`. When you compile your program, the compiler scans the text in your source code, parsing it into tokens. If it finds something unexpected or doesn't find something that was expected, then it aborts the compilation and issues error messages. For example, if you forget to append the semicolon that is required at the end of each statement, then the message will report the missing semicolon. Some syntax errors such as a missing second quotation mark or a missing closing brace may not be described explicitly; instead, the compiler will indicate only that it found something wrong near that location in your program.

### EXAMPLE 1.7 A Program's Tokens

```
int main()
{ // prints "n = 44":
  int n=44;
  cout << "n = " << n << endl;
}
```

The output is

```
n = 44
```

This source code has 19 tokens: "`int`", "`main`", "`(`", "`)`", "`{`", "`int`", "`n`", "`=`", "`44`", "`;`", "`cout`", "`<<`", "`"n = "`", "`<<`", "`n`", "`<<`", "`endl`", "`;`", and "`}`". Note that the compiler ignores the comment symbol `//` and the text that follows it on the second line.

### EXAMPLE 1.8 An Erroneous Program

This is the same program as above except that the required semicolon on the third line is missing:

```
int main()
{ // THIS SOURCE CODE HAS AN ERROR:
  int n=44
  cout << "n = " << n << endl;
}
```

One compiler issued the following error message:

```
Error   : ';' expected
Testing.cpp line 4     cout << "n = " << n << endl;
```

This compiler underlines the token where it finds the error. In this case, that is the "`cout`" token at the beginning of the fourth line. The missing token was not detected until the next token was encountered.

## 1.7 INITIALIZING VARIABLES

In most cases it is wise to initialize variables where they are declared.

### EXAMPLE 1.9 Initializing Variables

This program contains one variable that is not initialized and one that is initialized.

```
int main()
{ // prints "m = ?? and n = 44":
  int m;  // BAD: m is not initialized
  int n=44;
  cout << "m = " << m << " and n = " << n << endl;
}
```

```
m = ?? and n = 44
```

The output is shown in the shaded box.

This compiler handles uninitialized variables in a special way. It gives them a special value that appears as `??` when printed. Other compilers may simply leave "garbage" in the variable, producing output like this:

```
m = -2107339024 and n = 44
```

In larger programs, uninitialized variables can cause troublesome errors.

## 1.8 OBJECTS, VARIABLES, AND CONSTANTS

An *object* is a contiguous region of memory that has an address, a size, a type, and a value. The *address* of an object is the memory address of its first byte. The *size* of an object is simply the number of bytes that it occupies in memory. The *value* of an object is the constant determined by the actual bits stored in its memory location and by the object's type which prescribes how those bits are to be interpreted.

For example, with GNU C++ on a UNIX workstation, the object `n` defined by

```
int n = 22;
```

has the memory address `0x3fffcd6`, the size 4, the type **`int`**, and the value 22. (The memory address is a hexadecimal number. See Appendix G.)

The type of an object is determined by the programmer. The value of an object may also be determined by the programmer at compile time, or it may be determined at run-time. The size of an object is determined by the compiler. For example, in GNU C++ an **int** has size 4, while in Borland C++ its size is 2. The address of an object is determined by the computer's operating system at run-time.

Some objects do not have names. A *variable* is an object that has a name. The object defined above is a variable with name 'n'.

The word "variable" is used to suggest that the object's value can be changed. An object whose value cannot be changed is called a *constant*. Constants are declared by preceding its type specifier with the keyword **const**, like this:

```
const int N = 22;
```

Constants must be initialized when they are declared.

### EXAMPLE 1.10 The const Specifier

This program illustrates constant definitions:

```
int main()
{ // defines constants; has no output:
  const char BEEP = '\b';
  const int MAXINT = 2147483647;
  const int N = MAXINT/2;
  const float KM_PER_MI = 1.60934;
  const double PI = 3.14159265358979323846;
}
```

Constants are usually defined for values like $\pi$ that will be used more than once in a program but not changed.

It is customary to use all capital letters in constant identifiers to distinguish them from other kinds of identifiers. A good compiler will replace each constant symbol with its numeric value.

## 1.9 THE INPUT OPERATOR

In C++, input is almost as simple as output. The *input operator* >> (also called the *get operator* or the *extraction operator*) works like the output operator <<.

### EXAMPLE 1.11 Using the Input Operator

```
int main()
{ // tests the input of integers, floats, and characters:
  int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  cout << "m =  " << m << ", n = " << n << endl;
  double x, y, z;
  cout << "Enter three decimal numbers: ";
  cin >> x >> y >> z;
  cout << "x =  " << x << ", y = " << y << ", z = " << z << endl;
  char c1, c2, c3, c4;
  cout << "Enter four characters: ";
```

```
    cin >> c1 >> c2 >> c3 >> c4;
    cout << "c1 =  " << c1 << ", c2 = " << c2 << ", c3 = " << c3
         << ", c4 = " << c4 << endl;
}
Enter two integers: 22 44
m =  22, n = 44
Enter three decimal numbers: 2.2 4.4 6.6
x =  2.2, y = 4.4, z = 6.6
Enter four characters: ABCD
c1 =  A, c2 = B, c3 = C, c4 = D
```

The input is shown in boldface in the output panel.

## Review Questions

**1.1** Describe the two ways to include comments in a C++ program.

**1.2** What is wrong with this program?

```
#include <iostream>
int main()
{ //  prints "Hello, World!":
  cout << "Hello, World!\n"
}
```

**1.3** What is wrong with the following C-style comment?

```
cout << "Hello, /* change? */ World.\n";
```

**1.4** What's wrong with this program:

```
#include <iostream>;
int main
{ // prints "n = 22":
  n = 22;
  cout << "n = << n << endl;
}
```

**1.5** What does a declaration do?

**1.6** What is the purpose of the preprocessing directive:

```
#include <iostream>
```

**1.7** What is the shortest possible C++ program?

**1.8** Where does the name "C++" come from?

**1.9** What's wrong with these declarations:

```
int first = 22, last = 99, new = 44, old = 66;
```

**1.10** In each of the following, assume that `m` has the value 5 and `n` has the value 2 before the statement executes. Tell what the values of `m` and `n` will be after each of the following statements executes:

***a.*** `m *= n++;`

***b.*** `m += --n;`

**1.11** Evaluate each of the following expressions, assuming in each case that `m` has the value 25 and `n` has the value 7:

***a.*** `m - 8 - n`

***b.*** `m = n = 3`

***c.*** `m%n`

***d.*** `m%n++`

***e.*** `m%++n`

***f.*** `++m - n--`

**1.12** Parse the following program, identifying all the keywords, identifiers, operators, literals, punctuation, and comments:

```
int main()
{ int n;
  cin >> n;
  n *= 3;  // multiply n by 3
  cout << "n=" << n << endl;
}
```

**1.13** Identify and correct the error in each of the following:

***a.*** `cout >> count;`

***b.*** `int double=44;`

**1.14** How do the following two statements differ:

```
char ch = 'A';
char ch = 65;
```

**1.15** What code could you execute to find the character whose ASCII code is 100?

**1.16** What does "floating-point" mean, and why is it called that?

**1.17** What is numeric overflow?

**1.18** How is integer overflow different from floating-point overflow?

**1.19** What is a run-time error? Give examples of two different kinds of run-time errors.

**1.20** What is a compile-time error? Give examples of two different kinds of compile-time errors.

## Problems

**1.1** Write four different C++ statements, each subtracting 1 from the integer variable `n`.

**1.2** Write a block of C++ code that has the same effect as the statement

```
n = 100 + m++;
```

without using the post-increment operator.

**1.3** Write a block of C++ code that has the same effect as the statement

```
n = 100 + ++m;
```

without using the pre-increment operator.

**1.4** Write a single C++ statement that subtracts the sum of `x` and `y` from `z` and then increments `y`.

**1.5** Write a single C++ statement that decrements the variable `n` and then adds it to `total`.

**1.6** Write a program that prints the first sentence of the Gettysburg Address (or your favorite quotation).

**1.7** Write a program that prints the block letter "B" in a $7 \times 6$ grid of stars like this:

```
*****
*    *
*    *
*****
*    *
*    *
*****
```

**1.8** Write and run a program that prints the first letter of your last name as a block letter in a $7 \times 7$ grid of stars.

**1.9** Write and run a program that shows what happens when each of the following ten "escape sequences" is printed: `\a`, `\b`, `\n`, `\r`, `\t`, `\v`, `\'`, `\"`, `\\`, `\?`.

**1.10** Write and run a program that prints the sum, difference, product, quotient, and remainder of two integers. Initialize the integers with the values 60 and 7.

**1.11** Write and run a program that prints the sum, difference, product, quotient, and remainder of two integers that are input interactively.

**1.12** Write and run a test program that shows how your system handles uninitialized variables.

**1.13** Write and run a program that causes negative overflow of a variable of type `short`.

**1.14** Write and run a program that demonstrates round-off error by executing the following steps: (1) initialize a variable `a` of type `float` with the value 666666; (2) initialize a variable `b` of type `float` with the value `1-1/a`; (3) initialize a variable `c` of type `float` with the value `1/b - 1`; (4) initialize a variable `d` of type `float` with the value `1/c + 1`; (5) print all four variables. Show algebraically that `d` = `a` even though the computed value of `d` ≠ `a`. This is caused by round-off error.

## Answers to Review Questions

**1.1** One way is to use the standard C style comment

```
/* like this */
```

The other way is to use the standard C++ style comment

```
// like this
```

The first begins with a slash-star and ends with a star-slash. The second begins with a double-slash and ends at the end of the line.

**1.2** The semicolon is missing from the last statement.

**1.3** Everything between the double quotes will be printed, including the intended comment.

**1.4** There are four errors: the precompiler directive on the first line should not end with a semicolon, the parentheses are missing from `main()`, `n` is not declared, and the quotation mark on the last line has no closing quotation mark.

**1.5** A declaration tells the compiler the name and type of the variable being declared. It also may be initialized in the declaration.

**1.6** It includes contents of the header file `iostream` into the source code. This includes declarations needed for input and output; *e.g.*, the output operator `<<`.

**1.7** `int main() { }`

**1.8** The name refers to the C language and its increment operator `++`. The name suggests that C++ is an advance over C.

**1.9** The only thing wrong with these declarations is that `new` is a keyword. Keywords are reserved and cannot be used for names of variables. See Appendix B for a list of the 62 keywords in C++.

**1.10** ***a.*** `m` will be 10 and `n` will be 3.

***b.*** `m` will be 6 and `n` will be 1.

**1.11** ***a.*** `m - 8 - n` evaluates to (25 - 8) - 7 = 17 - 7 = 10

***b.*** `m = n = 3` evaluates to 3

**1.12** ***a.*** `m - 8 - n` evaluates to (25 - 8) - 7 = 17 - 7 = 10

***b.*** `m = n = 3` evaluates to 3

***c.*** `m%n` evaluates to 25%7 = 4

***d.*** `m%n++` evaluates to 25%(7++) = 25%7 = 4

***e.*** `m%++n` evaluates to 25%(++7) = 25%8 = 1

***f.*** `++m - n--` evaluates to (++25) - (7--) = 26 - 7 = 19

**1.13** The keyword is `int`. The identifiers are `main`, `n`, `cin`, `cout`, and `endl`. The operators are `()`, `>>`, `*=`, and `<<`. The literals are `3` and `"n="`. The punctuation symbols are `{`, `;`, and `}`. The comment is "`// multiply n by 3`".

**1.14** ***a.*** The output object `cout` requires the output operator `<<`. It should be `cout << count;`

***b.*** The word `double` is a keyword in C++; it cannot be used as a variable name. Use: `int d=44;`

**1.15** Both statements have the same effect: they declare `ch` to be a `char` and initialize it with the value 65. Since this is the ASCII code for `'A'`, that character constant can also be used to initialize `ch` to 65.

**1.16** `cout << "char(100) = " << char(100) << endl;`

**1.17** The term "floating-point" is used to describe the way decimal numbers (rational numbers) are stored in a computer. The name refers to the way that a rational number like 386501.294 can be represented in the form $3.86501294 \times 10^5$ by letting the decimal point "float" to the left 5 places.

**1.18** Numeric overflow occurs in a computer program when the size of a numeric variable gets too big for its type. For example, on most computers values variables of type `short` cannot exceed 32,767, so if a variable of that type has the value 32,767 and is then incremented (or increased by any arithmetic operation), overflow will occur.

**1.19** When integer overflow occurs the value of the offending variable will "wrap around" to negative values, producing erroneous results. When floating-point overflow occurs, the value of the offending variable will be set to the constant `inf` representing infinity.

**1.20** A run-time error is an error that occurs when a program is running. Numeric overflow and division by zero are examples of run-time errors.

**1.21** A compile-time error is an error that occurs when a program is being compiled. Examples: syntax errors such as omitting a required semicolon, using an undeclared variable, using a keyword for the name of a variable.

## Solutions to Problems

**1.1** Four different statements, each subtracting 1 from the integer variable `n`:

***a.*** `n = n - 1;`

***b.*** `n -= 1;`

***c.*** `--n;`

***d.*** `n--;`

**1.2**
```
n = 100 + m;
++m;
```

**1.3**
```
++m;
n = 100 + m;
```

**1.4**
```
z -= (x + y++);
```

**1.5**
```
total += --n;
```

**1.6**
```
int main()
{ // prints the first sentence of the Gettysburg Address
  cout << "\tFourscore and seven years ago our fathers\n";
  cout << "brought forth upon this continent a new nation,\n";
  cout << "conceived in liberty, and dedicated to the\n";
  cout << "proposition that all men are created equal.\n";
}
```
```
        Fourscore and seven years ago our fathers
brought forth upon this continent a new nation,
conceived in liberty, and dedicated to the
proposition that all men are created equal.
```

**1.7**
```
int main()
{ // prints "B" as a block letter
  cout << "*****"  << endl;
  cout << "*    *" << endl;
  cout << "*    *" << endl;
  cout << "*****"  << endl;
  cout << "*    *" << endl;
  cout << "*    *" << endl;
```

```
  cout << "*****"  << endl;
}
```

```
*****
*    *
*    *
*****
*    *
*    *
*****
```

**1.8**

```
int main()
{ // prints "W" as a block letter
  cout << "*                  *"  << endl;
  cout << " *                *" << endl;
  cout << "  *              *" << endl;
  cout << "   *     *     *"  << endl;
  cout << "    *   * *   *" << endl;
  cout << "     * *   * *" << endl;
  cout << "      *     *"  << endl;
}
```

```
*                  *
 *                *
  *              *
   *     *     *
    *   * *   *
     * *   * *
      *     *
```

**1.9**

```
int main()
{ // prints escape sequences
  cout << "Prints \"\\nXXYY\": " << "\nXXYY" << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints \"\\nXX\\bYY\": " << "\nXX\bYY" << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints \"\\n\\tXX\\tYY\": " << "\n\tXX\tYY" << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints the \'\\a\' character: " << '\a' << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints the \'\\r\' character: " << '\r' << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints the \'\\v\' character: " << '\v' << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints the \'\\?\' character: " << '\?' << endl;
  cout << "-----------------------------------------" << endl;
}
```

```
Prints the '\v' character:
-----------------------------------------
Prints the '\?' character: ?
-----------------------------------------
Prints "\nXXYY":
XXYY
-----------------------------------------
Prints "\nXX\bYY":
XYY
-----------------------------------------
Prints "\n\tXX\tYY":
```

```
        XX      YY
-----------------------------------------
Prints the '\a' character:
-----------------------------------------
Prints the '\r' character:
-----------------------------------------
```

**1.10**

```
int main()
{ // prints the results of arithmetic operators
  int m = 60, n = 7;
  cout << "The integers are " << m << " and " << n << endl;
  cout << "Their sum is        " << (m + n) << endl;
  cout << "Their difference is " << (m - n) << endl;
  cout << "Their product is    " << (m * n) << endl;
  cout << "Their quotient is   " << (m / n) << endl;
  cout << "Their remainder is  " << (m % n) << endl;
}
```

```
The integers are 60 and 7
Their sum is        67
Their difference is 53
Their product is    420
Their quotient is   8
Their remainder is  4
```

**1.11**

```
int main()
{ // prints the results of arithmetic operators
  int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  cout << "The integers are " << m << " and " << n << endl;
  cout << "Their sum is        " << (m + n) << endl;
  cout << "Their difference is " << (m - n) << endl;
  cout << "Their product is    " << (m * n) << endl;
  cout << "Their quotient is   " << (m / n) << endl;
  cout << "Their remainder is  " << (m % n) << endl;
}
```

```
Enter two integers: 60 7
The integers are 60 and 7
Their sum is        67
Their difference is 53
Their product is    420
Their quotient is   8
Their remainder is  4
```

**1.12**

```
int main()
{ // prints the values of uninitialized variables
  bool b;  // not initialized
  cout << "b = " << b << endl;
  char c;  // not initialized
  cout << "c = [" << c << "]" << endl;
  int m;  // not initialized
  cout << "m = " << m << endl;
  int n;  // not initialized
  cout << "n = " << n << endl;
  long nn;  // not initialized
```

```
  cout << "nn = " << nn << endl;
  float x;  // not initialized
  cout << "x = " << x << endl;
  double y;  // not initialized
  cout << "y = " << y << endl;
}
```

```
b = 0
c =
m = 4296913
n = 4296716
nn = 4296794
x = 6.02438e-39
y = 9.7869e-307
```

**1.13**

```
int main()
{ // prints the values an overflowing negative short int
  short m=0;
  cout << "m = " << m << endl;
  m -= 10000;  // m should be -10,000
  cout << "m = " << m << endl;
  m -= 10000;  // m should be -20,000
  cout << "m = " << m << endl;
  m -= 10000;  // m should be -30,000
  cout << "m = " << m << endl;
  m -= 10000;  // m should be -40,000
  cout << "m = " << m << endl;
}
```

```
m = 0
m = -10000
m = -20000
m = -30000
m = 25536
```

**1.14**

```
int main()
{ float a = 666666;              // = a = 666666
  float b = 1 - 1/a;             // = (a-1)/a = 666665/666666
  float c = 1/b - 1;             // = 1/(a-1) = 1/666665
  float d = 1/c + 1;             // = a = 666666 != 671089
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "c = " << c << endl;
  cout << "d = " << d << endl;
}
```

```
a = 666666
b = 0.999999
c = 1.49012e-06
d = 671089
```

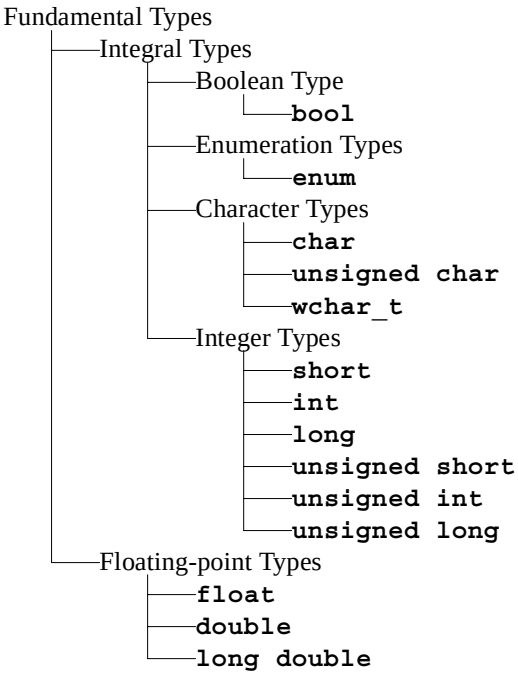
# Chapter 2

# Fundamental Types

## 2.1 NUMERIC DATA TYPES

In science there are two kinds of numbers: whole numbers (*e.g.*, 666) and decimal numbers (*e.g.*, 3.14159). Whole numbers, including 0 and negative whole numbers, are called *integers*. Decimal numbers, including negative decimal numbers and all integers, are called *rational numbers* because they can always be expressed as ratios of whole numbers (*i.e.*, fractions). Mathematics also uses irrational real numbers (*e.g.*, $\sqrt{2}$ and $\pi$), but these must be approximated with rational numbers to be used in computers.

