Chapter 2  Getting Started—Program Structure, screen output, and Comments

Chapter Topics

In this chapter, you will learn how to:

- Write simple C++ programs
- Display keyboard symbols on the screen
- Write program comments
- Display table headings
- Debug your first programs

The only way to learn programming is to write programs. You will find that the more time you spend at a computer, the more you will learn. In this chapter we show a number of programs that illustrate the basic form and structure of a C++ program. You should begin programming by imitating what we show and then customizing your program to meet your needs.

This chapter’s lessons show you how to create simple text output and how to add documentation (in the form of comments) to a program. The application examples illustrate how to print neat tabular headings and how to detect program errors. In themselves, the example programs described in this chapter have limited practical use. However, portions of these programs form integral pieces of many valuable C++ programs.

LESSON 2.1  BASIC STRUCTURE

TOPICS

- Writing a simple C++ program.
- Using the cout object to display text on the screen.
- Structure of a simple C++ program.
- Basic rules for writing a C++

The program that follows illustrates the basic structure of a C++ program. When you execute the program, the statement

This is C++!

appears on the screen and remains there until other tasks eliminate it. Examine the program and the output carefully before you read the description. The program is written in what is called code. You may understand why it is called code because it looks cryptic. Since at this point you know nothing about C++, you probably will find the first few lessons difficult. As you learn more, though, you will find that indeed you can interpret much of the code yourself (assisted by the boxed annotations) before we explain it fully to you.

Read the source code line by line, and follow the boxed annotations.
**Headers**  
C++ has many built-in features; too many to be automatically included in every program. In this lesson’s program, we need the C++ output system. Therefore, we have the line

```
#include <iostream>
```

This causes the C++ preprocessor to take code existing in a file (associated with the name iostream that is already in the C++ system) and group it with our program. All the code (that is, our code and the iostream code) is compiled to produce a single package of machine language instructions. With this particular line, our program is allowed access to C++ I/O (input/output) features. For this program it allows us to easily print output to the screen using the cout object in the program body. In almost all programs in this text, exactly this line is needed.

The C++ preprocessor is the part of the integrated development environment (IDE) that performs actions prior to the translation of the C++ program into machine language instructions. What the preprocessor does with headers (also called header files) is shown conceptually in Fig. 2.1. This figure illustrates the C++ preprocessor grouping existing code with our code. Not all C++ preprocessors work exactly as this figure illustrates, but the effect is the same—that using a statement that begins with `#include` gives a program additional information that allows it to access built-in features of C++.  

**Description**

**Source Code**

<table>
<thead>
<tr>
<th>Line</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;iostream&gt;</code></td>
<td>This line causes information about input and output to be included with the program. Use exactly this line in your programs.</td>
</tr>
<tr>
<td><code>using namespace std;</code></td>
<td>This line allows us to easily access C++ built-in resources for such things as input and output. Use exactly this line in your programs.</td>
</tr>
<tr>
<td><code>int main( )</code></td>
<td>This indicates that the <code>main</code> function is about to begin. Use exactly this line in your programs.</td>
</tr>
<tr>
<td><code>{</code></td>
<td>This brace begins a block of code, which in this case is the body of the <code>main</code> function. Use exactly this line in your programs.</td>
</tr>
<tr>
<td><code>cout &lt;&lt; &quot;This is C++!&quot;;</code></td>
<td>This line prints the words &quot;This is C++!&quot; on the screen. In your programs, modify this line to print the words you want printed.</td>
</tr>
<tr>
<td><code>}</code></td>
<td>This brace ends a block of code, which in this case is the body of the <code>main</code> function. Use exactly this line in your programs.</td>
</tr>
</tbody>
</table>
Figure 2.1 Conceptual illustration of the action performed by the C++ preprocessor caused by the line of code `#include <iostream>.

We will find that when we want to use standard C++ math functions such as `sin()` or `cos()` in the program body that we need to use the `cmath` header with the line

```cpp
#include <cmath>
```

We can use a number of headers in C++, and we describe these in other chapters. However, for most of the programs in this book, we include only two to four standard C++ headers per program, and therefore this is not a major part of the programming process.