Integers are used for counting; rational numbers are used for measuring. Integers are meant to be exact; rational numbers are meant to be approximate. When we say there are 12 people on the jury, we mean exactly 12, and anyone can count them to verify the statement. But when we say the tree is 12 meters high, we mean approximately 12.0 meters, and someone else may be just as accurate in saying that it is 12.01385 meters high.




This philosophical dichotomy is reflected in computers by the different ways in which these two fundamentally different kinds of numbers are stored and manipulated. Those differences are embodied in the two kinds of numeric types common to all programming languages: integral types and floating-point types. The term "floating-point" refers to the scientific notation that is used for rational numbers. For example, 1234.56789 can also be represented as $1.23456789 \times 10^{3}$, and 0.00098765 as $9.8765 \times 10^{-4}$. These alternatives are obtained by letting the decimal point "float" among the digits and using the exponent on 10 to count how many places it has floated to the left or right.

Standard C++ has 14 different *fundamental types*: 11 integral types and 3 floating-point types. These are outlined in the diagram shown above. The *integral types* include the boolean type **`bool`**, enumeration types defined with the **`enum`** keyword, three character types, and six explicit integer types. The three *floating-point types* are **`float`**, **`double`**, and **`long double`**. The most frequently used fundamental types are **`bool`**, **`char`**, **`int`**, and **`double`**.

## 2.2 THE BOOLEAN TYPE

A *boolean* type is an integral type whose variables can have only two values: **false** and **true**. These values are stored as the integers 0 and 1. The boolean type in Standard C++ is named **bool**.

### EXAMPLE 2.1 Boolean Variables

```
int main()
{ // prints the value of a boolean variable:
  bool flag=false;
  cout << "flag = " << flag << endl;
  flag = true;
  cout << "flag = " << flag << endl;
}
```

```
flag = 0
flag = 1
```

Note that the value `false` is printed as the integer `0` and the value `true` is printed as the integer `1`.

## 2.3 ENUMERATION TYPES

In addition to the predefined types such as **int** and **char**, C++ allows you to define your own special data types. This can be done in several ways, the most powerful of which use classes as described in Chapter 11. We consider here a much simpler kind of user-defined type.

An *enumeration type* is an integral type that is defined by the user with the syntax

**enum** *typename* **{** *enumerator-list* **};**

Here **enum** is a C++ keyword, *typename* stands for an identifier that names the type being defined, and *enumerator-list* stands for a list of names for integer constants. For example, the following defines the enumeration type `Semester`, specifying the three possible values that a variable of that type can have

```
enum Semester {FALL, SPRING, SUMMER};
```

We can then declare variables of this type:

```
Semester s1, s2;
```

and we can use those variables and those type values as we would with predefined types:

```
s1 = SPRING;
s2 = FALL;
if (s1 == s2) cout << "Same semester." << endl;
```

The actual values defined in the enumerator-list are called *enumerators*. In fact, they are ordinary integer constants. For example, the enumerators `FALL`, `SPRING`, and `SUMMER` that are defined for the `Semester` type above could have been defined like this:

```
const int FALL=0;
const int WINTER=1;
const int SUMMER=2;
```

The values 0, 1, … are assigned automatically when the type is defined. These default values can be overridden in the *enumerator-list*:

```
enum Coin {PENNY=1, NICKEL=5, DIME=10, QUARTER=25};
```

If integer values are assigned to only some of the enumerators, then the ones that follow are given consecutive values. For example,

```
enum Month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV
            DEC};
```

will assign the numbers 1 through 12 to the twelve months.

Since enumerators are simply integer constants, it is legal to have several different enumerators with the same value:

```
enum Answer {NO = 0, FALSE=0, YES = 1, TRUE=1, OK = 1};
```

This would allow the code

```
int answer;
cin >> answer;
  ⋮
  ⋮
if (answer == YES) cout << "You said it was o.k." << endl;
```

to work as expected. If the value of the variable `answer` is 1, then the condition will be true and the output will occur. Note that since the integer value 1 always means "true" in a condition, this selection statement could also be written

```
if (answer) cout << "You said it was o.k." << endl;
```

Notice the conspicuous use of capitalization here. Most programmers usually follow these conventions for capitalizing their identifiers:

1. Use only upper-case letters in names of constants.
2. Capitalize the first letter of each name in user-defined types.
3. Use all lower-case letters everywhere else.

These rules make it easier to distinguish the names of constants, types, and variables, especially in large programs. Rule 2 also helps distinguish standard C++ types like `float` and `string` from user-defined types like `Coin` and `Month`.

Enumeration types are usually defined to make code more *self-documenting*; *i.e.*, easier for humans to understand. Here are a few more typical examples:

```
enum Sex {FEMALE, MALE};
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
enum Radix {BIN=2, OCT=8, DEC=10, HEX=16};
enum Color {RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET};
enum Rank {TWO=2, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
           JACK, QUEEN, KING, ACE};
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
enum Roman {I=1, V=5, X=10, L=50, C=100, D=500, M=1000};
```

Definitions like these can help make your code more readable. But enumerations should not be overused. Each enumerator in an enumerator list defines a new identifier. For example, the definition of `Roman` above defines the seven identifiers `I`, `V`, `X`, `L`, `C`, `D`, and `M` as specific integer constants, so these letters could not be used for any other purpose within the scope of their definition.

Note that enumerators must be valid identifiers. So for example, this definition would not be valid

```
enum Grade {F, D, C-, C, C+, B-, B, B+, A-, A};          // ERRONEOUS
```

because the characters `'+'` and `'-'` cannot be used in identifiers. Also, the definitions for `Month` and `Radix` shown above could not both be in the same scope because they both define the symbol `OCT`.

Enumerations can also be anonymous in C++:

```
enum {I=1, V=5, X=10, L=50, C=100, D=500, M=1000};
```

This is just a convenient way to define integer constants.

## 2.4 CHARACTER TYPES

A *character type* is an integral type whose variables represent characters like the letter `'A'` or the digit `'8'`. Character literals are delimited by the apostrophe (`'`). Like all integral type values, character values are stored as integers.

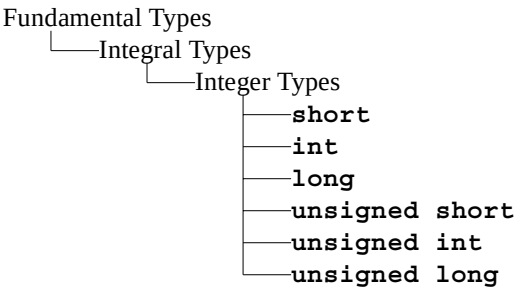
### EXAMPLE 2.2 Character Variables

```
int main()
{ // prints the character and its internally stored integer value:
  char c='A';
  cout << "c = " << c << ", int(c) = " << int(c) << endl;
  c='t';
  cout << "c = " << c << ", int(c) = " << int(c) << endl;
  c='\t';  // the tab character
  cout << "c = " << c << ", int(c) = " << int(c) << endl;
  c='!';
  cout << "c = " << c << ", int(c) = " << int(c) << endl;
}
```

```
c = A, int(c) = 65
c = t, int(c) = 116
c =     , int(c) = 9
c = !, int(c) = 33
```

Since character values are used for input and output, they appear in their character form instead of their integral form: the character `'A'` is printed as the letter "A", not as the integer 65 which is its internal representation. The *type cast operator* `int()` is used here to reveal the corresponding integral value. These are the characters' ASCII codes. (See Appendix A.)

## 2.5 INTEGER TYPES




There are 6 integer types in Standard C++: These types actually have several names. For example, **`short`** is also named **`short int`**, and **`int`** is also named **`signed int`**.

You can determine the numerical ranges of the integer types on your system by running the program in the following example.

### EXAMPLE 2.3 Integer Type Ranges

This program prints the numeric ranges of the 6 integer types in C++:

```
#include <iostream>
#include <climits>    // defines the constants SHRT_MIN, etc.
using namespace std;
int main()
{ // prints some of the constants stored in the <climits> header:
  cout << "minimum short = " << SHRT_MIN << endl;
  cout << "maximum short = " << SHRT_MAX << endl;
```

```
    cout << "maximum unsigned short = 0" << endl;
    cout << "maximum unsigned short = " << USHRT_MAX << endl;
    cout << "minimum int = " << INT_MIN << endl;
    cout << "maximum int = " << INT_MAX << endl;
    cout << "minimum unsigned int = 0" << endl;
    cout << "maximum unsigned int = " << UINT_MAX << endl;
    cout << "minimum long= " << LONG_MIN << endl;
    cout << "maximum long= " << LONG_MAX << endl;
    cout << "minimum unsigned long = 0" << endl;
    cout << "maximum unsigned long = " << ULONG_MAX << endl;
}
```

```
minimum short = -32768
maximum short = 32767
maximum unsigned short = 0
maximum unsigned short = 65535
minimum int = -2147483648
maximum int = 2147483647
minimum unsigned int= 0
maximum unsigned int= 4294967295
minimum long = -2147483648
maximum long = 2147483647
minimum unsigned long = 0
maximum unsigned long = 4294967295
```

The header file `<climits>` defines the constants `SHRT_MIN`, `SHRT_MAX`, `USHRT_MIN`, *etc.* These are the limits on the range of values that a variable of the indicated type can have. For example, the output shows that variables of type **`int`** can have values in the range –2,147,483,648 to 2,147,483,647 on this computer.

On this computer, the three **`signed`** integer types have the same range as their corresponding unqualified integer type. For example, **`signed short int`** is the same as **`short int`**. This tells us that the **`signed`** integer types are redundant on this computer.

The output also reveals that the range of the **`int`** type (–2,147,483,648 to 2,147,483,647) is the same as that of the **`long int`** type, and that the range of the **`unsigned int`** type (0 to 4,294,967,295) is the same as that of the **`unsigned long int`** type. This tells us that the **`long`** integer types are redundant on this computer.

The output from Example 2.3 shows that on this computer (a Pentium II PC running the Windows 98 operating system and the CodeWarrior 3.2 C++ compiler), the six integer types have the following ranges:

| | | |
|---|---|---|
| **`short`**: | –32,768 to 32,767; | ($2^8$ values ⇒ 1 byte) |
| **`int`**: | –2,147,483,648 to 2,147,483,647; | ($2^{32}$ values ⇒ 4 bytes) |
| **`long`**: | –2,147,483,648 to 2,147,483,647; | ($2^{32}$ values ⇒ 4 bytes) |
| **`unsigned short`**: | 0 to 65,535; | ($2^8$ values ⇒ 1 byte) |
| **`unsigned int`**: | 0 to 4,294,967,295; | ($2^{32}$ values ⇒ 4 bytes) |
| **`unsigned long`**: | 0 to 4,294,967,295; | ($2^{32}$ values ⇒ 4 bytes) |

Note that **`long`** is the same as **`int`** and **`unsigned long`** is the same as **`unsigned int`**.

The **`unsigned`** integer types are used for bit strings. A *bit string* is a string of 0s and 1s as is stored in the computer's random access memory (RAM) or on disk. Of course, everything stored in a computer, in RAM or on disk, is stored as 0s and 1s. But all other types of data are formatted; *i.e.*, interpreted as something such as a signed integer or a string of characters.

## 2.6 ARITHMETIC OPERATORS

Computers were invented to perform numerical calculations. Like most programming languages, C++ performs its numerical calculations by means of the five *arithmetic operators* `+`, `-`, `*`, `/`, and `%`.

### EXAMPLE 2.4 Integer Arithmetic

This example illustrates how the arithmetic operators work.

```
int main()
{ // tests operators +, -, *, /, and %:
  int m=54;
  int n=20;
  cout << "m = " << m << " and n = " << n << endl;
  cout << "m+n = " << m+n << endl;  // 54+20 = 74
  cout << "m-n = " << m-n << endl;  // 54-20 = 34
  cout << "m*n = " << m*n << endl;  // 54*20 = 1080
  cout << "m/n = " << m/n << endl;  // 54/20 = 2
  cout << "m%n = " << m%n << endl;  // 54%20 = 14
}
```

```
m = 54 and n = 20
m+n = 74
m-n = 34
m*n = 1080
m/n = 2
m%n = 14
```

Note that integer division results in another integer: 54/20 = 2, not 2.7.

The last two operators used in Example 2.4 are the *division operator* `/` and the *modulus operator* `%` (also called the *remainder operator*). The modulus operator results in the remainder from the division. Thus, 54%20 = 14 because 14 is the remainder after 54 is divided by 20.

## 2.7 THE INCREMENT AND DECREMENT OPERATORS

The values of integral objects can be incremented and decremented with the `++` and `--` operators, respectively. Each of these operators has two versions: a “pre” version and a “post” version. The “pre” version performs the operation (either adding 1 or subtracting 1) on the object before the resulting value is used in its surrounding context. The “post” version performs the operation after the object’s current value has been used.

### EXAMPLE 2.5 Applying the Pre-increment and Post-increment Operators

```
int main()
{ // shows the difference between m++ and ++m:
  int m, n;
  m = 44;
  n = ++m;  // the pre-increment operator is applied to m
  cout << "m = " << m << ", n = " << n << endl;
```

```
  m = 44;
  n = m++;  // the post-increment operator is applied to m
  cout << "m = " << m << ", n = " << n << endl;
}
m = 45, n = 45
m = 45, n = 44
```

The line

```
n = ++m;  // the pre-increment operator is applied to m
```

increments `m` to 45 and then assigns that value to `n`. So both variables have the same value 45 when the next output line executes.

The line

```
n = m++;  // the post-increment operator is applied to m
```

increments `m` to 45 only after it has assigned the value of `m` to `n`. So `n` has the value 44 when the next output line executes.

## 2.8 COMPOSITE ASSIGNMENT OPERATORS

The standard assignment operator in C++ is the equals sign `=`. In addition to this operator, C++ also includes the following *composite assignment operators*: `+=`, `-=`, `*=`, `/=`, and `%=`. When applied to a variable on the left, each applies the indicated arithmetic operation to it using the value of the expression on the right.

### EXAMPLE 2.6 Applying Composite Arithmetic Assignment Operators

```
int main()
{ // tests arithmetic assignment operators:
  int n=22;
  cout << "n = " << n << endl;
  n += 9;  // adds 9 to n
  cout << "After n += 9, n = " << n << endl;
  n -= 5;  // subtracts 5 from n
  cout << "After n -= 5, n = " << n << endl;
  n *= 2;  // multiplies n by 3
  cout << "After n *= 2, n = " << n << endl;
  n /= 3;  // divides n by 9
  cout << "After n /= 3, n = " << n << endl;
  n %= 7;  // reduces n to the remainder from dividing by 4
  cout << "After n %= 7, n = " << n << endl;
}
n = 22
After n += 9, n = 31
After n -= 5, n = 26
After n *= 2, n = 52
After n /= 3, n = 17
After n %= 7, n = 3
```

## 2.9 FLOATING-POINT TYPES

C++ supports three real number types: **float**, **double**, and **long double**. On most systems, **double** uses twice as many bytes as **float**. Typically, **float** uses 4 bytes, **double** uses 8 bytes, and **long double** uses 8, 10, 12, or 16 bytes.

Types that are used for real numbers are called "floating-point" types because of the way they are stored internally in the computer. On most systems, a number like 123.45 is first converted to binary form:

$$123.45 = 1111011.01110011_2 \times 2^7$$

Then the point is "floated" so that all the bits are on its right. In this example, the floating-point form is obtained by floating the point 7 bits to the left, producing a mantissa $2^7$ times smaller. So the original number is

$$123.45 = 0.111101101110011_2 \times 2^7$$

This number would be represented internally by storing the mantissa `111101101110011` and the exponent 7 separately. For a 32-bit **float** type, the *mantissa* is stored in a 23-bit segment and the exponent in an 8-bit segment, leaving 1 bit for the sign of the number. For a 64-bit **double** type, the mantissa is stored in a 52-bit segment and the exponent in an 11-bit segment.

### EXAMPLE 2.7 Floating-Point Arithmetic

This program is nearly the same as the one in Example 2.4. The important difference is that these variables are declared to have the floating-point type `double` instead of the integer type `int`.

```
int main()
{ // tests the floating-point operators +, -, *, and /:
  double x=54.0;
  double y=20.0;
  cout << "x = " << x << " and y = " << y << endl;
  cout << "x+y = " << x+y << endl;  // 54.0+20.0 = 74.0
  cout << "x-y = " << x-y << endl;  // 54.0-20.0 = 34.0
  cout << "x*y = " << x*y << endl;  // 54.0*20.0 = 1080.0
  cout << "x/y = " << x/y << endl;  // 54.0/20.0 = 2.7
}
```

```
x = 55 and y = 20
x+y = 75
x-y = 35
x*y = 1100
x/y = 2.7
```

Unlike integer division, floating-point division does not truncate the result: 54.0/20.0 = 2.7.

The next example can be used on any computer to determine how many bytes it uses for each type. The program uses the **sizeof** operator which returns the size in bytes of the type specified.

### EXAMPLE 2.8 Using the `sizeof` Operator

This program tells you how much space each of the 12 fundamental types uses:

```
int main()
{ // prints the storage sizes of the fundamental types:
  cout << "Number of bytes used:\n";
```

```
  cout << "\t          char: " <<           sizeof(char) << endl;
  cout << "\t         short: " <<          sizeof(short) << endl;
  cout << "\t           int: " <<            sizeof(int) << endl;
  cout << "\t          long: " <<           sizeof(long) << endl;
  cout << "\t unsigned char: " << sizeof(unsigned char) << endl;
  cout << "\tunsigned short: " << sizeof(unsigned short) << endl;
  cout << "\t  unsigned int: " <<  sizeof(unsigned int) << endl;
  cout << "\t unsigned long: " << sizeof(unsigned long) << endl;
  cout << "\t   signed char: " <<   sizeof(signed char) << endl;
  cout << "\t         float: " <<         sizeof(float) << endl;
  cout << "\t        double: " <<        sizeof(double) << endl;
  cout << "\t   long double: " <<   sizeof(long double) << endl;
}
```

```
Number of bytes used:
                  char: 1
                 short: 2
                   int: 4
                  long: 4
         unsigned char: 1
        unsigned short: 2
          unsigned int: 4
         unsigned long: 4
           signed char: 1
                 float: 4
                double: 8
           long double: 8
```

The output below shows the sizes for a typical UNIX workstation. On this machine, **int** and **long** are equivalent, **unsigned int** and **unsigned long** are equivalent, and **double** and **long double** are equivalent. In other words, 'long' is no different from 'regular' on this computer.

The next program can be used to investigate floating-point types on any computer system. It reads the values of various constants from the **<cfloat>** header file. To access it, the program must include the preprocessor directive:

```
#include <cfloat>
```

This is like the **#include <iostream>** directive that we always include in order to use the **cin** and **cout** objects.

### EXAMPLE 2.9 Reading from the <cfloat> Header File

This program tells you the precision and magnitude range that the **float** type has on your system:

```
#include <cfloat>  // defines the FLT constants
#include <iostream>  // defines the FLT constants
using namespace std;
int main()
{ // prints the storage sizes of the fundamental types:
  int fbits = 8*sizeof(float);        // each byte contains 8 bits
  cout << "float uses " << fbits << " bits:\n\t"
       << FLT_MANT_DIG - 1 << " bits for its mantissa,\n\t "
       << fbits - FLT_MANT_DIG << " bits for its exponent,\n\t "
       << 1 << " bit for its sign\n"
       << "          to obtain: " << FLT_DIG << " sig. digits\n"
```

```
        << " with minimum value: " << FLT_MIN << endl
        << "  and maximum value: " << FLT_MAX << endl;
}
```

```
float uses 32 bits:
        23 bits for its mantissa,
         8 bits for its exponent,
         1 bit for its sign
          to obtain: 6 sig. digits
 with minimum value: 1.17549e-38
  and maximum value: 3.40282e+38
```

The constants `FLT_MANT_DIG`, `FLT_DIG`, `FLT_MIN`, and `FLT_MAX` are defined in the `<cfloat>` header file.

This output is from a UNIX workstation. It shows that the 32 bits it uses to store a `float` are partitioned into 3 parts: 23 bits for the mantissa, 8 bits for the exponent, and 1 bit for the sign. The 23-bit mantissa produces a floating-point value with 6 significant digits, and the 8-bit exponent yields a range in magnitude from about $10^{-37}$ to about $3 \times 10^{38}$. *i.e.*,

$$0.00000000000000000000000000000000000001 < |x| < 300{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000$$

for any variable *x* declared to have type **`float`**.

All floating-point arithmetic is done in **`double`** precision. So the only time you should use **`float`** instead of **`double`** is when you are storing large quantities of real numbers and are concerned about storage space or access time.

## 2.10 TYPE CONVERSIONS

We saw in Chapter 1 how one integer type can be converted automatically to another. C++ also converts integral types into floating point types when they are expected. For example,

```
int n = 22;
float x = 3.14159;
x += n;          // the value 22 is automatically converted to 22.0
cout << x - 2 << endl; // value 2 is automatically converted to 2.0
```

Converting from integer to float like this is what one would expect and is usually taken for granted. But converting from a floating point type to an integral type is not automatic.

In general, if `T` is one type and `v` is a value of another type, then the expression

```
T(v)
```

converts `v` to type `T`. This is called *type casting*. For example, if *`expr`* is a floating-point expression and `n` is a variable of type **`int`**, then

```
n = int(expr);
```

converts the value of *`expr`* to type **`int`** and assigns it to `n`. The effect is to remove the real number's fractional part, leaving only its whole number part to be assigned to `n`. For example, 2.71828 would be converted to 2. Note that this is *truncating*, not *rounding*.

### EXAMPLE 2.10 Simple Type Casting

This program casts a `double` value into `int` value:

```
int main()
{ // casts a double value as an int:
  double v = 1234.56789;
  int n = int(v);
```

```
  cout << "v = " << v << ", n = " << n << endl;
}
v = 1234.57, n = 1234
```

The `double` value 1234.56789 is converted to the **`int`** value 1234.

When one type is to be converted to a "higher" type, the type case operator is not needed. This is called *type promotion*. Here's a simple example of *promotion* from **`char`** all the way up to **`double`**:

### EXAMPLE 2.11 Promotion of Types

This program promotes a `char` to a `short` to an `int` to a `float` to a `double`:

```
int main()
{ // prints promoted vales of 65 from char to double:
  char c='A';  cout << "  char c = " << c << endl;
  short k=c;   cout << " short k = " << k << endl;
  int m=k;     cout << "   int m = " << m << endl;
  long n=m;    cout << "  long n = " << n << endl;
  float x=m;   cout << " float x = " << x << endl;
  double y=x;  cout << "double y = " << y << endl;
}
  char c = A
 short k = 65
   int m = 65
  long n = 65
 float x = 65
double y = 65
```

The integer value of the character `'A'` is its ASCII code 65. This value is converted as a **`char`** in `c`, a **`short`** in `k`, an **`int`** in `m`, and a **`long`** in `n`. The value is then converted to the floating point value 65.0 and stored as a **`float`** in `x` and as a **`double`** in `y`. Notice that **`cout`** prints the integer `c` as a character, and that it prints the real numbers `x` and `y` as integers because their fractional parts are 0.

Because it is so easy to convert between integer types and real types in C++, it is easy to forget the distinction between them. In general, integers are used for counting discrete things, while reals are used for measuring on a continuous scale. This means that integer values are exact, while real values are approximate.

Note that type casting and promotion convert the type of the value of a variable or expression, but it does not change the type of the variable itself.

In the C programming language, the syntax for casting `v` as type `T` is `(T) v`. C++ inherits this form also, so we could have done `n = int(v)` as `n = (int) v`.

## 2.11 NUMERIC OVERFLOW

On most computers the `long int` type allows 4,294,967,296 different values. That's a lot of values, but it's still finite. Computers are finite, so the range of any type must also be finite. But in mathematics there are infinitely many integers. Consequently, computers are manifestly prone to error when their numeric values become too large. That kind of error is called *numeric overflow.*

### EXAMPLE 2.12 Integer Overflow

This program repeatedly multiplies `n` by 1000 until it overflows.

```
int main()
{ // prints n until it overflows:
  int n=1000;
  cout << "n = " << n << endl;
  n *= 1000;  // multiplies n by 1000
  cout << "n = " << n << endl;
  n *= 1000;  // multiplies n by 1000
  cout << "n = " << n << endl;
  n *= 1000;  // multiplies n by 1000
  cout << "n = " << n << endl;
}
```

```
n = 1000
n = 1000000
n = 1000000000
n = -727379968
```

This shows that the computer that ran this program cannot multiply 1,000,000,000 by 1000 correctly.

### EXAMPLE 2.13 Floating-point Overflow

This program is similar to the one in Example 2.12. It repeatedly squares `x` until it overflows.

```
int main()
{ // prints x until it overflows:
  float x=1000.0;
  cout << "x = " << x << endl;
  x *= x;  // multiplies n by itself; i.e., it squares x
  cout << "x = " << x << endl;
  x *= x;  // multiplies n by itself; i.e., it squares x
  cout << "x = " << x << endl;
  x *= x;  // multiplies n by itself; i.e., it squares x
  cout << "x = " << x << endl;
  x *= x;  // multiplies n by itself; i.e., it squares x
  cout << "x = " << x << endl;
}
```

```
x = 1000
x = 1e+06
x = 1e+12
x = 1e+24
x = inf
```

This shows that, starting with `x` = 1000, this computer cannot square `x` correctly more than three times. The last output is the special symbol `inf` which stands for “infinity.”

Note the difference between integer overflow and floating-point overflow. The last output in Example 2.12 is the negative integer –727,379,968 instead of the correct value of 1,000,000,000,000 = $10^{12}$. The last output in Example 2.13 is the infinity symbol `inf` instead of the correct value of $10^{48}$. Integer overflow “wraps around” to negative integers. Floating-point overflow “sinks” into the abstract notion of infinity.

## 2.12 ROUND-OFF ERROR

Round-off error is another kind of error that often occurs when computers do arithmetic on rational numbers. For example, the number 1/3 might be stored as 0.333333, which is not exactly equal to 1/3. The difference is called *round-off error.* In some cases, these errors can cause serious problems.

### EXAMPLE 2.14 Round-off Error

This program does some simple arithmetic to illustrate roundoff error:

```
int main()
{ // illustrates round-off error::
  double x = 1000/3.0;cout << "x = " << x << endl;  // x = 1000/3
  double y = x - 333.0;cout << "y = " << y << endl; // y = 1/3
  double z = 3*y - 1.0;cout << "z = " << z << endl; // z = 3(1/3) - 1
  if (z == 0) cout << "z == 0.\n";
  else cout << "z does not equal 0.\n";             // z != 0
}
x = 333.333
y = 0.333333
z = -5.68434e-14
z does not equal 0.
```

In exact arithmetic, the variables would have the values $x = 333\ 1/3$, $y = 1/3$, and $z = 0$. But 1/3 cannot be represented exactly as a floating-point value. The inaccuracy is reflected in the residue value for `z`.

Example 2.14 illustrates an inherent problem with using floating-point types within conditional tests of equality. The test `(z == 0)` will fail even if `z` is very nearly zero, which is likely to happen when `z` should algebraically be zero. So it is better to avoid tests for equality with floating-point types.

The next example shows that round-off error can be difficult to recognize.

### EXAMPLE 2.15 Hidden Round-off Error

This program implements the *quadratic formula* to solve quadratic equations.

```
#include <cmath>  // defines the sqrt() function
#include <iostream>
using namespace std;
int main()
{ // implements the quadratic formula
  float a, b, c;
  cout << "Enter the coefficients of a quadratic equation:" << endl;
  cout << "\ta: ";
  cin >> a;
  cout << "\tb: ";
  cin >> b;
  cout << "\tc: ";
  cin >> c;
  cout << "The equation is: " << a << "*x*x + " << b
       << "*x + " << c << " = 0" << endl;
```

```
    float d = b*b - 4*a*c;  // discriminant
    float sqrtd = sqrt(d);
    float x1 = (-b + sqrtd)/(2*a);
    float x2 = (-b - sqrtd)/(2*a);
    cout << "The solutions are:" << endl;
    cout << "\tx1 = " << x1 << endl;
    cout << "\tx2 = " << x2 << endl;
    cout << "Check:" << endl;
    cout << "\ta*x1*x1 + b*x1 + c = " <<  a*x1*x1 + b*x1 + c << endl;
    cout << "\ta*x2*x2 + b*x2 + c = " <<  a*x2*x2 + b*x2 + c << endl;
}
```

The quadratic formula requires computing the square root $\sqrt{b^2 - 4ac}$. This is done on the line

```
float sqrtd = sqrt(d);
```

which calls the square root function `sqrt()` defined in the header file `<cmath>`. The last two lines of the program check the solutions by substituting them back into the original quadratic equation. If the resulting expression on the left evaluates to 0 then the solutions are correct.

This run solves the equation $2x^2 + 1x - 3 = 0$ correctly:

```
Enter the coefficients of a quadratic equation:
        a: 2
        b: 1
        c: -3
The equation is: 2*x*x + 1*x + -3 = 0
The solutions are:
        x1 = 1
        x2 = -1.5
Check:
        a*x1*x1 + b*x1 + c = 0
        a*x2*x2 + b*x2 + c = 0
```

But this run attempts to solve the equation $x^2 + 10000000000x + 1 = 0$ and fails:

```
Enter the coefficients of a quadratic equation:
        a: 1
        b: 1e10
        c: 1
The equation is: 1*x*x + 1e10*x + 1 = 0
The solutions are:
        x1 = 0
        x2 = -1e10
Check:
        a*x1*x1 + b*x1 + c = 1
        a*x2*x2 + b*x2 + c = 1
```

The first solution, $x_1 = 0$, is obviously incorrect: the resulting quadratic expression $ax_1^2 + bx_1 + c$ evaluates to 1 instread of 0. The second solution, $x_2 = -1e10 = -10{,}000{,}000{,}000$ is even worse. The correct solutions are $x_1 = -0.00000\ 00000\ 99999\ 99999\ 99999\ 99519$ and $x_2 = 9{,}999{,}999{,}999.99999\ 99999$.

Numeric overflow and round-off errors are examples of *run-time errors*, which are errors that occur while the program is running. Such errors are more serious than compile-time errors such as neglecting to declare a variable or forgetting a semicolon because they are usually harder to detect and locate. Compile-time errors are caught by the compiler, which usually gives a pretty good report on where they are. But run-time errors are detected only when the user notices that the results are incorrect. Even if the program crashes, it still may be difficult to find where the problem is in the program.

### EXAMPLE 2.16 Other Kinds of Run-Time Errors

Here are two more runs of the quadratic formula program in Example 2.15:

```
Enter the coefficients of a quadratic equation:
        a: 1
        b: 2
        c: 3
The equation is: 1*x*x + 2*x + 3 = 0
The solutions are:
        x1 = nan
        x2 = nan
Check:
        a*x1*x1 + b*x1 + c = nan
        a*x2*x2 + b*x2 + c = nan
```

The quadratic equation $1x^2 + 2x + 3 = 0$ has no real solution because the discriminant $b^2 - 4ac$ is negative. When the program runs, the square root function `sqrt(d)` fails because `d` < 0. It returns the symbolic constant `nan` which stands for "not a number." Then every subsequent numeric operation that uses this constant results in the same value. That's why the check values come out as `nan` at the end of the run.

This run attempts to solve the equation $0x^2 + 2x + 5 = 0$. That equation has the solution $x = 2.5$. But the quadratic formula fails because $a = 0$:

```
Enter the coefficients of a quadratic equation:
        a: 0
        b: 2
        c: 5
The equation is: 0*x*x + 2*x + 5 = 0
The solutions are:
        x1 = nan
        x2 = -inf
Check:
        a*x1*x1 + b*x1 + c = nan
        a*x2*x2 + b*x2 + c = nan
```

Notice that $x_1$ comes out as `nan`, but $x_2$ comes out as `-inf`. The symbol `inf` stands for "infinity." That's what you get when you divide a nonzero number by zero. The quadratic formula computes $x_2$ as

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} = \frac{-(2) - \sqrt{(2)^2 - 4(0)(5)}}{2(0)} = \frac{-2-2}{0} = \frac{-4}{0}$$

which becomes `-inf`. But it computes $x_1$ as

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-(2) + \sqrt{(2)^2 - 4(0)(5)}}{2(0)} = \frac{-2+2}{0} = \frac{0}{0}$$

which becomes `nan`.

The three symbols `inf`, `-inf`, and `nan` are numeric constants. The usual numeric operators can be applied to them, although the results are usually useless. For example, you can multiply `nan` by any number, but the result will still be `nan`.

## 2.13 THE E-FORMAT FOR FLOATING-POINT VALUES

When input or output, floating-point values may be specified in either of two formats: *fixed-point* and *scientific*. The output in Example 2.16 illustrates both: `333.333` has fixed-point format, and `-5.68434e-14` has scientific format.

In scientific format, the letter `e` stands for "exponent on 10." So `e-14` means $10^{-14}$, and thus `-5.68434e-14` means $-5.68434 \times 10^{-14} = -0.000000000000568434$. Obviously, the scientific format is more efficient for very small or very large numbers.

Floating-point values with magnitude in the range 0.1 to 999,999 will normally be printed in fixed-point format; all others will be printed in scientific format.

### EXAMPLE 2.17 Scientific Format

This program shows how floating-point values may be input in scientific format:

```
int main()
{ // prints double values in scientific e-format:
  double x;
  cout << "Enter float: ";  cin >> x;
  cout << "Its reciprocal is: " << 1/x << endl;
}
```

```
Enter float: 234.567e89
Its reciprocal is: 4.26317e-92
```

You can use either `e` or `E` in the scientific format.

## 2.14 SCOPE

The *scope* of an identifier is that part of the program where it can be used. For example, variables cannot be used before they are declared, so their scopes begin where they are declared. This is illustrated by the next example.

### EXAMPLE 2.18 Scope of Variables

```
int main()
{ // illustrates the scope of variables:
  x = 11;    // ERROR: this is not in the scope of x
  int x;
  { x = 22;  // OK: this is in the scope of x
    y = 33;  // ERROR: this is not in the scope of y
    int y;
    x = 44;  // OK: this is in the scope of x
    y = 55;  // OK: this is in the scope of y
  }
  x = 66;    // OK: this is in the scope of x
  y = 77;    // ERROR: this is not in the scope of y
}
```

The scope of `x` extends from the point where it is declared to the end of `main()`. The scope of `y` extends from the point where it is declared to the end of the internal block within which it is declared.

A program may have several objects with the same name as long as their scopes are nested or disjoint. This is illustrated by the next example.

### EXAMPLE 2.19 Nested and Parallel Scopes

```
int x = 11;                                          // this x is global

int main()
{ // illustrates the nested and parallel scopes:
  int x = 22;
  { // begin scope of internal block
    int x = 33;
    cout << "In block inside main(): x = " << x << endl;
  }                                          // end scope of internal block
  cout << "In main(): x = " << x << endl;
  cout << "In main(): ::x = " << ::x << endl;
}                                                    // end scope of main()
In block inside main(): x = 33
In main(): x = 22
In main(): ::x = 11
```

There are three different objects named `x` in this program. The `x` that is initialized with the value 11 is a global variable, so its scope extends throughout the file. The `x` that is initialized with the value 22 has scope limited to `main()`. Since this is nested within the scope of the first `x`, it hides the first `x` within `main()`. The `x` that is initialized with the value 33 has scope limited to the internal block within `main()`, so it hides both the first and the second `x` within that block.

The last line in the program uses the *scope resolution operator* **`::`** to access the global `x` that is otherwise hidden in `main()`.

## Review Questions

**2.1** Write a single C++ statement that prints `"Too many"` if the variable `count` exceeds 100.

**2.2** What is wrong with the following code:

***a.*** `cin << count;`

***b.***
```
if x < y min = x
  else min = y;
```

**2.3** What is wrong with this code:

```
cout << "Enter n: ";
cin >> n;
if (n < 0)
  cout << "That is negative. Try again." << endl;
  cin >> n;
else
  cout << "o.k. n = " << n << endl;
```

**2.4** What is the difference between a reserved word and a standard identifier?

**2.5** What is wrong with this code:

```
enum Semester {FALL, SPRING, SUMMER};
enum Season {SPRING, SUMMER, FALL, WINTER};
```

**2.6** What is wrong with this code:

```
enum Friends {"Jerry", "Henry", "W.D."};
```

## Problems

**2.1** Write and run a program like the one in Example 2.2 on page 19 that prints the ASCII codes for only the 10 upper case and lower case vowels. Use Appendix A to check your output.

**2.2** Modify the program in Example 2.15 on page 28 so that it uses type `double` instead of `float`. Then see how much better it performs on the input that illustrated round-off error.

**2.3** Write and run a program to find which, if any, arithmetic operations can be applied to a variable that will change its value from any of the three numeric constants `inf`, `-inf`, and `nan` to something else.

**2.4** Write a program that converts inches to centimeters. For example, if the user enters 16.9 for a length in inches, the output would be `42.926 cm`. (One inch equals 2.54 centimeters.)

## Answers to Review Questions

**2.1** `if (count > 100) cout << "Too many";`

**2.2** ***a.*** Either `cout` should be used in place of `cin`, or the extraction operator `>>` should be used in place of the insertion operator `<<`.

***b.*** Parentheses are required around the condition `x < y`, and a semicolon is required at the end of the `if` clause before the `else`.

**2.3** There is more than one statement between the `if` clause and the `else` clause. They need to be made into a compound statement by enclosing them in braces { }.

**2.4** A *reserved word* is a keyword in a programming language that serves to mark the structure of a statement. For example, the keywords `if` and `else` are reserved words. A *standard identifier* is a keyword that defines a type. Among the 63 keywords in C++, `if`, `else`, and `while` are some of the reserved words, and `char`, `int`, and `float` are some of the standard identifiers.

**2.5** The second `enum` definition attempts to redefine the constants `SPRING`, `SUMMER`, and `FALL`.

**2.6** Enumerators must be valid identifiers. String literals like `"Jerry"` and `"Henry"` are not identifiers.

## Solutions to Problems

**2.1**

```
int main()
{ // prints the ASCII codes of the vowels
  cout << "int('A') = " << int('A') << endl;
  cout << "int('E') = " << int('E') << endl;
  cout << "int('I') = " << int('I') << endl;
  cout << "int('O') = " << int('O') << endl;
  cout << "int('U') = " << int('U') << endl;
  cout << "int('a') = " << int('a') << endl;
  cout << "int('e') = " << int('e') << endl;
  cout << "int('i') = " << int('i') << endl;
  cout << "int('o') = " << int('o') << endl;
  cout << "int('u') = " << int('u') << endl;
}
```

```
int('A') = 65
int('E') = 69
int('I') = 73
int('O') = 79
int('U') = 85
```

```
int('a') = 97
int('e') = 101
int('i') = 105
int('o') = 111
int('u') = 117
```

**2.2**

```
int main()
{ // implements the quadratic formula
  double a, b, c;
  cout << "Enter the coefficients:" << endl;
  cout << "\ta: ";
  cin >> a;
  cout << "\tb: ";
  cin >> b;
  cout << "\tc: ";
  cin >> c;
  cout << "The equation is: " << a << "*x*x + " << b
       << "*x + " << c << " = 0" << endl;
  double d = b*b - 4*a*c;
  double sqrtd = sqrt(d);
  double x1 = (-b + sqrtd)/(2*a);
  double x2 = (-b - sqrtd)/(2*a);
  cout << "The solutions are:" << endl;
  cout << "\tx1 = " << x1 << endl;
  cout << "\tx2 = " << x2 << endl;
  cout << "Check:" << endl;
  cout << "\ta*x1*x1 + b*x1 + c = " <<  a*x1*x1 + b*x1 + c << endl;
  cout << "\ta*x2*x2 + b*x2 + c = " <<  a*x2*x2 + b*x2 + c << endl;
}
```

```
Enter the coefficients of a quadratic equation:
        a: 2
        b: 8.001
        c: 8.002
The equation is: 2*x*x + 8.001*x + 8.002 = 0
The solutions are:
        x1 = -2
        x2 = -2.0005
Check:
        a*x1*x1 + b*x1 + c = 0
        a*x2*x2 + b*x2 + c = 0
```

**2.3** The following program changes the value of `x` from `inf` to `-inf` and *vice versa*. But no arithmetic operation will change the value of a variable once it becomes `nan`.

```
int main()
{ // changes the value of x after it becomes inf:
  float x=1e30;
  cout << "x= " << x << endl;
  x *= x;
  cout << "x= " << x << endl;
  x *= -1.0;
  cout << "x= " << x << endl;
  x *= -1.0;
  cout << "x= " << x << endl;
}
```

```
x= 1e+30
x= inf
x= -inf
x= inf
```

**2.4** We use two variables of type **`float`**

```
int main()
{ // converts inches to centimeters:
  float inches, cm;
  cout << "Enter length in inches: ";
  cin >> inches;
  cm = 2.54*inches;
  cout << inches << " inches = " << cm << " centimeters.\n";
}
```

```
Enter length in inches: 16.9
16.9 inches = 42.926 centimeters.
```

# Chapter 3

# Selection

The programs in the first two chapters all have *sequential execution*: each statement in the program executes once, and they are executed in the same order that they are listed. This chapter shows how to use selection statements for more flexible programs. It also describes the various integral types that are available in C++.