A popular list of headers is:

- algorithm
- cstdlib
- iostream
- set
- bitset
- cstring
- iterator
- sstream
- cassert
- ctime
- limits
- stack
- cctype
- deque
- list
- stl_except
- cfloat
- fstream
- locale
- string
- climits
- functional
- map
- typeinfo
- cmath
- exception
- memory
- utility
- cstdio
- iomanip
- queue
- vector
We use many of these headers in this text. You do not need to understand them now. Throughout the text we tell you when you need each header and why. You will find that if you do not include the proper header, the C++ compiler will send you an error message when you attempt to compile your program, and your program will not compile successfully.

**Namespaces** A namespace is somewhat like the name (or number) of a chapter in a book for avoiding naming clash. The name or number of the chapter uniquely indicates a particular region of the book where, for instance, certain terms are defined. If, at another point in the book, we want to refer to some terms that are defined in that chapter, we need to indicate the chapter name each time we refer to material in that chapter to be perfectly clear. If we make many references to terms in that chapter we would need many statements indicating the chapter name. You could imagine that although this could be done, writing and reading the text could be very tedious because of the repetition of the chapter references.

An alternative to constantly referring to the chapter would be to state at the beginning of the section which chapter is being referred to. For instance, if we were referring to Chapter 5 in a particular section, we could say at the beginning of the section something like, “In this section, terms not defined here are defined in Chapter 5.” With this statement it is no longer necessary to mention Chapter 5 for every reference. A much abbreviated way to state this would be to have the statement, “Using Chapter 5” at the beginning of the section. This is more cryptic than the longer statement, but it is understandable if it were an agreed upon convention for making references to other chapters.

C++ has such a convention for making references. C++ has something called the standard namespace which is abbreviated std. The standard namespace is somewhat like a different chapter in a book. At the beginning of our program we use the statement

```cpp
using namespace std;
```

which indicates that certain terms not defined in the program are defined in the standard namespace (which is invisible to the reader of our program’s source code but is built into C++)

In this particular program, the cout object is within the standard namespace. A typical program in this textbook uses cout many times. Without the using namespace std; statement, we would need to refer to the standard namespace each time we use cout in the following way (for example, for the statement in this lesson’s program):

```cpp
std :: cout << "This is C++";
```

This statement explicitly indicates that cout is in the standard namespace. By having the using namespace std; statement in the program, we do not need to explicitly state that cout is in the standard namespace each time we use it. We can simply write:

```cpp
cout << "This is C++";
```

If you only want to authorize the use of cout object instead of everything defined under the namespace std, you should use “using std::cout” instead of “using namespace std”. In all the programs in this book,
we have the “using namespace std;” statement and do not need to concern ourselves with namespaces beyond this. At this point, we do not need to go into more detail with namespaces.

The main function  Typical C++ programs are composed of classes, objects, and functions among others things. C++ not only allows you to use functions in its library (such as sin() or cos() ), it allows you to write your own functions. Every C++ program has a primary function or starting point that must be assigned the name main. The name main is mandatory and cannot be altered by you, the programmer. The function must be called main. The C++ compiler searches for the function named main and compiles the program in a manner to ensure that main is the first function executed.

The first line of the main function in this lesson’s program is

```cpp
int main( )
```

This line gives the name of the function main( ) and information about what goes into and out of the function. The int indicates that main returns an integer value to the operating system (which is considered to be the unit that calls the function main) when main finishes execution. In standard C++, the value 0 is automatically returned to the operating system upon successful execution of main. You could also use the return keyword to pass back any integer for indication of any problem encountered. The empty parentheses indicate that no information is passed from the operating system to the main( ) function. We will not go into more detail here. For now, you should simply memorize the form of this line because you will use it in all your programs.

After the line that contains the function name is the function body. The function body has the following features:

- It begins with an opening brace {.
- It ends with a closing brace }.
- The pair of braces, {}, are used to enclose what is called a block of code. We use braces quite frequently to form blocks of code. Sometimes we use blocks within blocks. In this case, the braces enclose the block of code that is the function body.
- The function body consists of C++ declaration(s) and statement(s). The structure of a simple C++ main function is as follows:

```cpp
int main( )
{
    declaration 1;
    declaration 2;
    statement 1;
    statement 2;
}
```

In this chapter, we discuss some statements you can use. In Chapter 3, we discuss declarations.