## 3.1 THE `if` STATEMENT

The **`if`** statement allows conditional execution. Its syntax is

```
if (condition) statement;
```

where *`condition`* is an integral expression and *`statement`* is any executable statement. The statement will be executed only if the value of the integral expression is nonzero. Notice the required parentheses around the condition.

### EXAMPLE 3.1 Testing for Divisibility

This program tests if one positive integer is not divisible by another:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (n%d) cout << n << " is not divisible by " << d << endl;
}
```

On the first run, we enter 66 and 7:

```
Enter two positive integers: 66 7
66 is not divisible by 7
```

The value 66%7 is computed to be 3. Since that integral value is not zero, the expression is interpreted as a true condition and consequently the divisibility message is printed.

On the second run, we enter 56 and 7:

```
Enter two positive integers: 56 7
```

The value 56%7 is computed to be 0, which is interpreted to mean “false,” so the divisibility message is not printed.

In C++, whenever an integral expression is used as a condition, the value 0 means “false” and all other values mean “true.”

The program in Example 3.1 is inadequate because it provides no affirmative information when `n` is divisible by `d`. That fault can be remedied with an **`if..else`** statement.

## 3.2 THE `if..else` STATEMENT

The **`if..else`** statement causes one of two alternative statements to execute depending upon whether the condition is true. Its syntax is

```
if (condition) statement1;
else statement2;
```

where *condition* is an integral expression and *statement1* and *statement2* are executable statements. If the value of the condition is nonzero then *statement1* will execute; otherwise *statement2* will execute.

### EXAMPLE 3.2 Testing for Divisibility Again

This program is the same as the program in Example 3.1 except that the `if` statement has been replaced by an `if..else` statement:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (n%d) cout << n << " is not divisible by " << d << endl;
  else cout << n << " is divisible by " << d << endl;
}
```

Now when we enter 56 and 7, we get an affirmative response:

```
Enter two positive integers: 56 7
56 is divisible by 7
```

Since 56%7 is zero, the expression is interpreted as being a false condition and consequently the statement after the **else** is executed.

Note that the **if..else** is only one statement, even though it requires two semicolons.

## 3.3 KEYWORDS

A *keyword* in a programming language is a word that is already defined and is reserved for a unique purpose in programs written in that language. Standard C++ now has 74 keywords:

| | | | | |
|---|---|---|---|---|
| **and** | **and_eq** | **asm** | **auto** | **bitand** |
| **bitor** | **bool** | **break** | **case** | **catch** |
| **char** | **class** | **compl** | **const** | **const_cast** |
| **continue** | **default** | **delete** | **do** | **double** |
| **dynamic_cast** | **else** | **enum** | **explicit** | **export** |
| **extern** | **dfalse** | **float** | **for** | **friend** |
| **goto** | **if** | **inline** | **int** | **long** |
| **mutable** | **namespace** | **new** | **not** | **not_eq** |
| **operator** | **or** | **or_eq** | **private** | **protected** |
| **public** | **register** | **reinterpret_cast** | **return** | **short** |
| **signed** | **sizeof** | **static** | **static_cast** | **struct** |
| **switch** | **template** | **this** | **throw** | **true** |
| **try** | **typedef** | **typeid** | **typename** | **using** |
| **union** | **unsigned** | **virtual** | **void** | **volatile** |
| **wchar_t** | **while** | **xor** | **xor_eq** | |

Keywords like **if** and **else** are found in nearly every programming language. Other keywords such as **dynamic_cast** are unique to C++. The 74 keywords of C++ include all 32 of the keywords of the C language.

There are two kinds of keywords: reserved words and standard identifiers. A *reserved word* is a keyword that serves as a structure marker, used to define the syntax of the language. The keywords **if** and **else** are reserved words. A *standard identifier* is a keyword that names a specific element of the language. The keywords **bool** and **int** are standard identifiers because they are names of standard types in C++.

See Appendix B for more information on the C++ keywords.

## 3.4 COMPARISON OPERATORS

The six *comparison operators* are

```
x < y     // x is less than y
x > y     // x is greater than y
x <= y    // x is less than or equal to y
x >= y    // x is greater than or equal to y
x == y    // x is equal to y
x != y    // x is not equal to y
```

These can be used to compare the values of expressions of any ordinal type. The resulting integral expression is interpreted as a condition that is either false or true according to whether the value of the expression is zero. For example, the expression `7*8 < 6*9` evaluates to zero, which means that the condition is false.

### EXAMPLE 3.3 The Minimum of Two Integers

This program prints the minimum of the two integers entered:

```
int main()
{ int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  if (m < n) cout << m << " is the minimum." << endl;
  else cout << n << " is the minimum." << endl;
}
```

```
Enter two integers: 77 55
55 is the minimum.
```

Note that in C++ the single equal sign "=" is the *assignment operator*, and the double equal sign "==" is the *equality operator*:

```
x = 33;   // assigns the value 33 to x
x == 33;  // evaluates to 0 (for false) unless 33 is the value of x
```

This distinction is critically important.

### EXAMPLE 3.4 A Common Programming Error

This program is erroneous:

```
int main()
{ int n;
  cout << "Enter an integer: ";
```

```
  cin >> n;
  if (n = 22) cout << n << " = 22" << endl;  // LOGICAL ERROR!
  else cout << n << " != 22" << endl;
}
Enter an integer: 77
22 = 22
```

The expression `n = 22` assigns the value 22 to `n`, changing it from its previous value of 77. But the expression `n = 22` itself is an integral expression that evaluates to 22 after it executes. Thus the condition `(n = 22)` is interpreted as being true, because only 0 yields false, so the statement before the `else` executes. The line should have been written as

```
if (n == 22) cout << n << " = 22" << endl;  // CORRECT
```

The error illustrated in Example 3.4 is called a *logical error*. This is the worst kind of error. Compile-time errors (*e.g.*, omitting a semicolon) are caught by the compiler. Run-time errors (*e.g.*, dividing by zero) are caught by the operating system. But no such help exists for catching logical errors.

### EXAMPLE 3.5 The Minimum of Three Integers

This program is similar to the one in Example 3.3 except that it applies to three integers:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  int min=n1;               // now min <= n1
  if (n2 < min) min = n2;  // now min <= n1 and min <= n2
  if (n3 < min) min = n3;  // now min <= n1, min <= n2, and min <= n3
  cout << "Their minimum is " << min << endl;
}
Enter two integers: 77 33 55
Their minimum is 33
```

The three comments track the progress of the program: `min` is initialized to equal `n1`, so it is the minimum of the set {`n1`}. After the first `if` statement executes, `min` is equal to either `n1` or `n2`, whichever is smaller, so it is the minimum of the set {`n1`, `n2`}. The last `if` statement changes the value of `min` to `n3` only if `n3` is less than the current value of `min` which is the minimum of the set {`n1`, `n2`}. So in either case, `min` becomes the minimum of the set {`n1`, `n2`, `n3`}.

## 3.5 STATEMENT BLOCKS

A *statement block* is a sequence of statements enclosed by braces **{ }**, like this:

```
{ int temp=x; x = y; y = temp; }
```

In C++ programs, a statement block can be used anywhere that a single statement can be used.

### EXAMPLE 3.6 A Statement Block within an `if` Statement

This program inputs two integers and then outputs them in increasing order:

```
int main()
{ int x, y;
  cout << "Enter two integers: ";
  cin >> x >> y;
```

```
  if (x > y) { int temp=x; x = y; y = temp; }  // swap x and y
  cout << x << " <= " << y << endl;
}
```

```
Enter two integers: 66 44
44 <= 66
```

The three statements within the statement block sort the values of x and y into increasing order by swapping them if they are out of order. Such an interchange requires three separate steps along with the temporary storage location named `temp` here. The program either should execute all three statements or it should execute none of them. That alternative is accomplished by combining the three statements into the statement block.

Note that the variable `temp` is declared inside the block. That makes it *local* to the block; *i.e.*, it only exists during the execution of the block. If the condition is false (*i.e.*, `x ≤ y`), then `temp` will never exist. This illustrates the recommended practice of localizing objects so that they are created only when needed.

Note that a C++ program itself is a statement block preceded by `int main()`.

Recall (Section 1.5 on page 5) that the *scope* of a variable is that part of a program where the variable can be used. It extends from the point where the variable is declared to the end of the block which that declaration controls. So a block can be used to limit the scope of a variable, thereby allowing the same name to be used for different variables in different parts of a program.

### EXAMPLE 3.7 Using Blocks to Limit Scope

This program uses the same name `n` for three different variables:

```
int main()
{ int n=44;
  cout << "n = " << n << endl;
  { int n;                          // scope extends over 4 lines
    cout << "Enter an integer: ";
    cin >> n;
    cout << "n = " << n << endl;
  }
  { cout << "n = " << n << endl;    // the n that was declared first
  }
  { int n;                          // scope extends over 2 lines
    cout << "n = " << n << endl;
  }
  cout << "n = " << n << endl;      // the n that was declared first
}
```

```
n = 44
Enter an integer: 77
n = 77
n = 44
n = 4251897
n = 44
```

This program has three internal blocks. The first block declares a new `n` which exists only within that block and overrides the previous variable `n`. So the original `n` retains its value of 44 when this `n` is given the input value 77. The second block does not redeclare `n`, so the scope of the original `n` includes this block. Thus the third output is the original value 44. The third block is like the first block: it declares a new `n` which overrides the original `n`. But this third block does not initialize its local `n`, so the fourth output is a garbage value (4251897). Finally, since the scope of each redeclared `n` extends only to the block where it is declared, the last line of the program is in the scope of the original `n`, so it prints 44.

## 3.6 COMPOUND CONDITIONS

Conditions such as `n % d` and `x >= y` can be combined to form compound conditions. This is done using the *logical operators* **`&&`** (and), `||` (or), and **`!`** (not). They are defined by

| | |
|---|---|
| **`p && q`** | evaluates to true if and only if both `p` and `q` evaluate to true |
| **`p || q`** | evaluates to false if and only if both `p` and `q` evaluate to false |
| **`!p`** | evaluates to true if and only if `p` evaluates to false |

For example, `(n % d || x >= y)` will be false if and only if `n % d` is zero and `x` is less than `y`.

The definitions of the three logical operators are usually given by the *truth tables* below.

| p | q | p && q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

| p | q | p \|\| q |
|---|---|---|
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

| p | !p |
|---|---|
| T | F |
| F | T |

These show, for example, that if `p` is true and q is false, then the expression `p && q` will be false and the expression `p || q` will be true.

The next example solves the same problem that Example 3.5 on page 39 solved, except that it uses compound conditions.

### EXAMPLE 3.8 Using Compound Conditions

This program has the same effect as the one in Example 3.5 on page 39. This version uses compound conditions to find the minimum of three integers:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 <= n2 && n1 <= n3) cout << "Their minimum is " << n1 <<endl;
  if (n2 <= n1 && n2 <= n3) cout << "Their minimum is " << n2 <<endl;
  if (n3 <= n1 && n3 <= n2) cout << "Their minimum is " << n3 <<endl;
}
```

```
Enter two integers: 77 33 55
Their minimum is 33
```

Note that Example 3.8 is no improvement over Example 3.5. Its purpose was simply to illustrate the use of compound conditions.

Here is another example using a compound condition:

### EXAMPLE 3.9 User-Friendly Input

This program allows the user to input either a “Y” or a “y” for “yes”:

```
int main()
{ char ans;
  cout << "Are you enrolled (y/n): ";
  cin >> ans;
  if (ans == 'Y' || ans == 'y') cout << "You are enrolled.\n";
  else cout << "You are not enrolled.\n";
}
```

```
Are you enrolled (y|n): N
You are not enrolled.
```

It prompts the user for an answer, suggesting a response of either `y` or `n`. But then it accepts any character and concludes that the user meant "no" unless either a `Y` or a `y` is input.

## 3.7 SHORT-CIRCUITING

Compound conditions that use **`&&`** and `||` will not even evaluate the second operand of the condition unless necessary. This is called *short-circuiting*. As the truth tables show, the condition `p && q` will be false if `p` is false. In that case there is no need to evaluate `q`. Similarly if `p` is true then there is no need to evaluate `q` to determine that `p || q` is true. In both cases the value of the condition is known as soon as the first operand is evaluated.

### EXAMPLE 3.10 Short-Circuiting

This program tests integer divisibility:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (d != 0 && n%d == 0) cout << d << " divides " << n << endl;
  else cout << d << " does not divide " << n << endl;
}
```

In this run, `d` is positive and `n%d` is zero, so the compound condition is true:

```
Enter two positive integers: 300 6
6 divides 300
```

In this run, `d` is positive but `n%d` is not zero, so the compound condition is false:

```
Enter two positive integers: 300 7
7 does not divide 300
```

In this run,`d` is zero, so the compound condition is immediately determined to be false without evaluating the second expression "`n%d == 0`":

```
Enter two positive integers: 300 0
0 does not divide 300
```

This short-circuiting prevents the program from crashing because when `d` is zero the expression `n%d` cannot be evaluated.

## 3.8 BOOLEAN EXPRESSIONS

A *boolean expression* is a condition that is either true or false. In the previous example the expressions `d > 0`, `n%d == 0`, and `(d > 0 && n%d == 0)` are boolean expressions. As we have seen, boolean expressions evaluate to integer values. The value 0 means "false" and every nonzero value means "true."

Since all nonzero integer values are interpreted as meaning "true," boolean expressions are often disguised. For example, the statement

```
if (n) cout << "n is not zero";
```

will print `n is not zero` precisely when `n` is not zero because that is when the boolean expression `(n)` is interpreted as "true". Here is a more realistic example:

```
if (n%d) cout << "n is not a multiple of d";
```

The output statement will execute precisely when `n%d` is not zero, and that happens precisely when `d` does not divide `n` evenly, because n%d is the remainder from the integer division.

The fact that boolean expressions have integer values can lead to some surprising anomalies in C++.

### EXAMPLE 3.11 Another Logical Error

This program is erroneous:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 >= n2 >= n3) cout << "max = x";          // LOGICAL ERROR!
}
```

```
Enter an integer: 0 0 1
max = 0
```

The source of this error is the fact that boolean expressions have numeric values. Since the expression `(n1 >= n2 >= n3)` is evaluated from left to right, the first part `n1 >= n2` evaluates to "true" since 0 >= 0. But "true" is stored as the numeric value 1. That value is then compared to the value of `n3` which is also 1, so the complete expression evaluates to "true" even though it is really false! (0 is not the maximum of 0, 0, and 1.)

The problem here is that the erroneous line is syntactically correct, so the compiler cannot catch the error. Nor can the operating system. This is another logical error, comparable to that in the program in Example 3.4 on page 38.

The moral from Example 3.11 is to remember that boolean expressions have numeric values, so compound conditions can be tricky.

## 3.9 NESTED SELECTION STATEMENTS

Like compound statements, selection statements can be used wherever any other statement can be used. So a selection statement can be used within another selection statement. This is called *nesting* statements.

### EXAMPLE 3.12 Nesting Selection Statements

This program has the same effect as the one in Example 3.10 on page 42:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (d != 0)
    if (n%d == 0) cout << d << " divides " << n << endl;
    else cout << d << " does not divide " << n << endl;
  else cout << d << " does not divide " << n << endl;
}
```

The second `if..else` statement is nested within the `if` clause of the first `if..else` statement. So the second `if..else` statement will execute only when `d` is not zero.

Note that the `" does not divide "` statement has to be used twice here. The first one, nested within the `if` clause of the first `if..else` statement, executes when `d` is not zero and `n%d` is zero. The second one executes when `d` is zero.

When **`if..else`** statements are nested, the compiler uses the following rule to parse the compound statement:

*Match each* **`else`** *with the last unmatched* **`if`**.

Using this rule, the compiler can easily decipher code as inscrutable as this:

```
if (a > 0) if (b > 0) ++a; else if (c > 0)        // BAD CODING STYLE
if (a < 4) ++b; else if (b < 4) ++c; else --a;    // BAD CODING STYLE
else if (c < 4) --b; else --c; else a = 0;        // BAD CODING STYLE
```

To make this readable for humans it should be written either like this:

```
if (a > 0)
  if (b > 0) ++a;
  else
    if (c > 0)
      if (a < 4) ++b;
      else
        if (b < 4) ++c;
        else --a;
    else
      if (c < 4) --b;
      else --c;
else a = 0;
```

or like this:

```
if (a > 0)
  if (b > 0) ++a;
  else if (c > 0)
    if (a < 4) ++b;
    else if (b < 4) ++c;
    else --a;
  else if (c < 4) --b;
  else --c;
else a = 0;
```

This second rendering aligns the `else if` pairs when they form parallel alternatives. (See Section 3.10 on page 46.)

### EXAMPLE 3.13 Using Nested Selection Statements

This program has the same effect as those in Example 3.5 on page 39 and Example 3.8 on page 41. This version uses nested `if..else` statements to find the minimum of three integers:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 < n2)
    if (n1 < n3) cout << "Their minimum is " << n1 << endl;
    else cout << "Their minimum is " << n3 << endl;
  else  // n1 >= n2
    if (n2 < n3) cout << "Their minimum is " << n2 << endl;
```

```
    else cout << "Their minimum is " << n3 << endl;
}
Enter three integers: 77 33 55
Their minimum is 33
```

In this run, the first condition `(n1 < n2)` is false, and the third condition `(n2 < n3)` is true, so it reports that `n2` is the minimum.

This program is more efficient than the one in Example 3.8 on page 41 because on any run it will evaluate only two simple conditions instead of three compound conditions. Nevertheless, it should be considered inferior because its logic is more complicated. In the trade-off between efficiency and simplicity, it is usually best to choose simplicity.

### EXAMPLE 3.14 A Guessing Game

This program finds a number that the user selects from 1 to 8:

```
int main()
{ cout << "Pick a number from 1 to 8." << endl;
  char answer;
  cout << "Is it less than 5? (y|n): ";  cin >> answer;
  if (answer == 'y')  // 1 <= n <= 4
  { cout << "Is it less than 3? (y|n): ";  cin >> answer;
    if (answer == 'y')  // 1 <= n <= 2
    { cout << "Is it less than 2? (y|n): ";  cin >> answer;
      if (answer == 'y') cout << "Your number is 1." << endl;
      else cout << "Your number is 2." << endl;
    }
    else  // 3 <= n <= 4
    { cout << "Is it less than 4? (y|n): ";  cin >> answer;
      if (answer == 'y') cout << "Your number is 3." << endl;
      else cout << "Your number is 4." << endl;
    }
  }
  else  // 5 <= n <= 8
  { cout << "Is it less than 7? (y|n): ";  cin >> answer;
    if (answer == 'y')  // 5 <= n <= 6
    { cout << "Is it less than 6? (y|n): ";  cin >> answer;
      if (answer == 'y') cout << "Your number is 5." << endl;
      else cout << "Your number is 6." << endl;
    }
    else  // 7 <= n <= 8
    { cout << "Is it less than 8? (y|n): ";  cin >> answer;
      if (answer == 'y') cout << "Your number is 7." << endl;
      else cout << "Your number is 8." << endl;
    }
  }
}
```

By repeatedly subdividing the problem, it can discover any one of the 8 numbers by asking only three questions. In this run, the user's number is 6.

```
Pick a number from 1 to 8.
Is it less than 5? (y|n): n
Is it less than 7? (y|n): y
Is it less than 6? (y|n): n
Your number is 6.
```

The algorithm used in Example 3.14 is called the *binary search*. It can be implemented more simply. (See Example 6.14 on page 135.)

## 3.10 THE `else if` CONSTRUCT

Nested **`if..else`** statements are often used to test a sequence of parallel alternatives, where only the **`else`** clauses contain further nesting. In that case, the resulting compound statement is usually formatted by lining up the **`else if`** phrases to emphasize the parallel nature of the logic.

### EXAMPLE 3.15 Using the `else if` Construct for Parallel Alternatives

This program requests the user's language and then prints a greeting in that language:

```
int main()
{ char language;
  cout << "Engl., Fren., Ger., Ital., or Rus.? (e|f|g|i|r): ";
  cin >> language;
  if (language == 'e') cout << "Welcome to ProjectEuclid.";
  else if (language == 'f') cout << "Bon jour, ProjectEuclid.";
  else if (language == 'g') cout << "Guten tag, ProjectEuclid.";
  else if (language == 'i') cout << "Bon giorno, ProjectEuclid.";
  else if (language == 'r') cout << "Dobre utre, ProjectEuclid.";
  else cout << "Sorry; we don't speak your language.";
}
```

```
Engl., Fren., Ger., Ital., or Rus.? (e|f|g|i|r): i
Bon giorno, ProjectEuclid.
```

This program uses nested `if..else` statements to select from the five given alternatives.

As ordinary nested `if..else` statements, the code could also be formatted as

```
if (language == 'e') cout << "Welcome to ProjectEuclid.";
else
  if (language == 'f') cout << "Bon jour, ProjectEuclid.";
  else
    if (language == 'g') cout << "Guten tag, ProjectEuclid.";
    else
      if (language == 'i') cout << "Bon giorno, ProjectEuclid.";
       else
         if (language == 'r') cout << "Dobre utre, ProjectEuclid.";
         else cout << "Sorry; we don't speak your language.";
```

But the given format is preferred because it displays the parallel nature of the logic more clearly. It also requires less indenting.

### EXAMPLE 3.16 Using the `else if` Construct to Select a Range of Scores

This program converts a test score into its equivalent letter grade:

```
int main()
{ int score;
  cout << "Enter your test score: ";  cin >> score;
  if (score > 100) cout << "Error: that score is out of range.";
  else if (score >= 90) cout << "Your grade is an A." << endl;
  else if (score >= 80) cout << "Your grade is a B."  << endl;
  else if (score >= 70) cout << "Your grade is a C."  << endl;
  else if (score >= 60) cout << "Your grade is a D."  << endl;
  else if (score >=  0) cout << "Your grade is an F." << endl;
  else cout << "Error: that score is out of range.";
}
```

```
Enter your test score: 83
Your grade is a B.
```

The variable `score` is tested through a cascade of selection statements, continuing until either one of the conditions is found to be true, or the last `else` is reached.

## 3.11 THE `switch` STATEMENT

The `switch` statement can be used instead of the `else if` construct to implement a sequence of parallel alternatives. Its syntax is

```
switch (expression)
{ case constant1: statementList1;
  case constant2: statementList2;
  case constant3: statementList3;
    :
    :
  case constantN: statementListN;
  default: statementList0;
}
```

This evaluates the *expression* and then looks for its value among the **case** constants. If the value is found among the constants listed, then the statements in the corresponding *statementList* are executed. Otherwise if there is a **default** (which is optional), then the program branches to its *statementList*. The *expression* must evaluate to an integral type (see Section 2.1 on page 16) and the *constant*s must be integral constants.

### EXAMPLE 3.17 Using a `switch` Statement to Select a Range of Scores

This program has the same effect as the one in Example 3.16:

```
int main()
{ int score;
  cout << "Enter your test score: ";  cin >> score;
  switch (score/10)
  { case 10:
    case  9: cout << "Your grade is an A." << endl;  break;
    case  8: cout << "Your grade is a B."  << endl;  break;
    case  7: cout << "Your grade is a C."  << endl;  break;
```

```
    case  6: cout << "Your grade is a D."  << endl;  break;
    case  5:
    case  4:
    case  3:
    case  2:
    case  1:
    case  0: cout << "Your grade is an F." << endl;  break;
    default: cout << "Error: score is out of range.\n";
  }
  cout << "Goodbye." << endl;
}
```

```
Enter your test score: 83
Your grade is a B.
Goodbye.
```

First the program divides the `score` by 10 to reduce the range of values to 0–10. So in the test run, the score 83 reduces to the value 8, the program execution branches to `case 8`, and prints the output shown. Then the break statement causes the program execution to branch to the first statement after the `switch` block. That statement prints "`Goodbye.`".

Note that scores in the ranges 101 to 109 and -9 to -1 produce incorrect results. (See Problem 3.14.)

It is normal to put a `break` statement at the end of each case clause in a `switch` statement. Without it, the program execution will not branch directly out of the `switch` block after it finishes executing its `case` statement sequence. Instead, it will continue within the `switch` block, executing the statements in the next `case` sequence. This (usually) unintended consequence is called a *fall through*.

### EXAMPLE 3.18 An Erroneous Fall-through in a `switch` Statement

This program was intended to have the same effect as the one in Example 3.17. But with no break statements, the program execution falls through all the case statements it encounters:

```
int main()
{ int score;
  cout << "Enter your test score: ";  cin >> score;
  switch (score/10)
  { case 10:
    case  9: cout << "Your grade is an A." << endl;  // LOGICAL ERROR
    case  8: cout << "Your grade is a B."  << endl;  // LOGICAL ERROR
    case  7: cout << "Your grade is a C."  << endl;  // LOGICAL ERROR
    case  6: cout << "Your grade is a D."  << endl;  // LOGICAL ERROR
    case  5:
    case  4:
    case  3:
    case  2:
    case  1:
    case  0: cout << "Your grade is an F." << endl;  // LOGICAL ERROR
    default: cout << "Error: score is out of range.\n";
  }
  cout << "Goodbye." << endl;
}
```

```
Enter your test score: 83
Your grade is a B.
Your grade is a C.
```

```
Your grade is a D.
Your grade is an F.
Error: score is out of range.
Goodbye.
```

After branching to `case 8`, and printing "`Your grade is a B.`", the program execution goes right on to `case 7` and prints "`Your grade is a C.`" Since the `break` statements have been removed, it keeps falling through, all the way down to the `default` clause, executing each of the `cout` statements along the way.

## 3.12 THE CONDITIONAL EXPRESSION OPERATOR

C++ provides a special operator that often can be used in place of the **`if...else`** statement. It is called the *conditional expression operator*. It uses the **`?`** and the **`:`** symbols in this syntax:

*`condition`* **`?`** *`expression1`* **`:`** *`expression2`*

It is a *ternary operator*; *i.e.*, it combines three operands to produce a value. That resulting value is either the value of *`expression1`* or the value of *`expression2`*, depending upon the boolean value of the *`condition`*. For example, the assignment

```
min = ( x<y ? x : y );
```

would assign the minimum of `x` and `y` to `min`, because if the condition **`x<y`** is true, the expression **`( x<y ? x : y )`** evaluates to `x`; otherwise it evaluates to `y`.

Conditional expression statements should be used sparingly: only when the condition and both expressions are very simple.

### EXAMPLE 3.19 Finding the Minimum Again

This program has the same effect as the program in Example 3.3 on page 38:

```
int main()
{ int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  cout << ( m<n ? m : n ) << " is the minimum." << endl;
}
```

The conditional expression **`( m<n ? m : n )`** evaluates to `m` if `m<n`, and to `n` otherwise.

### Review Questions

**3.1** Write a single C++ statement that prints `"Too many"` if the variable `count` exceeds 100.

**3.2** What is wrong with the following code:

***a.*** `cin << count;`

***b.***
```
if x < y min = x
  else min = y;
```

**3.3** What is wrong with this code:

```
cout << "Enter n: ";
cin >> n;
if (n < 0)
  cout << "That is negative. Try again." << endl;
  cin >> n;
```

```
else
  cout << "o.k. n = " << n << endl;
```

**3.4** What is the difference between a reserved word and a standard identifier?

**3.5** State whether each of the following is true or false. If false, tell why.

***a.*** `!(p || q)` is the same as `!p || !q`

***b.*** `!!!p` is the same as `!p`

***c.*** `p && q || r` is the same as `p && (q || r)`

**3.6** Construct a truth table for each of the following boolean expressions, showing its truth value (0 or 1) for all 4 combinations of truth values of its operands p and q.

***a.*** `!p || q`

***b.*** `p&&q || !p&&!q`

***c.*** `(p||q) && !(p&&q)`

**3.7** Use truth tables to determine whether the two boolean expressions in each of the following are equivalent.

***a.*** `!(p && q)` and `!p && !q`

***b.*** `!!p` and `p`

***c.*** `!p || q` and `p || !q`

***d.*** `p && (q && r)` and `(p && q) && r`

***e.*** `p || (q && r)` and `(p || q) && r`

**3.8** What is short-circuiting and how is it helpful?

**3.9** What is wrong with this code:

```
if (x = 0) cout << x << " = 0\n";
else cout << x << " != 0\n";
```

**3.10** What is wrong with this code:

```
if (x < y < z) cout << x << " < " << y << " < " << z << endl;
```

**3.11** Construct a logical expression to represent each of the following conditions:

***a.*** `score` is greater than or equal to 80 but less than 90;

***b.*** `answer` is either `'N'` or `'n'`;

***c.*** `n` is even but not 8;

***d.*** `ch` is a capital letter.

**3.12** Construct a logical expression to represent each of the following conditions:

***a.*** `n` is between 0 and 7 but not equal to 3;

***b.*** `n` is between 0 and 7 but not even;

***c.*** `n` is divisible by 3 but not by 30;

***d.*** `ch` is a lowercase or uppercase letter.

**3.13** What is wrong with this code:

```
if (x == 0)
  if (y == 0) cout << "x and y are both zero." << endl;
else cout << "x is not zero." << endl;
```

**3.14** What is the difference between the following two statements:

```
if (n > 2) { if (n < 6) cout << "OK"; } else cout << "NG";
if (n > 2) { if (n < 6) cout << "OK"; else cout << "NG"; }
```

**3.15** What is a “fall-through?

**3.16** How is the following expression evaluated?

```
(x < y ? -1 : (x == y ? 0 : 1) );
```

**3.17** Write a single C++ statement that uses the conditional expression operator to assign the absolute value of `x` to `absx`.

**3.18** Write a single C++ statement that prints "too many" if the variable `count` exceeds 100, using
***a.*** an `if` statement;
***b.*** the conditional expression operator.

## Problems

**3.1** Modify the program in Example 3.1 on page 36 so that it prints a response only if `n` is divisible by `d`.
**3.2** Modify the program in Example 3.5 on page 39 so that it prints the minimum of four input integers.
**3.3** Modify the program in Example 3.5 on page 39 so that it prints the median of three input integers.
**3.4** Modify the program in Example 3.6 on page 39 so that it has the same effect without using a statement block.
**3.5** Predict the output from the program in Example 3.7 on page 40 after removing the declaration on the fifth line of the program. Then run that program to check your prediction.
**3.6** Write and run a program that reads the user's age and then prints "You are a child." if the age < 18, "You are an adult." if 18 ≤ age < 65, and "You are a senior citizen." if age ≥ 65.
**3.7** Write and run a program that reads two integers and then uses the conditional expression operator to print either "multiple" or "not" according to whether one of the integers is a multiple of the other.
**3.8** Write and run a program that simulates a simple calculator. It reads two integers and a character. If the character is a `+`, the sum is printed; if it is a `-`, the difference is printed; if it is a `*`, the product is printed; if it is a `/`, the quotient is printed; and if it is a `%`, the remainder is printed. Use a `switch` statement.
**3.9** Write and run a program that plays the game of "Rock, paper, scissors." In this game, two players simultaneously say (or display a hand symbol representing) either "rock," "paper," or "scissors." The winner is the one whose choice dominates the other. The rules are: paper dominates (wraps) rock, rock dominates (breaks) scissors, and scissors dominate (cut) paper. Use enumerated types for the choices and for the results.
**3.10** Modify the solution to Problem 3.9 by using a `switch` statement.
**3.11** Modify the solution to Problem 3.10 by using conditional expressions where appropriate.
**3.12** Write and test a program that solves quadratic equations. A *quadratic equation* is an equation of the form $ax^2 + bx + c = 0$, where $a$, $b$, and $c$ are given coefficients and $x$ is the unknown. The coefficients are real number inputs, so they should be declared of type `float` or `double`. Since quadratic equations typically have two solutions, use `x1` and `x2` for the solutions to be output. These should be declared of type `double` to avoid inaccuracies from round-off error. (See Example 2.15 on page 28.)
**3.13** Write and run a program that reads a six-digit integer and prints the sum of its six digits. Use the *quotient operator* `/` and the *remainder operator* `%` to extract the digits from the integer. For example, if `n` is the integer 876,543, then `n/1000%10` is its thousands digit 6.
**3.14** Correct Example 3.17 on page 47 so that it produces the correct response for all inputs.

## Answers to Review Questions

**3.1**
```
if (count > 100) cout << "Too many";
```

**3.2** ***a.*** Either `cout` should be used in place of `cin`, or the extraction operator `>>` should be used in place of the insertion operator `<<`.

***b.*** Parentheses are required around the condition `x < y`, and a semicolon is required at the end of the `if` clause before the `else`.

**3.3** There is more than one statement between the `if` clause and the `else` clause. They need to be made into a compound statement by enclosing them in braces { }.

**3.4** A *reserved word* is a keyword in a programming language that serves to mark the structure of a statement. For example, the keywords `if` and `else` are reserved words. A *standard identifier* is a keyword that defines a type. Among the 63 keywords in C++, `if`, `else`, and `while` are some of the reserved words, and `char`, `int`, and `float` are some of the standard identifiers.

**3.5** ***a.*** `!(p || q)` is not the same as `!p || !q`; for example, if `p` is true and `q` is false, the first expression will be false but the second expression will be true. The correct equivalent to the expression `!(p || q)` is the expression `!p && !q`.

***b.*** `!!!p` is the same as `!p`.

***c.*** `p && q || r` is not the same as `p && (q || r)`; for example, if `p` is false and `r` is true, the first expression will be true, but the second expression will be false: `p && q || r` is the same as `(p && q) || r`.

**3.6** Truth tables for boolean expressions:

| p | q | !p \|\| q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

| p | q | p&&q \|\| !p&&!q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | T |

| p | q | (p\|\|q) && !(p&&q) |
|---|---|---|
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

**3.7** ***a.*** These two boolean expressions are not equivalent:

| p | q | !(p&&q) |
|---|---|---|
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | T |

| p | q | !p && !q |
|---|---|---|
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

***b.*** These two boolean expressions are equivalent:

| p | !pp |
|---|---|
| T | T |
| F | F |

| p | p |
|---|---|
| T | T |
| F | F |

***c.*** These two boolean expressions are not equivalent:

| p | q | !p \|\| q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

| p | q | p \|\| !q |
|---|---|---|
| T | T | T |
| T | F | T |
| F | T | F |
| F | F | T |

***d.*** These two boolean expressions are equivalent:

| p | q | r | p && (q&&r) |
|---|---|---|---|
| T | T | T | T |
| T | T | F | F |
| T | F | T | F |
| T | F | F | F |
| F | T | T | F |
| F | T | F | F |
| F | F | T | F |
| F | F | F | F |

| p | q | r | (p&&q) && r |
|---|---|---|---|
| T | T | T | T |
| T | T | F | F |
| T | F | T | F |
| T | F | F | F |
| F | T | T | F |
| F | T | F | F |
| F | F | T | F |
| F | F | F | F |

***e.*** These two boolean expressions are not equivalent:

| p | q | r | p \|\| (q&&r) |
|---|---|---|---|
| T | T | T | T |
| T | T | F | T |
| T | F | T | T |
| T | F | F | T |
| F | T | T | T |
| F | T | F | F |
| F | F | T | F |
| F | F | F | F |

| p | q | r | (p\|\|q) && r |
|---|---|---|---|
| T | T | T | T |
| T | T | F | F |
| T | F | T | T |
| T | F | F | F |
| F | T | T | T |
| F | T | F | F |
| F | F | T | F |
| F | F | F | F |

**3.8** The term "short-circuiting" is used to describe the way C++ evaluates compound logical expressions like `(x > 2 || y > 5)` and `(x > 2 && y > 5)`. If `x` is greater than 2 in the first expression, then `y` will not be evaluated. If `x` is less than or equal to 2 in the second expression, then `y` will not be evaluated. In these cases only the first part of the compound expression is evaluated because that value alone determines the truth value of the compound expression.

**3.9** The programmer probably intended to test the condition `(x == 0)`. But by using assignment operator "=" instead of the equality operator "==" the result will be radically different from what was intended. For example, if `x` has the value 22 prior to the `if` statement, then the `if` statement will change the value of `x` to 0. Moreover, the assignment expression `(x = 0)` will be evaluated to 0 which means "false," so the `else` part of the selection statement will execute, reporting that `x` is not zero!

**3.10** The programmer probably intended to test the condition `(x < y && y < z)`. The code as written will compile and run, but not as intended. For example, if the prior values of `x`, `y`, and `z` are 44, 66, and 22, respectively, then the algebraic condition "$x < y < z$" is false. But as written, the code will be evaluated from left to right, as `(x < y) < z`. First the condition `x < y` will be evaluated as "true." But this has the numeric value 1, so the expression `(x < y)` is evaluated to 1. Then the combined expression `(x < y) < z` is evaluated as (1) < 66 which is also true. So the output statement will execute, erroneously reporting that 44 < 66 < 22.

**3.11** ***a.*** `(score >= 80 && score < 90)`

***b.*** `(answer == 'N' || answer == 'n')`

***c.*** `(n%2 == 0 && n != 8)`

***d.*** `(ch >= 'A' && ch <= 'Z')`

**3.12** ***a.*** `(n > 0 && n < 7 && n != 3)`

***b.*** `(n > 0 && n < 7 && n%2 != 0)`

***c.*** `((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'))`

**3.13** The programmer clearly intended for the second output `"x is not zero."` to be printed if the first condition `(x == 0)` is false, regardless of the second condition `(y == 0)`. That is, the `else` was intended to be matched with the first `if`. But the "`else` matching" rule causes it to be matched with the second condition, which means that the output `"x is not zero."` will be printed only when `x` is zero and `y` is not zero. The "`else` matching" rule can be overridden with braces:

```
if (x == 0)
{ if (y == 0) cout << "x and y are both zero." << endl;
}
else cout << "x is not zero." << endl;
```

Now the `else` will be matched with the first `if`, the way the programmer had intended it to be.

**3.14** In the first statement, the `else` is matched with the first `if`. In the second statement, the `else` is matched with the second `if`. If $n \le 2$, the first statement will print `NG` while the second statement will do nothing. If $2 < n < 6$, both statements will print `OK`. If $n \ge 6$, the first statement will do nothing while the second statement will print NG. Note that this code is difficult to read because it does not follow standard indentation conventions. The first statement should be written

```
if (n > 2)
{ if (n < 6) cout << "OK";
}
else cout << "NG";
```

The braces are needed here to override the "`else` matching" rule. This `else` is intended to match the first `if`. The second statement should be written

```
if (n > 2)
  if (n < 6) cout << "OK";
  else cout << "NG";
```

Here the braces are not needed because the `else` is intended to be matched with the second `if`.

**3.15** A "fall through" in a `switch` statement is a case that does not include a `break` statement, thereby causing control to continue right on to the next case statement.