The cout object  In this lesson’s program, we have only one executable statement

```cpp
cout << "This is C++!";
```
It uses the cout object and the insertion operator, <<, to print the characters enclosed in double quotes to the screen. An object is a region of storage in memory, and in C++ programs, the cout region of memory is linked to the standard output device (usually the screen). The line enclosed in double quotes, "This is C++!" is called a string constant or string literal. The string constant is stored in another region of memory when the program is executed. The insertion operator is called that because it inserts a copy of information into the cout region of memory. In this case, a copy of the string constant "This is C++!" is inserted into cout and then is transferred to the screen. This is illustrated schematically in Fig. 2.2.

![Figure 2.2](image)

**Figure 2.2** Schematic of action of a cout statement to print a string to the screen. The insertion operator (<<) copies the string to cout. Then the string is passed to the screen.

We will use cout in most of the programs in this book. You should become adept at using cout in your programs. We discuss cout in detail in Chapters 2 and 3.

**C++ syntax**

C++ has rules for writing statements. These are called syntax rules. The syntax rules must be followed strictly by a programmer. Most programmer syntax errors will be detected by the compiler at compile time (that is, when you compile the program). Application Program 2.2 describes in detail how to correct syntax errors. Here, we describe some syntax rules. Throughout the text we describe the syntax rules for each C++ feature introduced.

- **Semicolons**—The line cout "This is C++!"; is an executable C++ statement. Executable C++ statements appear in the body of functions and must be terminated by a semicolon. The semicolon at the end of a C++ statement acts much like a period at the end of a sentence. It serves as a statement terminator. A typical program has many C++ statements, each terminated with a semicolon. Some (but not all) other types of statements also require semicolons. The using namespace std; statement is an example of another type of statement that requires a semicolon. Throughout this book, we indicate where semicolons are required and not required.

- **Case sensitivity**—The C++ language distinguishes between lower and uppercase letters. Thus, cout is different from COUT, Cout, or CoUt. It is said, therefore, that C++ is considered to be case sensitive. In this lesson, all letters except those between the double quotes must be written in lowercase letters. When you begin naming your own variables and functions, you may use whatever case you consider appropriate. However, C++ traditionally is written primarily in lowercase letters. We describe situations when other than lowercase letters commonly are used, as the need arises.
Blank spaces—C++ code consists of a number of tokens. A C++ token is the smallest element that the C++ compiler does not break down into smaller parts. A token can be a function name (e.g., main), an object name (e.g., cout), or a C++ reserved word, which we discuss in Lesson 3.1. All C++ words should be written continuously. For example, the line

```cpp
int main()
```

is not legal because no blank characters are allowed between the characters ‘a’ and ‘l’ in the word main.

Between tokens, extra white-space characters (white-space characters are blank, tab, and the carriage return “Enter”) can be inserted, but this is optional. For example, the line

```cpp
int main ( )
```

is equivalent to

```cpp
int main ( )
```

or

```cpp
int main ( )
```

In general, it is acceptable to add blanks between tokens, but not acceptable to add blanks within tokens.

Spacing—A white-space type character is created when you press the Enter key. In most cases, when extra spaces are used between tokens, they are invisible to the C++ compiler. Therefore, you have the freedom to write most of your C++ code at any row or column you like. The C++ compiler, for example, allows you to rewrite and pack this lesson’s program into two lines

```cpp
#include <iostream>
using namespace std; int main(){cout<<"This is C++!";}
```

or rewrite it as

```cpp
#include<iostream>
using namespace std; int main( ) { cout<<"This is C++!"; }
```

However, these styles will make your program more difficult to understand and should not be used.