**3.16** This expression evaluates to –1 if `x < y`, it evaluates to 0 if `x == y`, and it evaluates to 1 if `x > y`.

**3.17**
```
absx = ( x>0 ? x : -x );
```

**3.18** ***a.***
```
if (count > 100) cout << "too many";
```
***b.***
```
cout << (count > 100 ? "too many" : "" );
```

## Solutions to Problems

**3.1** This version of Example 3.1 on page 36 prints a response only when `n` is divisible by `d`:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (n%d == 0) cout << n << " is divisible by " << d << endl;
}
```

```
Enter two positive integers: 56 7
56 is divisible by 7
```

**3.2** This version of Example 3.5 on page 39 prints the minimum of four input integers:

```
int main()
{ int n1, n2, n3, n4;
  cout << "Enter four integers: ";
  cin >> n1 >> n2 >> n3 >> n4;
  int min=n1;               // now min <= n1
  if (n2 < min) min = n2;  // now min <= n1, n2
  if (n3 < min) min = n3;  // now min <= n1, n2, n3
  if (n4 < min) min = n4;  // now min <= n1, n2, n3, n4
  cout << "Their minimum is " << min << endl;
}
```

```
Enter four integers: 44 88 22 66
Their minimum is 22
```

**3.3** This program finds the median of three input integers:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  cout << "Their median is ";
  if (n1 < n2)
    if (n2 < n3) cout << n2;       // n1 <  n2 <  n3
```

```
    else if (n1 < n3) cout << n3;  // n1 <  n3 <= n2
    else cout << n1;               // n3 <= n1 <  n2
  else if (n1 < n3) cout << n1;    // n2 <= n1 <  n3
  else if (n2 < n3) cout << n2;    // n2 <  n3 <= n1
  else cout << n3;                 // n3 <= n2 <= n1
}
```

```
Enter three integers: 44 88 22
Their median is 44
```

**3.4** This program has the same effect as the one in Example 3.6 on page 39:

```
int main()
{ int x, y;
  cout << "Enter two integers: ";
  cin >> x >> y;
  if (x > y) cout << y << " <= " << x << endl;
  else cout << x << " <= " << y << endl;
}
```

```
Enter two integers: 66 44
44 <= 6
```

**3.5** Modification of the program in Example 3.7 on page 40:

```
int main()
{ int n=44;
  cout << "n = " << n << endl;
  { cout << "Enter an integer: ";
    cin >> n;
    cout << "n = " << n << endl;
  }
  { cout << "n = " << n << endl;
  }
  { int n;
    cout << "n = " << n << endl;
  }
  cout << "n = " << n << endl;
}
```

```
n = 44
Enter an integer: 77
n = 77
n = 77
n = 4251897
n = 77
```

**3.6** Here we used the `else if` construct because the three outcomes depend upon `age` being in one of three disjoint intervals:

```
int main()
{ int age;
  cout << "Enter your age: ";
  cin >> age;
  if (age < 18) cout << "You are a child.\n";
  else if (age < 65) cout << "You are an adult.\n";
  else cout << "you are a senior citizen.\n";
}
```

```
Enter your age: 44
You are an adult.
```

If control reaches the second condition `(age < 65)`, then the first condition must be false so in fact 18 ≤ age < 65. Similarly, if control reaches the second `else`, then both conditions must be false so in fact age ≥ 65.

**3.7** An integer `m` is a multiple of an integer `n` if the remainder from the integer division of `m` by `n` is 0. So the compound condition `m % n == 0 || n % m == 0` tests whether either is a multiple of the other:

```
int main()
{ int m, n;
  cin >> m >> n;
  cout << (m % n == 0 || n % m == 0 ? "multiple" : "not") << endl;
}
```

```
30 4
not
```

```
30 5
multiple
```

The value of the conditional expression will be either `"multiple"` or `"not"`, according to whether the compound condition is true. So sending the complete conditional expression to the output stream produces the desired result.

**3.8** The character representing the operation should be the control variable for the `switch` statement:

```
int main()
{ int x, y;
  char op;
  cout << "Enter two integers: ";
  cin >> x >> y;
  cout << "Enter an operator: ";
  cin >> op;
  switch (op)
  { case '+': cout << x + y << endl;  break;
    case '-': cout << x - y << endl;  break;
    case '*': cout << x * y << endl;  break;
    case '/': cout << x / y << endl;  break;
    case '%': cout << x % y << endl;  break;
  }
}
```

```
Enter two integers: 30 13
Enter an operator: %
4
```

In each of the five cases, we simply print the value of the corresponding arithmetic operation and then break.

**3.9** First define the two `enum` types `Choice` and `Result`. Then declare variables `choice1`, `choice2`, and `result` of these types, and use an integer `n` to get the required input and assign it to them:

```
enum Choice {ROCK, PAPER, SCISSORS};
enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int n;
  Choice choice1, choice2;
  Winner winner;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> n;
  choice1 = Choice(n);
```

```
  cout << "Player #2: ";
  cin >> n;
  choice2 = Choice(n);
  if (choice1 == choice2) winner = TIE;
  else if (choice1 == ROCK)
    if (choice2 == PAPER) winner = PLAYER2;
    else winner = PLAYER1;
  else if (choice1 == PAPER)
    if (choice2 == SCISSORS) winner = PLAYER2;
    else winner = PLAYER1;
  else // (choice1 == SCISSORS)
    if (choice2 == ROCK) winner = PLAYER2;
    else winner = PLAYER1;
  if (winner == TIE) cout << "\tYou tied.\n";
  else if (winner == PLAYER1) cout << "\tPlayer #1 wins." <<endl;
  else cout << "\tPlayer #2 wins." << endl;
}
```

```
Choose rock (0), paper (1), or scissors (2):
Player #1: 1
Player #2: 1
    You tied.
```

```
Choose rock (0), paper (1), or scissors (2):
Player #1: 2
Player #2: 1
    Player #1 wins.
```

```
Choose rock (0), paper (1), or scissors (2):
Player #1: 2
Player #2: 0
        Player #2 wins.
```

Through a series of nested `if` statements, we are able to cover all the possibilities.

**3.10** Using a `switch` statement:

```
enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int choice1, choice2;
  Winner winner;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> choice1;
  cout << "Player #2: ";
  cin >> choice2;
  switch (choice2 - choice1)
  { case  0:
      winner = TIE;
      break;
    case -1:
    case  2:
      winner = PLAYER1;
      break;
    case -2:
    case  1:
      winner = PLAYER2;
  }
```

```
  if (winner == TIE) cout << "\tYou tied.\n";
  else if (winner == PLAYER1) cout << "\tPlayer #1 wins." << endl;
  else cout << "\tPlayer #2 wins." << endl;
}
```

**3.11** Using a `switch` statement and conditional expressions:

```
enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int choice1, choice2;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> choice1;
  cout << "Player #2: ";
  cin >> choice2;
  int n = (choice1 - choice2 + 3) % 3;
  Winner winner = ( n==0 ? TIE : (n==1?PLAYER1:PLAYER2) );
  if (winner == TIE) cout << "\tYou tied.\n";
  else if (winner == PLAYER1) cout << "\tPlayer #1 wins." << endl;
  else cout << "\tPlayer #2 wins." << endl;
}
```

**3.12** The solution(s) to the quadratic equation is given by the *quadratic formula*:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

But this will not apply if $a$ is zero, so that condition must be checked separately. The formula also fails to work (for real numbers) if the expression under the square root is negative. That expression $b^2 + 4ac$ is called the *discriminant* of the quadratic. We define that as the separate variable `d` and check its sign.

```
#include <iostream>
#include <cmath>  // defines the sqrt() function
int main()
{ // solves the equation a*x*x + b*x + c == 0:
  float a, b, c;
  cout << "Enter coefficients of quadratic equation: ";
  cin >> a >> b >> c;
  if (a == 0)
  { cout << "This is not a quadratic equation: a == 0\n";
    return 0;
  }
  cout << "The equation is: " << a << "x^2 + " << b
       << "x + " << c << " = 0\n";
  double d, x1, x2;
  d = b*b - 4*a*c;  // the discriminant
  if (d < 0)
  { cout << "This equation has no real solutions: d < 0\n";
    return 0;
  }
  x1 = (-b + sqrt(d))/(2*a);
  x2 = (-b - sqrt(d))/(2*a);
  cout << "The solutions are: " << x1 << ", " << x2 << endl;
}
```

```
Enter coefficients of quadratic equation: 2 1 -6
The equation is: 2x^2 + 1x + -6 = 0
The solutions are: 1.5, -2
```

```
Enter coefficients of quadratic equation: 1 4 5
The equation is: 1x^2 + 4x + 5 = 0
This equation has no real solutions: d < 0
```

```
Enter coefficients of quadratic equation: 0 4 5
This is not a quadratic equation: a == 0
```

Note how we use the `return` statement inside the selection statements to terminate the program if either `a` is zero or `d` is negative. The alternative would have been to use an `else` clause in each `if` statement.

**3.13** This program prints the sum of the digits of the given integer:

```
int main()
{ int n, sum;
  cout << "Enter a six-digit integer: ";
  cin >> n;
  sum = n%10 + n/10%10 + n/100%10 + n/1000%10 + n/10000%10
             + n/100000;
  cout << "The sum of the digits of " << n << " is " << sum <<endl;
}
```

```
Enter a six-digit integer: 876543
The sum of the digits of 876543 is 33
```

**3.14** A corrected version of Example 3.17 on page 47:

```
int main()
{ // reports the user's grade for a given test score:
  int score;
  cout << "Enter your test score: ";
  cin >> score;
  if (score > 100 || score < 0)
    cout << "Error: that score is out of range.\n";
  else
    switch (score/10)
    { case 10:
      case  9: cout << "Your grade is an A.\n";  break;
      case  8: cout << "Your grade is a B.\n";   break;
      case  7: cout << "Your grade is a C.\n";   break;
      case  6: cout << "Your grade is a D.\n";   break;
      default: cout << "Your grade is an F.\n";  break;
    }
  cout << "Goodbye." << endl;
}
```

```
Enter your test score: 103
Error: that score is out of range.
Goodbye.
```

```
Enter your test score: 93
Your grade is an A.
Goodbye.
```

```
Enter your test score: -3
Error: that score is out of range.
Goodbye.
```

# Chapter 4

# Iteration

Iteration is the repetition of a statement or block of statements in a program. C++ has three iteration statements: the `while` statement, the `do..while` statement, and the `for` statement. Iteration statements are also called *loops* because of their cyclic nature.

## 4.1 THE `while` STATEMENT

The syntax for the `while` statement is

**`while`** (*`condition`*) *`statement`*;

where *`condition`* is an integral expression and *`statement`* is any executable statement. If the value of the expression is zero (meaning "false") then the *`statement`* is ignored and program execution immediately jumps to the next statement that follows the **`while`** statement. If the value of the *`expression`* is nonzero (meaning "true") then the *`statement`* is executed repeatedly until the *`expression`* evaluates to zero. Note that the *`condition`* must be enclosed by parentheses.

### EXAMPLE 4.1 Using a `while` Loop to Compute a Sum of Consecutive Integers

This program computes the sum 1 + 2 + 3 + ··· + *n*, for an input integer *n*:

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (i <= n)
    sum += i++;
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

This program uses three local variables: `n`, `i`, and `sum`. Each time the `while` loop iterates, `i` is incremented and then added to `sum`. The loop stops when `i` = `n`, so `n` is the last value added to `sum`. The trace at right shows the values of `i` and `sum` on each iteration after the user input 8 for `n`. The output for this run is

| i | sum |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 3 |
| 3 | 6 |
| 4 | 10 |
| 5 | 15 |
| 6 | 21 |
| 7 | 28 |
| 8 | 36 |

```
Enter a positive integer: 8
The sum of the first 8 integers is 36
```

The program computed 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36.

On the second run the user inputs 100 for `n`, so the `while` loop iterated 100 times to compute the sum 1 + 2 + 3 + ··· + 98 + 99 + 100 = 5050:

```
Enter a positive integer: 100
The sum of the first 100 integers is 5050
```

Note that the statement inside the loop is indented. This convention makes the program's logic easier to follow, especially in large programs.

### EXAMPLE 4.2 Using a `while` Loop to Compute a Sum of Reciprocals

This program computes the sum of reciprocals $s = 1 + 1/2 + 1/3 + \cdots + 1/n$, where $n$ is the smallest integer for which $n \geq s$:

```
int main()
{ int bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  double sum=0.0;
  int i=0;
  while (sum < bound)
    sum += 1.0/++i;
  cout << "The sum of the first " << i
       << " reciprocals is " << sum << endl;
}
```

With input 3 for `n`, this run computes $1 + 1/2 + 1/3 + \cdots + 1/11 = 3.01988$:

```
Enter a positive integer: 3
The sum of the first 11 reciprocals is 3.01988
```

The trace of this run is shown at right. The `sum` does not exceed 3 until the 11th iteration.

| i | sum |
|---|---|
| 0 | 0.00000 |
| 1 | 1.00000 |
| 2 | 1.50000 |
| 3 | 1.83333 |
| 4 | 2.08333 |
| 5 | 2.28333 |
| 6 | 2.45000 |
| 7 | 2.59286 |
| 8 | 2.71786 |
| 9 | 2.82897 |
| 10 | 2.92897 |
| 11 | 3.01988 |

### EXAMPLE 4.3 Using a `while` Loop to Repeat a Computation

This program prints the square root of each number input by the user. It uses a `while` loop to allow any number of computations in a single run of the program:

```
int main()
{ double x;
  cout << "Enter a positive number: ";
  cin >> x;
  while (x > 0)
  { cout << "sqrt(" << x << ") = " << sqrt(x) << endl;
    cout << "Enter another positive number (or 0 to quit): ";
    cin >> x;
  }
}
```

```
Enter a positive number: 49
sqrt(49) = 7
Enter another positive number (or 0 to quit): 3.14159
sqrt(3.14159) = 1.77245
Enter another positive number (or 0 to quit): 100000
sqrt(100000) = 316.228
Enter another positive number (or 0 to quit): 0
```

The condition `(x > 0)` in Example 4.3 uses the variable `x` to control the loop. Its value is changed inside the loop by means of an input statement. A variable that is used this way is called a *loop control variable*.

## 4.2 TERMINATING A LOOP

We have already seen how the `break` statement is used to control the `switch` statement. (See Example 3.17 on page 47.) The `break` statement is also used to control loops.

### EXAMPLE 4.4 Using a `break` Statement to Terminate a Loop

This program has the same effect as the one in Example 4.1 on page 60:

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (true)
  { if (i > n) break;  // terminates the loop immediately
    sum += i++;
  }
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

```
Enter a positive integer: 100
The sum of the first 100 integers is 5050
```

This runs the same as in Example 4.1: as soon as the value of `i` reaches `n`, the loop terminates and the output statement at the end of the program executes.

Note that the control condition on the `while` loop itself is `true`, which means continue forever. This is the standard way to code a `while` loop when it is being controlled from within.

One advantage of using a `break` statement inside a loop is that it causes the loop to terminate immediately, without having to finish executing the remaining statements in the loop block.

### EXAMPLE 4.5 The Fibonacci Numbers

The *Fibonacci numbers* $F_0, F_1, F_2, F_3, \ldots$ are defined recursively by the equations

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

For example, letting $n = 2$ in the third equation yields

$$F_2 = F_{2-1} + F_{2-2} = F_1 + F_0 = 0 + 1 = 1$$

Similarly, with $n = 3$,

$$F_3 = F_{3-1} + F_{3-2} = F_2 + F_1 = 1 + 1 = 2$$

and with $n = 4$,

$$F_4 = F_{4-1} + F_{4-2} = F_3 + F_2 = 2 + 1 = 3$$

The first ten Fibonacci numbers are shown in the table at right.

| $n$ | $F_n$ |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 3 |
| 5 | 5 |
| 6 | 8 |
| 7 | 13 |
| 8 | 21 |
| 9 | 35 |

This program prints all the Fibonacci numbers up to an input limit:

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
```

```
  while (true)
  { long f2 = f0 + f1;
    if (f2 > bound) break;  // terminates the loop immediately
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

```
Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
```

This `while` loop contains a block of five statements. When the condition **`(f2 > bound)`** is evaluated to be true, the `break` statement executes, terminating the loop immediately, without executing the last three statements in that iteration.

Note the use of the *newline character* `\n` in the string `":\n0, 1"`. This prints the colon `:` at the end of the current line, and then prints `0, 1` at the beginning of the next line.

### EXAMPLE 4.6 Using the `exit(0)` Function

The `exit()` function provides another way to terminate a loop. When it executes, it terminates the program itself:

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
  while (true)
  { long f2 = f0 + f1;
    if (f2 > bound) exit(0);  // terminates the program immediately
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

```
Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
```

Since this program has no statements following its loop, terminating the loop is the same as terminating the program. So this program runs the same as the one in Example 4.5.

The program in Example 4.6 illustrates one way to break out of an infinite loop. The next example shows how to abort an infinite loop. But the preferred method is to use a break statement, as illustrated in Example 4.20 on page 71.

### EXAMPLE 4.7 Aborting Infinite Loop

Without some termination mechanism, the loop will run forever. To abort its execution after it starts, press <Ctrl>+C (*i.e.*, hold the Ctrl key down and press the C key on your keyboard):

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
  while (true)          // ERROR: INFINITE LOOP!     (Press <Ctrl>+C.)
  { long f2 = f0 + f1;
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

```
Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597
81, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 5
040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817,
63245986, 102334155, 165580141, 267914296, 433494437, 701408733, 11349
```

Since this program has no statements following its loop, terminating the loop is the same as terminating the program. So this program runs the same as the one in Example 4.5.

## 4.3 THE `do..while` STATEMENT

The syntax for the `do..while` statement is

**`do`** *`statement`* **`while (`***`condition`***`);`**

where *`condition`* is an integral expression and *`statement`* is any executable statement. It repeatedly executes the *`statement`* and then evaluates the *`condition`* until that condition evaluates to `false`.

The `do..while` statement works the same as the `while` statement except that its condition is evaluated at the end of the loop instead of at the beginning. This means that any control variables can be defined within the loop instead of before it. It also means that a `do...while` loop will always iterate at least once, regardless of the value of its control condition.

### EXAMPLE 4.8 Using a `do..while` Loop to Compute a Sum of Consecutive Integers

This program has the same effect as the one in Example 4.1 on page 60:

```
int main()
{ int n, i=0;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  do
    sum += i++;
  while (i <= n);
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

### EXAMPLE 4.9 The Factorial Numbers

The *factorial numbers* 0!, 1!, 2!, 3!, ⋯ are defined recursively by the equations

$$\begin{cases} 0! = 1 \\ n! = n(n-1) \end{cases}$$

For example, letting $n = 1$ in the second equation yields

$$1! = 1((1-1)!) = 1(0!) = 1(1) = 1$$

Similarly, with $n = 2$:

$$2! = 2((2-1)!) = 2(1!) = 2(1) = 2$$

and with $n = 3$:

$$3! = 3((3-1)!) = 3(2!) = 3(2) = 6$$

The first seven factorial numbers are shown in the table at right.

| $n$ | $n!$ |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 2 |
| 3 | 6 |
| 4 | 24 |
| 5 | 120 |
| 6 | 720 |

This program prints all the factorial numbers up to an input limit:

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Factorial numbers < " << bound << ":\n1, 1";
  long f=1, i=1;
  do
  { f *= ++i;
    cout << ", " << f;
  }
  while (f < bound);
}
```

```
Enter a positive integer: 1000000
Factorial numbers < 1000000:
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880
```

The `do..while` loop iterates until its control condition **`(f < bound)`** is false.

## 4.4 THE `for` STATEMENT

The syntax for the `for` statement is

**`for`** (*`initialization`*`;` *`condition`*`;` *`update`*`)` *`statement`*`;`

where *`initialization`*, *`condition`*, and *`update`* are optional expressions, and *`statement`* is any executable statement. The three-part (*`initialization`*`;` *`condition`*`;` *`update`*) controls the loop. The *`initialization`* expression is used to declare and/or initialize control variable(s) for the loop; it is evaluated first, before any iteration occurs. The *`condition`* expression is used to determine whether the loop should continue iterating; it is evaluated immediately after the initialization; if it is true, the statement is executed. The *`update`* expression is used to update the control variable(s); it is evaluated after the statement is executed. So the sequence of events that generate the iteration are:

1. evaluate the *`initialization`* expression;
2. if the value of the *`condition`* expression is false, terminate the loop;
3. execute the *`statement`*;
4. evaluate the *`update`* expression;
5. repeat steps 2–4.

### EXAMPLE 4.10 Using a `for` Loop to Compute a Sum of Consecutive Integers

This program has the same effect as the one in Example 4.1 on page 60:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  for (int i=1; i <= n; i++)
    sum += i;
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

Here, the initialization expression is **`int i=1`**, the condition expression is **`i <= n`**, and the update expression is **`i++`**. Note that these same expressions are used in the programs in Example 4.1 on page 60, Example 4.4 on page 62, and Example 4.8 on page 64.