There is no required form for spacing within your program. However, your instructor or employer may want you to adhere to certain standard accepted styles. Our example programs are meant to illustrate acceptable style; however, at times, publishing constraints do not allow us to follow any one accepted style rigorously. Indentation and spacing (both within a line and
between lines) are considered important for the look of a program, even though they do not affect performance.

Within a line, many programmers like to put a space before and after operators (which we discuss in more detail in Chapter 3). For instance, the << in a cout statement is an operator, and many programmers put a space before and after it.

In general, to make your programs readable, do such things as write one statement per line, line up your braces, and add blank lines and spaces where there are natural breaks in code instructions. Remember, the look of a program is important because programs continually undergo change. A program that is neat and organized is easier to understand and therefore easier to modify. As a result, the likelihood for error is reduced in programs that follow a certain visual and organizational style compared to those that do not.

Accepted modifications—Each C++ IDE (also called C++ implementation) is somewhat different. Some are more lenient than standard C++ and allow some statements to be omitted. For instance, the following program may compile and work like this lesson’s program. However, it is not adhere to standard and should be avoided.

```cpp
using namespace std;
main()
{
    cout << "This is C++!";
}
```

The reason it may work is that

1. The iostream header is used so frequently in C++ that, for some IDEs, it is included automatically to programs despite no specific direction to do this.

2. Even though we have not written int before main, C++ assigns a default type for this. The word default is used commonly in computing. A default value is a value that is used when none is specified. A default type is a type that is used when none is specified. Without going into detail, because C++ assigns a default type for the missing int, this program may work.

Summary  In summary, this lesson’s program, line by line, does the following:
LESSON 2.2   WRITING COMMENTS

TOPICS

- Reasons for writing comments.
- Structure of comments.
- Location of comments.
- Continuing comments.
- Style of writing comments.

Sometimes even the most experienced programmers find it difficult to understand the tasks that a program or a portion of a program are to perform. You very likely will find that you have trouble understanding code you have written! Also, others may be working with your code and find that what you have written is difficult to understand.

Fortunately, C++ allows you to write comments within the code of your programs. Comments are notes describing what a particular portion of your program does and how it does it (or anything else you would like to write in the middle of your code). Comments are a very important addition to programs because they convey information that is difficult to convey through the code itself. When written well, comments reduce the likelihood for error because a programmer making changes to a program can read the comments to understand how a program operates. Comments serve as an important part of the program documentation and therefore are required by employers and instructors.

A properly written comment performs no action in a program. If not written correctly, though, the C++ compiler may think that the words written in the comments are objects, such as cout, or other C++ expressions or statements. If this occurs, the compiler will indicate that your program has errors when you compile it.

The program that follows is not meant to be an example of good commenting but only an illustration of the mechanics of commenting. It is meant to illustrate a program that performs a task identical to that performed by the program in Lesson 2.1. It prints a line of code to the screen. However, unlike Lesson 2.1’s program, it is filled with comments. Compare this code to Lesson 2.1’s code and note the differences.

**Source Code**

```cpp
// This is a single line comment.
#include <iostream>
using namespace std;
int main ()
{ /* This is a multiline
   comment */
   cout<< "Comment structure lesson.; //End of line comment.
} // A comment can be written at the end of a program.
```

Single line comments begin with two slashes. Comments perform no actions.

Multiline comments begin with /* and end with */.

Comments can be placed after executable statements or at the end of a program.
Description

Single line comment structure  The structure of a single line comment is

    // comments

where there should be no blanks between the two slashes. All the comments must be on a single line and consist of acceptable C++ symbols (listed in Table 3.5). A comment causes no action to take place. It simply is a message to a reader of the source code.

A comment does not necessarily need to be alone on the line. It can be placed after a C++ statement. For instance,

    cout << "Comment structure lesson.";  //End of line comment.

puts a comment on the same line but after the cout statement. Note that we cannot place the comment before the cout statement like this

    //End of line comment. cout << "Comment structure lesson.";

because the cout portion would be regarded as part of the comment and not an executable statement.

Multiline comment structure  The structure of a multiline comment is

    /* comments */

where there should be no blanks between the / and *. The /* and */ are called comment delimiters.

The /* and */ must form a couple, but they need not be on the same line. Therefore, such a comment may occupy more than one line. A multiline comment starts with /*, followed by multiple lines of text consisting of numbers, characters, or symbols. The multiline comment terminates with */. For example,

    /* This is a multiline
       comment */

These are examples of incorrect multiline comments:

    /* Wrong comment 1, no
       end asterisk and slash
    /* Wrong
       comment 2,      no end slash *
    / *Wrong comment 3, there is
       a space between /and * */

Comment locations  A comment line can be written on the very first line, very last line, or in the middle of a program. The C++ compiler treats comments like a single white-space character. Therefore, comments can appear anywhere a white-space character is allowed. This means that comments may be placed on any line. Within a line, a comment can appear between tokens, but not within a token. (Note: a string constant is a token.)
Usage of comments  Be wise in your use of comments. Remember that comments enhance the understandability of your programs. Make them pleasing to the eye and clear. With practice, you can develop a clear style of writing comments. In this text, we do not use many comments because we have the ability to write boxed annotations that stand out more clearly than comments do. Your programs should use comments far more frequently than we use them.

We recommend that you avoid writing comments on the same line as other C++ code unless you can clearly distinguish the comments from the code by using many tabs.

There is no standard for writing comments, but we like a style that highlights comments and separates them from other C++ text. The reason is, if not highlighted, comments tend to blend in with the rest of the code and make following the logic of the code confusing. In other words, do not hide your comments. Make them stand out—they are there to help you and others. You can make your comments stand out by using repeating stars (*) or some other symbol that is easily distinguished from the rest of the code.

We strongly recommend that you add a banner at the beginning of your program. A banner is a set of comments that describe such things as the name, parameters used, history, author, purpose, and date of the program. A better look for the program for this lesson is with a banner as follows:

```cpp
//*******************************************************************************
// Name: Lesson 2.2
// Purpose: Learning how to write comments in C++
// Date: Written on 11/22/2005
// Author: Joe Kelly
// Reference: None
//*******************************************************************************
#include <iostream>
using namespace std;
int main()  
{
    cout << "Comment structure lesson";
}
```

The disadvantage of making comments stand out is that it takes time to type them in this way. When you are in a hurry, you will tend to skip the comments. Do not let this style of programming continue for very long. Plan ahead to set aside half an hour or so each day to do nothing but write comments. In the long run, the time that you spend writing comments will save you considerable frustration in finding errors in your programs. As you gain experience with programming, you will become aware of how many and what types of comments are useful.

Take writing comments seriously. You will be regarded as a better programmer and your programs will have fewer errors if you write comments properly. Your coworkers and employer will appreciate a good commenting style.
**Nested comments** Comment statements cannot be nested (meaning that we cannot write a comment within a comment) in C++. For example,

```cpp
//**/ This is an illegal comment because it is */ nested */
```

is not legal.

**“Commenting out” code** We would never find a need to deliberately write a nested comment, but it is very easy to create one accidentally. A common technique for isolating operations of source code is to “comment them out,” that is, remove some of the operations temporarily by converting them into comments to see the effect on the program’s performance. For instance, suppose we were working with this lesson’s program (meaning we were modifying and running it repeatedly to create a new program). We could prevent the `cout` statement from executing by putting comment delimiters `/*` and `*/` before and after it to give

```cpp
/*
   cout << "Comment structure lesson";
*/
```

If we compile and run the program with the code like this, the compiler will think that the `cout` statement is a comment and not convert it into machine language instructions.

It then is a simple task to later recreate the line of code by removing the comment delimiters. In other words, if you had deleted the line of code to prevent its execution, you would have had to retype it to get it back. Commenting the line out saves you typing and easily gives you exactly what you had before. For this program, such action would not help us very much, but for other programs, you will find this technique quite useful for finding errors in your programs.