In Standard C++, when a loop control variable is declared within a `for` loop, as `i` is in Example 4.10, its scope is limited to that `for` loop. That means that it cannot be used outside that `for` loop. It also means that the same name can be used for different variables outside that `for` loop.

### EXAMPLE 4.11 Reusing `for` Loop Control Variable Names

This program has the same effect as the one in Example 4.1 on page 60:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  for (int i=1; i < n/2; i++)  // the scope of this i is this loop
    sum += i;
  for (int i=n/2; i <= n; i++)  // the scope of this i is this loop
    sum += i;
  cout << "The sum of the first " << n << " integers is "
       << sum << endl;
}
```

The two `for` loops in this program do the same computations as the single `for` loop in the program in Example 4.10. They simply split the job in two, doing the first `n/2` accumulations in the first loop and the rest in the second. Each loop independently declares its own control variable `i`.

**Warning:** Most pre-Standard C++ compilers extend the scope of a `for` loop's control variable past the end of the loop.

### EXAMPLE 4.12 The Factorial Numbers Again

This program has the same effect as the one in Example 4.9 on page 65:

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
```

```
  cout << "Factorial numbers that are <= " << bound << ":\n1, 1";
  long f=1;
  for (int i=2; f <= bound; i++)
  { f *= i;
    cout << ", " << f;
  }
}
```

```
Enter a positive integer: 1000000
Factorial numbers < 1000000:
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880
```

This **for** loop program has the same effect as the **do..while** loop program because it executes the same instructions. After initializing `f` to 1, both programs initialize `i` to 2 and then repeat the following five instructions: print `f`, multiply `f` by `i`, increment `i`, check the condition `(f <= bound)`, and terminate the loop if the condition is false.

The **for** statement is quite flexible, as the following examples demonstrate.

### EXAMPLE 4.13 Using a Descending `for` Loop

This program prints the first ten positive integers in reverse order:

```
int main()
{ for (int i=10; i > 0; i--)
    cout << " " << i;
}
```

```
 10 9 8 7 6 5 4 3 2 1
```

### EXAMPLE 4.14 Using a `for` Loop with a Step Greater than One

This program determines whether an input number is prime:

```
int main()
{ long n;
  cout << "Enter a positive integer: ";
  cin >> n;
  if (n < 2) cout << n << " is not prime." << endl;
  else if (n < 4) cout << n << " is prime." << endl;
  else if (n%2 == 0) cout << n << " = 2*" << n/2 << endl;
  else
  { for (int d=3; d <= n/2; d += 2)
      if (n%d == 0)
      { cout << n << " = " << d << "*" << n/d << endl;
        exit(0);
      }
    cout << n << " is prime." << endl;
  };
}
```

```
Enter a positive integer: 101
101 is prime.
```

```
Enter a positive integer: 975313579
975313579 = 17*57371387
```

Note that this **for** loop uses an increment of 2 on its control variable `i`.

### EXAMPLE 4.15 Using a Sentinel to Control a `for` Loop

This program finds the maximum of a sequence of input numbers:

```
int main()
{ int n, max;
  cout << "Enter positive integers (0 to quit): ";
  cin >> n;
  for (max = n; n > 0; )
  { if (n > max) max = n;
    cin >> n;
  }
  cout << "max = " << max << endl;
}
```

```
Enter positive integers (0 to quit): 44 77 55 22 99 33 11 66 88 0
max = 99
```

This **`for`** loop is controlled by the input variable `n`; it continues until $n \leq 0$. When an input variable controls a loop this way, it is called a *sentinel*.

Note the control mechanism **`(max = n; n > 0; )`** in this **`for`** loop. Its update part is missing, and its initialization **`max = n`** has no declaration. The variable `max` has to be declared before the **`for`** loop because it is used outside of its block, in the last output statement in the program.

### EXAMPLE 4.16 Using a Loop Invariant to Prove that a `for` Loop is Correct

This program finds the minimum of a sequence of input numbers. It is similar to the program in Example 4.15:

```
int main()
{ int n, min;
  cout << "Enter positive integers (0 to quit): ";
  cin >> n;
  for (min = n; n > 0; )
  { if (n < min) min = n;
    // INVARIANT: min <= n for all n, and min equals one of the n
    cin >> n;
  }
  cout << "min = " << min << endl;
}
```

```
Enter positive integers (0 to quit): 44 77 55 22 99 33 11 66 88 0
min = 11
```

The full-line comment inside the block of the **`for`** loop is called a *loop invariant*. It states a condition that has two characteristic properties: (1) it is true at that point on every iteration of the loop; (2) the fact that it is true when the loop terminates proves that the loop performs correctly. In this case, the condition **`min <= n for all n`** is always true because the preceding **`if`** statement resets the value of `min` if the last input value of `n` was less than the previous value of `min`. And the condition that **`min equals one of the n`** is always true because **`min`** is initialized to the first `n` and the only place where `min` changes its value is when it is assigned to a new input value of `n`. Finally, the fact that the condition is true when the loop terminates means that `min` is the minimum of all the input numbers. And that outcome is precisely the objective of the `for` loop.

### EXAMPLE 4.17 More than One Control Variable in a `for` Loop

The **`for`** loop in this program uses two control variables:

```
int main()
{ for (int m=95, n=11; m%n > 0; m -= 3, n++)
    cout << m << "%" << n << " = " << m%n << endl;
}
```

```
95%11 = 7
92%12 = 8
89%13 = 11
86%14 = 2
83%15 = 8
```

The two control variables `m` and `n` are declared and initialized in the control mechanism of this **`for`** loop. Then `m` is decremented by 3 and `n` is incremented on each iteration of the loop, generating the sequence of (`m`,`n`) pairs (95,11), (92,12), (89,13), (86,14), (83,15), (80,16). The loop terminates with the pair (80,16) because 16 divides 80.

### EXAMPLE 4.18 Nesting `for` Loops

This program prints a multiplication table:

```
#include <iomanip>   // defines setw()
#include <iostream>  // defines cout
using namespace std;
int main()
{ for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
      cout << setw(4) << x*y;
    cout << endl;
  }
}
```

```
   1   2   3   4   5   6   7   8   9  10  11  12
   2   4   6   8  10  12  14  16  18  20  22  24
   3   6   9  12  15  18  21  24  27  30  33  36
   4   8  12  16  20  24  28  32  36  40  44  48
   5  10  15  20  25  30  35  40  45  50  55  60
   6  12  18  24  30  36  42  48  54  60  66  72
   7  14  21  28  35  42  49  56  63  70  77  84
   8  16  24  32  40  48  56  64  72  80  88  96
   9  18  27  36  45  54  63  72  81  90  99 108
  10  20  30  40  50  60  70  80  90 100 110 120
  11  22  33  44  55  66  77  88  99 110 121 132
  12  24  36  48  60  72  84  96 108 120 132 144
```

Each iteration of the outer `x` loop prints one row of the multiplication table. For example, on the first iteration when `x` = 1, the inner `y` loop iterates 12 times, printing `1*y` for each value of `y` from 1 to 12. And then on the second iteration of the outer `x` loop when `x` = 2, the inner `y` loop iterates 12 times again, this time printing `2*y` for each value of `y` from 1 to 12. Note that the separate `cout << endl` statement must be inside the outer loop and outside the inner loop in order to produce exactly one line for each iteration of the outer loop.

This program uses the *stream manipulator* `setw` to set the width of the output field for each integer printed. The expression `setw(4)` means to "set the output field width to 4 columns" for the next output.

This aligns the outputs into a readable table of 12 columns of right-justified integers. Stream manipulators are defined in the `<iomanip>` header, so this program had to include the directive

```
#include <iomanip>
```

in addition to including the `<iostream>` header.

### EXAMPLE 4.19 Testing a Loop Invariant

This program computes and prints the *discrete binary logarithm* of an input number (the greatest integer ≤ the base 2 logarithm of the number). It tests its loop invariant by printing the relevant values on each iteration:

```
#include <cmath>     // defines pow() and log()
#include <iostream>  // defines cin and cout
#include <iomanip>   // defines setw()
using namespace std;

int main()
{ long n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int d=0;  // the discrete binary logarithm of n
  double p2d=1;  // = 2^d
  for (int i=n; i > 1; i /= 2, d++)
  { // INVARIANT: 2^d <= n/i < 2*2^d
    p2d=pow(2,d);  // = 2^d
    cout << setw(2) << p2d << " <= " << setw(2) << n/i
         << " < " << setw(2) << 2*p2d << endl;
  }
  p2d=pow(2,d);  // = 2^d
  cout << setw(2) << p2d << " <= " << setw(2) << n
       << " < " << setw(2) << 2*p2d << endl;
  cout << "  The discrete binary logarithm of " << n
       << " is " << d << endl;
  double lgn = log(n)/log(2);  // base 2 logarithm of n
  cout << "The continuous binary logarithm of " << n
       << " is " << lgn << endl;
}
```

```
Enter a positive integer: 63
 1 <=  1 <  2
 2 <=  2 <  4
 4 <=  4 <  8
 8 <=  9 < 16
16 <= 21 < 32
32 <= 63 < 64
  The discrete binary logarithm of 63 is 5
The continuous binary logarithm of 63 is 5.97728
```

The discrete binary logarithm is computed to be the number of times the input number can be divided by 2 before reaching 1. So the **`for`** loop initializes `i` to `n` and then divides `i` by 2 once on each iteration. The counter `c` counts the number of iterations. So when the loop terminates, `c` contains the value of the discrete binary logarithm of `n`.

In addition to using the `setw()` function that is defined in the `<iomanip>` header, this program also uses the `log()` function that is defined in the `<cmath>` header. That function returns the natural

(base e) logarithm of $n$: `log(n)` = $\log_e n = \ln n$. It is used in the expression `log(n)/log(2)` to compute the binary (base 2) logarithm of $n$: $\log_2 n = \lg n = (\ln n)/(\ln 2)$. The printed results compare the discrete binary logarithm with the continuous binary logarithm. The former is equal to the latter truncated downward to its nearest integer (the *floor* of the number).

The loop invariant in this example is the condition **`2^d <= n/i < 2*2^d`** (*i.e.*, $2^d \le n/i < 2 \cdot 2^d$). It is tested by printing the values of the three expressions **`p2d`**, **`n`**, and **`2*p2d`**, where the quantity **`p2d`** is computed with the power function `pow()` that is defined in the `<cmath>` header.

We can prove that this **`for`** loop will always compute the discrete binary logarithm correctly. When it starts, $d = 0$ and $i = n$, so $2^d = 2^0 = 1$, $n/i = n/n = 1$, and $2 \cdot 2^d = 2 \cdot 1 = 2$; thus $2^d \le n/i < 2 \cdot 2^d$. On each iteration, $d$ increments and $i$ is halved, so $n/i$ is doubled. Thus the condition $2^d \le n/i < 2 \cdot 2^d$ remains invariant; *i.e.*, it is true initially and it remains true throughout the life of the loop. When the loop terminates, $i = 1$, so the condition becomes $2^d \le n/1 < 2 \cdot 2^d$, which is equivalent to $2^d \le n < 2^{d+1}$. The logarithm of this expression is $d = \lg(2^d) \le \lg n < \lg(2^{d+1}) = d+1$, so $d$ is greatest integer $\le \lg n$.

## 4.5 THE `break` STATEMENT

We have already seen the **`break`** statement used in the **`switch`** statement. It is also used in loops. When it executes, it terminates the loop, "breaking out" of the iteration at that point.

### EXAMPLE 4.20 Using a `break` Statement to Terminate a Loop

This program has the same effect as the one in Example 4.1 on page 60. It uses a **`break`** statement to control the loop:

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (true)
  { if (i > n) break;
    sum += i++;
  }
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

```
Enter a positive integer: 8
The sum of the first 8 integers is 36
```

As long as `(i <= n)`, the loop will continue, just as in Example 4.1. But as soon as `i > n`, the **`break`** statement executes, immediately terminating the loop.

The **`break`** statement provides extra flexibility in the control of loops. Normally a **`while`** loop, a **`do..while`** loop, or a **`for`** loop will terminate only at the beginning or at the end of the complete sequence of statements in the loop's block. But the **`break`** statement can be placed anywhere among the other statements within a loop, so it can be used to terminate a loop anywhere from within the loop's block. This is illustrated by the following example.

### EXAMPLE 4.21 Controlling Input with a Sentinel

This program reads a sequence of positive integers, terminated by 0, and prints their average:

```
int main()
{ int n, count=0, sum=0;
  cout << "Enter positive integers (0 to quit):" << endl;
  for (;;)  // "forever"
  { cout << "\t" << count + 1 << ": ";
    cin >> n;
    if (n <= 0) break;
    ++count;
    sum += n;
  }
  cout << "The average of those " << count << " positive numbers is "
       << float(sum)/count << endl;
}
```

```
Enter positive integers (0 to quit):
        1: 4
        2: 7
        3: 1
        4: 5
        5: 2
        6: 0
The average of those 5 positive numbers is 3.8
```

When 0 is input, the **break** executes, immediately terminating the **for** loop and transferring execution to the final output statement. Without the **break** statement, the `++count` statement would have to be put in a conditional, or `count` would have to be decremented outside the loop or initialized to –1.

Note that all three parts of the **for** loop's control mechanism are empty: **for** `(;;)`. This construct is pronounced "forever." Without the **break**, this would be an infinite loop.

When used within nested loops, the **break** statement applies only to the loop to which it directly belongs; outer loops will continue, unaffected by the break. This is illustrated by the following example.

### EXAMPLE 4.22 Using a `break` Statement with Nested Loops

Since multiplication is commutative (*e.g.*, 3×4 = 4×3), multiplication tables are often presented with the numbers above the main diagonal omitted. This program modifies that of Example 4.18 on page 69 to print a triangular multiplication table:

```
int main()
{ for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
      if (y > x) break;
      else cout << setw(4) << x*y;
    cout << endl;
  }
}
```

```
  1
  2   4
  3   6   9
  4   8  12  16
  5  10  15  20  25
  6  12  18  24  30  36
  7  14  21  28  35  42  49
  8  16  24  32  40  48  56  64
  9  18  27  36  45  54  63  72  81
 10  20  30  40  50  60  70  80  90 100
 11  22  33  44  55  66  77  88  99 110 121
 12  24  36  48  60  72  84  96 108 120 132 144
```

When `y > x`, the execution of the inner `y` loop terminates and the next iteration of the outer `x` loop begins. For example, when `x` = 3, the `y` loop iterates 3 times (with `y` = 1, 2, 3), printing `3 6 9`. Then on its 4th iteration, the condition `(y > x)` is true, so the **`break`** statement executes, transferring control immediately to the `cout << endl` statement (which is outside of the inner `y` loop). Then the outer `x` loop begins its 4th iteration with `x` = 4.

## 4.6 THE `continue` STATEMENT

The **`break`** statement skips the rest of the statements in the loop's block, jumping immediately to the next statement outside of the loop. The **`continue`** statement is similar. It also skips the rest of the statements in the loop's block, but instead of terminating the loop, it transfers execution to the next iteration of the loop. It continues the loop after skipping the remaining statements in its current iteration.

### EXAMPLE 4.23 Using `continue` and `break` Statements

This little program illustrates the **`continue`** and **`break`** statements:

```
int main()
{ int n;
  for (;;)
  { cout << "Enter int: ";  cin >> n;
    if (n%2 == 0) continue;
    if (n%3 == 0) break;
    cout << "\tBottom of loop.\n";
  }
  cout << "\tOutside of loop.\n";
}
```

```
Enter int: 7
        Bottom of loop.
Enter int: 4
Enter int: 9
        Outside of loop.
```

When `n` has the value 7, both **`if`** conditions are false and control reaches the bottom of the loop. When `n` has the value 4, the first **`if`** condition is true (4 is a multiple of 2), so control skips over the rest of the statements in the loop and jumps immediately to the top of the loop again to continue with its next iteration. When `n` has the value 9, the first **`if`** condition is false (9 is not a multiple of 2) but the second **`if`** condition is true (9 is a multiple of 3), so control breaks out of the loop and jumps immediately to the first statement that follows the loop.

## 4.7 THE goto STATEMENT

The **break** statement, the **continue** statement, and the **switch** statement each cause the program control to branch to a location other than where it normally would go. The destination of the branch is determined by the context: **break** goes to the next statement outside the loop, **continue** goes to the loop's continue condition, and **switch** goes to the correct case constant. All three of these statements are called *jump statements* because they cause the control of the program to "jump over" other statements.

The **goto** statement is another kind of jump statement. Its destination is specified by a label within the statement.

A *label* is simply an identifier followed by a colon placed in front of a statement. Labels work like the **case** statements inside a **switch** statement: they specify the destination of the jump.

Example 4.22 illustrated how a **break** normally behaves within nested loops: execution breaks out of only the innermost loop that contains the **break** statement. Breaking out of several or all of the loops in a nest requires a **goto** statement, as the next example illustrates.

### EXAMPLE 4.24 Using a goto Statement to Break Out of a Nest of Loops

```
int main()
{ const int N=5;
  for (int i=0; i<N; i++)
  { for (int j=0; j<N; j++)
    { for (int k=0; k<N; k++)
        if (i+j+k>N) goto esc;
        else cout << i+j+k << " ";
      cout << "* ";
    }
  esc: cout << "." << endl;  // inside the i loop, outside the j loop
  }
}
```

```
0 1 2 3 4 * 1 2 3 4 5 * 2 3 4 5 .
1 2 3 4 5 * 2 3 4 5 .
2 3 4 5 .
3 4 5 .
4 5 .
```

When the **goto** is reached inside the innermost `k` loop, program execution jumps out to the labeled output statement at the bottom of the outermost `i` loop. Since that is the last statement in the `i` loop, the `i` loop will go on to its next iteration after executing that statement.

When `i` and `j` are 0, the `k` loop iterates 5 times, printing `0 1 2 3 4` followed by a star `*`. Then `j` increments to 1 and the `k` loop iterates 5 times again, printing `1 2 3 4 5` followed by a star `*`. Then `j` increments to 2 and the `k` loop iterates 4 times, printing `2 3 4 5`. But then on the next iteration of the `k` loop, `i` = 0, `j` = 2, and `k` = 4, so `i+j+k` = 6, causing the **goto** statement to execute for the first time. So execute jumps immediately to the labeled output statement, printing a dot and advancing to the next line. Note that both the `k` loop and the `j` loop are aborted before finishing all their iterations.

Now `i` = 1 and the middle `j` loop begins iterating again with `j` = 0. The `k` loop iterates 5 times, printing `1 2 3 4 5` followed by a star `*`. Then `j` increments to 1 and the `k` loop iterates 4 times, printing `2 3 4 5`. But then on the next iteration of the `k` loop, `i` = 1, `j` = 2, and `k` = 3, so `i+j+k` = 6, causing the **goto** statement to execute for the second time. Again execution jumps immediately to the labeled output statement, printing a dot and advancing to the next line.

On the subsequent three iterations of the outer `i` loop, the inner `k` loop never completes its iterations because `i+j+4` is always greater than 5 (because `i` is greater than 2). So no more stars are printed.

Note that the labeled output statement could be placed inside any of the loops or even outside of all of them. In the latter case, the **`goto`** statement would terminate all three of the loops in the nest.

Also note how the labeled statement is indented. The convention is to shift it to the left one indentation level to make it more visible. If it were not a labeled statement, it would be indented as

```
        }
        cout << "." << endl;
    }
```

instead of

```
        }
    esc: cout << "." << endl;
    }
```

Example 4.24 illustrates one way to break out of a nest of loops. Another method is to use a flag. A *flag* is a boolean variable that is initialized to `false` and then later set to `true` to signal an exceptional event; normal program execution is interrupted when the flag becomes true. This is illustrated by the following example.