However, if we make a mistake in where we put our comment delimiters, we accidentally could enclose a comment with the statements we want to comment out. For instance, suppose we wanted to comment out a large number of lines, and these many lines included a comment that we did not see. A convenient way of commenting out a large number of lines is to simply put comment delimiters on the lines above and below the section of code we are commenting out as shown here.

```cpp
/*
   executable statement 1
   executable statement 2
   executable statement 3
   */ comment */
   executable statement 4
   executable statement 5
*/
```

An error would have been generated during compilation because we would have accidentally created a comment within a comment, which is illegal. Remember, if you comment out code while you are modifying your programs, make sure you do not accidentally create a nested comment. You may also use `#if` and `#endif` preprocessor directive instead of using `/*` and `*/` to avoid the problem.
LESSON 2.3  CREATING NEW LINES IN OUTPUT

TOPICS
- Formatting output.
- Line feeding.
- Connecting strings.
- Other escape sequences.

The first example program of this chapter showed how to print a single line to the screen. However, in most cases you will want to print multiple lines to the screen, and you will want to display these lines so they have proper spacing. Proper spacing can be achieved by what is called line feeding. This program shows two ways to create a new line on the screen. The first is using \n in a string constant, and the second is using the endl manipulator.

Included in the Description, but not shown in this lesson’s program, are other output formatting features. For instance, we can use a \ in a string constant to connect a string constant part on one line with another string constant part on another line. Also, we can insert symbols in string constants to do such things as indent a tab or backspace. We do not describe all the possibilities but list them in a table.

Read the code to see how to create new lines, and read the Description to see how to connect string constants.

```
#include <iostream>
using namespace std;
int main() {
    cout << "This is";
    cout << "C++!";
    cout << "\nWe can\njump\n\ntwo lines."
    cout << endl;
    cout << "Here, we show 2 ways to\ncreate a new line." << endl;
}
```

Output
This is C++!
We can
jump
two lines.
Here, we show 2 ways to
create a new line.
Creating a new line with \n  A new line is not automatically created for each cout statement executed. You, the programmer, must specify the creation of a new line at each location you need one. For instance,

```cpp
    cout << "This is";
    cout << "C++!";
```

prints "This is C++!" on one line not two. In fact there is no space between “is” and “C++!” because no space is contained between these words and the double quotes.

A new line can be created using the linefeed symbol, \n, in the string constant. The symbol \n consists of two characters, \ (backslash, not to be confused with slash, /) and n, with no blank in between. In C++, the two character symbol \n is one of many character escape sequences. The C++ compiler considers an escape sequence within a string constant as one character (not two). The importance of this will be seen in Lesson 3.3. The escape sequence \n causes the cursor to move to the next line and will not be displayed on the screen. Any data behind this symbol is written at the beginning of the next line. You can use \n at any location in the string constant. The number of \n can be more than one. For example, in the statement

```cpp
    cout << "\nWe can\njump\n\ntwo lines.\n";
```

the program uses the first \n to move the cursor to a new line, displays “We can”, uses the second \n to jump to a new line, prints “jump”, uses the next two \n to jump another two lines, and prints “two lines.”

Creating a new line with endl  We can also use the endl manipulator to create a new line. The statement

```cpp
    cout << endl;
```

performs the single action of creating a new line, and the statement

```cpp
    cout << "Here, we show 2 ways to\ncreate a new line." << endl;
```

creates a new line between “to” and “create” using \n and a new line after “line.” using endl. Each endl in a cout statement must be alone between the << operators. Some forms for using endl are:

```cpp
    cout << endl << endl << endl;
    cout << endl << "string constant" << endl;
```

where string constant is any text. Whether to use \n or endl to create a new line is primarily a matter of preference (although there are some small differences between the mechanics of \n and endl, for example, endl guarantees the output will be flushed before the program moves on) for a programmer. Generally if many new lines are to be created within a string, \n is more convenient. Otherwise, endl is usually easier to write.
Connecting strings  Here, we illustrate two methods for connecting strings (not shown in this lesson’s program). In method 1, we use a backslash at the end of a line to indicate that a string constant in the source code has not finished and continues on the next line. For example, the statement

```cpp
    cout << "We can connect \nstrings on two lines." << endl << endl;
```

is equivalent to

```cpp
    cout << "We can connect strings on two lines." << endl << endl;
```

Since the C++ compiler disregards all blank characters behind a statement, the connection to the next line starts at the end of the preceding statement. If you want to include blank characters in a string constant that occupies two lines, either place them before the backslash in the first line or at the beginning of the second line. In method 2, we enclose each unfinished string constant in double quotes; for example, the statement

```cpp
    cout << "We can " "use " "separate" " strings.
```

is equivalent to the statement

```cpp
    cout << "We can use separate strings.
```

Other escape sequences  Not shown in this lesson’s program are other escape sequences that are used less frequently than \n. Here is one example. Others are listed in Table 2.1.