### EXAMPLE 4.25 Using a Flag to Break Out of a Nest of Loops

This program has the same output as that in Example 4.24:

```
int main()
{ const int N=5;
  bool done=false;
  for (int i=0; i<N; i++)
  { for (int j=0; j<N && !done; j++)
    { for (int k=0; k<N && !done; k++)
        if (i+j+k>N) done = true;
        else cout << i+j+k << " ";
      cout << "* ";
    }
    cout << "." << endl;  // inside the i loop, outside the j loop
    done = false;
  }
}
```

When the `done` flag becomes true, both the innermost `k` loop and the middle `j` loop will terminate, and the outer `i` loop will finish its current iteration by printing the dot, advancing to the beginning of the next line, and resetting the `done` flag to false. Then it starts its next iteration, the same as in Example 4.24.

## 4.8 GENERATING PSEUDO-RANDOM NUMBERS

One of the most important applications of computers is the *simulation* of real-world systems. Most high-tech research and development is heavily dependent upon this technique for studying how systems work without actually having to interact with them directly.

Simulation requires the computer generation of *random numbers* to model the uncertainty of the real world. Of course, computers cannot actually generate truly random numbers because computers are *deterministic*: given the same input, the same computer will always produce the

same output. But it is possible to generate numbers that appear to be randomly generated; *i.e.*, numbers that are uniformly distributed within a given interval and for which there is no discernible pattern. Such numbers are called *pseudo-random numbers*.

The Standard C header file `<cstdlib>` defines the function `rand()` which generates pseudo-random integers in the range 0 to `RAND_MAX`, which is a constant that is also defined in `<cstdlib>`. Each time the `rand()` function is called, it generates another **`unsigned`** integer in this range.

### EXAMPLE 4.26 Generating Pseudo-Random Numbers

This program uses the `rand()` function to generate pseudo-random numbers:

```
#include <cstdlib> // defines the rand() function and RAND_MAX const
#include <iostream>
using namespace std;

int main()
{ // prints pseudo-random numbers:
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
  cout << "RAND_MAX = " << RAND_MAX << endl;
}
```

```
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
RAND_MAX = 2147483647
```

```
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
RAND_MAX = 2147483647
```

On each run, the computer generates 8 **`unsigned`** integers that are uniformly distributed in the interval 0 to `RAND_MAX`, which is 2,147,483,647 on this computer. Unfortunately each run produces the same sequence of numbers. This is because they are generated from the same "seed."

Each pseudo-random number is generated from the previously generated pseudo-random number by applying a special "number crunching" function that is defined internally. The first pseudo-random number is generated from an internally defined variable, called the *seed* for the sequence. By default, this seed is initialized by the computer to be the same value every time the program is run. To overcome this violation of pseudo-randomness, we can use the `srand()` function to select our own seed.

### EXAMPLE 4.27 Setting the Seed Interactively

This program is the same as the one in Example 4.26 except that it allows the pseudo-random number generator's seed to be set interactively:

```
#include <cstdlib>  // defines the rand() and srand() functions
#include <iostream>
using namespace std;

int main()
{ // prints pseudo-random numbers:
  unsigned seed;
  cout << "Enter seed: ";
  cin >> seed;
  srand(seed);                          // initializes the seed
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
}
```

```
Enter seed: 0
12345
1406932606
654583775
1449466924
229283573
1109335178
1051550459
1293799192
```

```
Enter seed: 1
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
```

```
Enter seed: 12345
1406932606
654583775
1449466924
229283573
1109335178
1051550459
1293799192
794471793
```

The line `srand(seed)` assigns the value of the variable `seed` to the internal "seed" used by the `rand()` function to initialize the sequence of pseudo-random numbers that it generates. Different seeds produce different results.

Note that the `seed` value 12345 used in the third run of the program is the first number generated by `rand()` in the first run. Consequently the first through seventh numbers generated in the third run are the same as the second through eighth numbers generated in the first run. Also note that the sequence generated in the second run is the same as the one produced in Example 4.26. This suggests that, on this computer, the default seed value is 1.

The problem of having to enter a `seed` value interactively can be overcome by using the computer's system clock. The *system clock* keeps track of the current time in seconds. The `time()` function defined in the header file `<ctime>` returns the current time as an **`unsigned`** integer. This then can be used as the seed for the `rand()` function.

### EXAMPLE 4.28 Setting the Seed from the System Clock

This program is the same as the one in Example 4.27 except that it sets the pseudo-random number generator's seed from the system clock.

**Note:** if your compiler does not recognize the `<ctime>` header, then use the pre-standard `<time.h>` header instead.

```
#include <cstdlib>  // defines the rand() and srand() functions
#include <ctime>      // defines the time() function
#include <iostream>
//#include <time.h>   // use this if <ctime> is not recognized
using namespace std;
int main()
{ // prints pseudo-random numbers:
  unsigned seed = time(NULL);          // uses the system clock
  cout << "seed = " << seed << endl;
  srand(seed);                         // initializes the seed
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
}
```

Here are two runs using a UNIX workstation running a Motorola processor:

```
seed = 808148157
1877361330
352899587
1443923328
1857423289
200398846
1379699551
1622702508
715548277
```

```
seed = 808148160
892939769
1559273790
1468644255
952730860
1322627253
1305580362
844657339
440402904
```

On the first run, the `time()` function returns the integer 808,148,157 which is used to "seed" the random number generator. The second run is done 3 seconds later, so the `time()` function returns the integer 808,148,160 which generates a completely different sequence.

Here are two runs using a Windows PC running an Intel processor:

In many simulation programs, one needs to generate random integers that are uniformly distributed in a given range. The next example illustrates how to do that.

```
seed = 943364015
2948
15841
72
25506
30808
29709
13115
2527
```

```
seed = 943364119
17427
20464
13149
5702
12766
1424
16612
31746
```

### EXAMPLE 4.29 Generating Pseudo-Random Numbers in Given Range

This program is the same as the one in Example 4.28 except that the pseudo-random numbers that it generates are restricted to given range:

```
#include <cstdlib>
#include <ctime>      // defines the time() function
#include <iostream>
//#include <time.h>   // use this if <ctime> is not recognized
using namespace std;
int main()
{ // prints pseudo-random numbers:
  unsigned seed = time(NULL);          // uses the system clock
  cout << "seed = " << seed << endl;
  srand(seed);                         // initializes the seed
  int min, max;
  cout << "Enter minimum and maximum: ";
  cin >> min >> max;                  // lowest and highest numbers
  int range = max - min + 1;          // number of numbers in range
  for (int i = 0; i < 20; i++)
  { int r = rand()/100%range + min;
    cout << r << " ";
  }
  cout << endl;
}
```

Here are two runs:

```
seed = 808237677
Enter minimum and maximum: 1 100
85 57 1 10 5 73 81 43 46 42 17 44 48 9 3 74 41 4 30 68
```

```
seed = 808238101
Enter minimum and maximum: 22 66
63 29 56 22 53 57 39 56 43 36 62 30 41 57 26 61 59 26 28
```

The first run generates 20 integers uniformly distributed between 1 and 100. The second run generates 20 integers uniformly distributed between 22 and 66.

In the **for** loop, we divide `rand()` by 100 first to strip way the two right-most digits of the random number. This is to compensate for the problem that this particular random number generator has of producing numbers that alternate odd and even. Then `rand()/100%range` produces random numbers in the range `0` to `range-1`, and `rand()/100%range + min` produces random numbers in the range `min` to `max`.

## Review Questions

**4.1** What happens in a **while** loop if the control condition is false (*i.e.*, zero) initially?

**4.2** When should the control variable in a **for** loop be declared before the loop (instead of within its control mechanism)?

**4.3** How does the **break** statement provide better control of loops?

**4.4** What is the minimum number of iterations that

***a.*** a while loop could make?

***b.*** a **do..while** loop could make?

**4.5** What is wrong with the following loop:

```
while (n <= 100)
sum += n*n;
```

**4.6** If *s* is a compound statement, and *e1*, *e2*, and *e3* are expressions, then what is the difference between the program fragment:

```
for (e1; e2; e3)
s;
```

and the fragment:

```
e1;
while (e2)
{ s;
  e3;
}
```

**4.7** What is wrong with the following program:

```
int main()
{ const double PI;
  int n;
  PI = 3.14159265358979;
  n = 22;
}
```

**4.8** What is an "infinite loop," and how can it be useful?

**4.9** How can a loop be structured so that it terminates with a statement in the middle of its block?

**4.10** Why should tests for equality with floating-point variables be avoided?

## Problems

**4.1** Trace the following code fragment, showing the value of each variable each time it changes:

```
float x = 4.15;
for (int i=0; i < 3; i++)
  x *= 2;
```

**4.2** Assuming that *e* is an expression and *s* is a statement, convert each of the following **for** loops into an equivalent **while** loop:

***a.*** for ( ; e; ) s

***b.*** `for ( ; ; e) s`

**4.3** Convert the following **for** loop into a **while** loop:

```
for (int i=1; i <= n; i++)
  cout << i*i << " ";
```

**4.4** Describe the output from this program:

```
int main()
{ for (int i = 0; i < 8; i++)
    if (i%2 == 0) cout << i + 1 << "\t";
    else if (i%3 == 0) cout << i*i << "\t";
    else if (i%5 == 0) cout << 2*i - 1 << "\t";
    else cout << i << "\t";
}
```

**4.5** Describe the output from this program:

```
int main()
{ for (int i=0; i < 8; i++)
  { if (i%2 == 0) cout << i + 1 << endl;
    else if (i%3 == 0) continue;
    else if (i%5 == 0) break;
    cout << "End of program.\n";
  }
  cout << "End of program.\n";
}
```

**4.6** In a 32-bit **float** type, 23 bits are used to store the mantissa and 8 bits are used to store the exponent.

***a.*** How many significant digits of precision does the 32-bit **float** type yield?

b. What is the range of magnitude for the 32-bit **float** type?

**4.7** Write and run a program that uses a **while** loop to compute and prints the sum of a given number of squares. For example, if 5 is input, then the program will print 55, which equals $1^2 + 2^2 + 3^2 + 4^2 + 5^2$.

**4.8** Write and run a program that uses a **for** loop to compute and prints the sum of a given number of squares.

**4.9** Write and run a program that uses a **do..while** loop to compute and prints the sum of a given number of squares.

**4.10** Write and run a program that directly implements the quotient operator **/** and the remainder operator **%** for the division of positive integers.

**4.11** Write and run a program that reverses the digits of a given positive integer. (See Problem 3.13 on page 51.)

**4.12** Apply the *Babylonian Algorithm* to compute the square root of 2. This algorithm (so called because it was used by the ancient Babylonians) computes $\sqrt{2}$ by repeatedly replacing one estimate $x$ with the closer estimate $(x + 2/x)/2$. Note that this is simply the average of $x$ and $2/x$.

**4.13** Write a program to find the integer square root of a given number. That is the largest integer whose square is less than or equal to the given number.

**4.14** Implement the *Euclidean Algorithm* for finding the greatest common divisor of two given positive integers. This algorithm transforms a pair of positive integers $(m, n)$ into a pair $(d, 0)$ by repeatedly dividing the larger integer by the smaller integer and replacing the larger with

the remainder. When the remainder is 0, the other integer in the pair will be the greatest common divisor of the original pair (and of all the intermediate pairs). For example, if *m* is 532 and *n* is 112, then the Euclidean Algorithm reduces the pair (532,112) to (28,0) by

$$(532,112) \rightarrow (112,84) \rightarrow (84,28) \rightarrow (28,0).$$

So 28 is the greatest common divisor of 532 and 112. This result can be verified from the facts that $532 = 28 \cdot 19$ and $112 = 28 \cdot 8$. The reason that the Euclidean Algorithm works is that each pair in the sequence has the same set of divisors, which are precisely the factors of the greatest common divisor. In the example above, that common set of divisors is {1, 2, 4, 7, 14, 28}. The reason that this set of divisors is invariant under the reduction process is that when $m = n \cdot q + r$, a number is a common divisor of *m* and *n* if and only if it is a common divisor of *n* and *r*.

## Answers to Review Questions

**4.1** If the control condition of a `while` loop is initially false, then the loop is skipped altogether; the statement(s) inside the loop are not executed at all.

**4.2** The control variable in a **`for`** loop has to be declared before the loop (instead of within its control mechanism) if it is used outside of the loop's statement block, as in Example 4.14 on page 67.

**4.3** The **`break`** statement provides better control of loops by allowing immediate termination of the loop after any statement within its block. Without a **`break`** statement, the loop can terminate only at the beginning or at the end of the block.

**4.4** ***a.*** The minimum number of iterations that a **`while`** loop could make is 0.

***b.*** The minimum number of iterations that a **`do..while`** loop could make is 1.

**4.5** That is an infinite loop because the value of its control variable `n` does not change.

**4.6** There is no difference between the effects of those two program fragments, unless *s* is a **`break`** statement or *s* is a compound statement (*i.e.*, a block) that contains a **`break`** statement or a **`continue`** statement. For example, this **`for`** statement will iterate 4 times and then terminate normally:

```
for (i = 0; i < 4; i++)
  if (i == 2) continue;
```

but this **`while`** statement will be an infinite loop:

```
i = 0;
while (i < 4)
{ if (i == 2) continue;
  i++;
}
```

**4.7** The constant `PI` is not initialized. Every constant must be initialized at its declaration.

**4.8** An infinite loop is one that continues without control; it can be stopped only by a branching statement within the loop (such as a **`break`** or **`goto`** statement) or by aborting the program (*e.g.*, with Ctrl+C). Infinite loops are useful if they are stopped with branching statements.

**4.9** A loop can be terminated by a statement in the middle of its block by using a **`break`** or a **`goto`** statement.

**4.10** Floating-point variables suffer from round-off error. After undergoing arithmetic transformations, exact values may not be what would be expected. So a test such as `(y == x)` may not work correctly.

## Solutions to Problems

**4.1** First, `x` is initialized to 4.15 and `i` is initialized to 0. Then `x` is doubled three times by the three iterations of the **`for`** loop.

**4.2** The equivalent **`while`** loops are:

***a.*** `while (e) s;`

***b.*** `while (true) { s; e; }`, assuming that *s* contains no **`break`** or **`continue`** statements.

**4.3** The equivalent **`while`** loop is:

```
int i=1;
while (i <= n)
{ cout << i*i << " ";
  i++;
}
```

**4.4** The output is

```
1       1       3       9       5       9       7       7
```

**4.5** The output is

```
End of program.
End of program.
3
End of program.
5
End of program.
End of program.
```

**4.6** ***a.*** The 23 bits hold the 2nd through 24th bit of the mantissa. The first bit must be a 1, so it is not stored. Thus 24 bits are represented. These 24 bits can hold $2^{24}$ numbers. And $2^{24}$ = 16,777,216, which has 7 digits with full range, so 7 complete digits can be represented. But the last digit is in doubt because of rounding. Thus, the 32-bit `float` type yields 6 significant digits of precision.

***b.*** The 8 bits that the 32-bit `float` type uses for its exponent can hold $2^8$ = 256 different numbers. Two of these are reserved for indicating underflow and overflow, leaving 254 numbers for exponents. So an exponent can range from –126 to +127, yielding a magnitude range of $2^{-126} = 1.175494 \times 10^{-38}$ to $2^{127} = 1.70141 \times 10^{38}$.

**4.7** This program uses a **`while`** loop to compute the sum of the first *n* squares, where *n* is input:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0, i=0;
  while (i++ < n)
    sum += i*i;
  cout << "The sum of the first " << n << " squares is "
       << sum << endl;
}
```

```
Enter a positive integer: 6
The sum of the first 6 squares is 91
```

**4.8** This program uses a **`for`** loop to compute the sum of the first *n* squares, where *n* is input:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0;
  for (int i=1; i <= n; i++)
```

```
    sum += i*i;
  cout << "The sum of the first " << n << " squares is "
       << sum << endl;
}
```

```
Enter a positive integer: 6
The sum of the first 6 squares is 91
```

**4.9** This program uses a **`do..while`** loop to compute the sum of the first *n* squares, where *n* is input:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0, i=1;
  do
  { sum += i*i;
  }
  while (i++ < n);
  cout << "The sum of the first " << n << " squares is "
       << sum << endl;
}
```

```
Enter a positive integer: 6
The sum of the first 6 squares is 91
```

**4.10** This program directly implements the quotient operator `/` and the remainder operator `%` for the division of positive integers. The algorithm used here, applied to the fraction `n/d`, repeatedly subtracts the `d` from the `n` until `n` is less than `d`. At that point, the value of `n` will be the remainder, and the number `q` of iterations required to reach it will be the quotient:

```
int main()
{ int n, d, q, r;
  cout << "Enter numerator: ";
  cin >> n;
  cout << "Enter denominator: ";
  cin >> d;
  for (q = 0, r = n; r >= d; q++)
    r -= d;
  cout << n << " / " << d << " = " << q << endl;
  cout << n << " % " << d << " = " << r << endl;
  cout << "(" << q << ")(" << d << ") + (" << r << ") = "
       << n << endl;
}
```

```
Enter numerator: 30
Enter denominator: 7
30 / 7 = 4
30 % 7 = 2
(4)(7) + (2) = 30
```

This run iterated 4 times: 30 – 7 = 23, 23 – 7 = 16, 16 – 7 = 9, and 9 – 7 = 2. So the quotient is 4, and the remainder is 2. Note that this relationship must always be true for integer division:

```
(quotient)(denominator) + (remainder) = numerator
```

**4.11** The trick here is to strip off the digits one at a time from the given integer and "accumulate" them in reverse in another integer:

```
int main()
{ long m, d, n = 0;
  cout << "Enter a positive integer: ";
  cin >> m;
```

```
  while (m > 0)
  { d = m % 10;     // d will be the right-most digit of m
    m /= 10;        // then remove that digit from m
    n = 10*n + d;   // and append that digit to n
  }
  cout << "The reverse is " << n << endl;
}
```

```
Enter a positive integer: 123456
The reverse is 654321
```

In this run, `m` begins with the value 123,456. In the first iteration of the loop, `d` is assigned the digit 6, `m` is reduced to 12,345, and `n` is increased to 6. On the second iteration, `d` is assigned the digit 5, `m` is reduced to 1,234, and `n` is increased to 65. On the third iteration, `d` is assigned the digit 4, `m` is reduced to 123, and `n` is increased to 654. This continues until, on the sixth iteration, `d` is assigned the digit 1, `m` is reduced to 0, and `n` is increased to 654,321.

**4.12** This implements the Babylonian Algorithm:

```
#include <cmath>  // defines the fabs() function
#include <iostream>
using namespace std;
int main()
{ const double TOLERANCE = 5e-8;
  double x = 2.0;
  while (fabs(x*x - 2.0) > TOLERANCE)
  { cout << x << endl;
    x = (x + 2.0/x)/2.0;  // average of x and 2/x
  }
  cout << "x = " << x << ", x*x = " << x*x << endl;
}
```

```
2
1.5
1.41667
1.41422
x = 1.41421, x*x = 2
```

We use a "tolerance" of `5e-8` (= 0.00000005) to ensure accuracy to 7 decimal places. The `fabs()` function (for "floating-point absolute value"), defined in the `<cmath>` header file, returns the absolute value of the expression passed to it. So the loop continues until `x*x` is within the given tolerance of 2.

**4.13** This program finds the integer square root of a given number. This method uses an "exhaustive" algorithm to find all the positive integers whose square is less than or equal to the given number:

```
int main()
{ float x;
  cout << "Enter a positive number: ";
  cin >> x;
  int n = 1;
  while (n*n <= x)
    ++n;
  cout << "The integer square root of " << x << " is "
       << n-1 << endl;
}
```

```
Enter a positive number: 1234.56
The integer square root of 1234.56 is 35
```

It starts with `n=1` and continues to increment `n` until `n*n > x`. When the **`for`** loop terminates, `n` is the smallest integer whose square is greater than `x`, so `n-1` is the integer square root of `x`. Note the use of the *null statement* in the **`for`** loop. Everything that needs to be done in the loop is done within the control parts of the loop. But the semicolon is still necessary at the end of the loop.

**4.14** This implements the Euclidean Algorithm:

```
int main()
{ int m, n, r;
  cout << "Enter two positive integers: ";
  cin >> m >> n;
  if (m < n) { int temp = m; m = n; n = temp; }    // make m >= n
  cout << "The g.c.d. of " << m << " and " << n << " is ";
  while (n > 0)
  { r = m % n;
    m = n;
    n = r;
  }
  cout << m << endl;
}
```

```
Enter two positive integers: 532 112
The g.c.d. of 532 and 112 is 28
```