Printing double quotes. Because double quotes are special symbols that could be misinterpreted if used alone within a string constant, we must put a backslash immediately in front of them to display them on the screen. No space is allowed between the backslash and the double quotes following it. Thus, the statement

```cpp
    cout << "Print 3 double quotes -\" \" \" \n";
```

produces the following output:

```
Print 3 double quotes   -" " "
```

List of escape sequences. Character escape sequences consist of a backslash followed by a letter, symbol, or a combination of digits. Each represents a character that has special meaning or specifies an action. (Note: Table 2.1 is a complete list of escape sequences that you can use as a reference later. At this point you do not need to understand the meanings of all of them.)
APPLICATION EXAMPLE 2.1 CREATING LOGO AND TABLE HEADINGS

Problem Statement

Neatly displayed output is essential for effectively communicating results. Properly written table headings are extremely useful because program output is frequently displayed in the form of a table. Write a program that creates a logo and two sets of table headings for a monthly bank statement. The logo is for Harrison Bank in the form:

```
HARRISON
HH H
HH H
HARRISON
HH H
HH H
HARRISON
```

The title of the document is “MONTHLY BANK STATEMENT.” The first set of table headings should consist of five columns: Account Name, Interest Rate, Number of Transactions, Beginning Balance, and Ending Balance. The number of spaces for each column is 20, 12, 12, 15, and 15, respectively. The second set of table headings should consist of three columns repeated once: Check Number, Date Paid, and Amount. The number of spaces for each column is 12, 9, and 10, respectively.
Solution

Because this particular program is not very complicated, we need not do much planning. (Beginning in Chapter 3, we follow our formal procedure for developing programs.) We can immediately start writing on paper or on the computer. In either case, you should begin writing your programs with the lines of code that you need for most of your programs. You can immediately write the following:

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

Memorize this form so you can write it within a few seconds of sitting down to write your source code!

For this program, the next step is to write the cout statements. One difficulty in this program is that the number of spaces allocated for many columns is insufficient for a one-line column heading. Therefore, many column headings must be put on two lines of the cout statements. We show the statements in the source code. Look at the source code and make sure that you understand exactly what all the symbols do. We finish the program by writing a short comment banner at the beginning of the code.
cout << "\n" "MONTHLY BANK STATEMENT \n"
"\n" "Account" "Interest" "Number of" "Beginning" "Ending" "\n"
"Name" "Rate" "Transactions" "Balance" "Balance" "\n"
"\n" "\n" "\n" "\n" "\n"; cout << "\n" "Check" "Date" "Amount" "Check" "Date" "Amount" "\n"
"Number" "Paid" "Number" "Paid" "\n" "\n" "\n" "\n"; }

**Output**

HARRISON
HH H
HH H
HARRISON
HARRISON
HH H
HH H
HARRISON

MONTHLY BANK STATEMENT

<table>
<thead>
<tr>
<th>Account Name</th>
<th>Interest Rate</th>
<th>Number of Transactions</th>
<th>Beginning Balance</th>
<th>Ending Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check Number</td>
<td>Date</td>
<td>Amount</td>
<td>Check Number</td>
<td>Date</td>
</tr>
<tr>
<td>Paid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Comments**

You can see how similar the look of the `cout` statements is to the output. To create table headings, simply make your `cout` statements resemble the desired form. If you line up the headings in the `cout`
statements, they will be lined up in the output. Also, it is easier to use separate double-quote-enclosed strings than to use the backslash string continuation symbol.

Note that, as with writing all programs, many different ways are acceptable. You could write a program using endl instead of \n, and it would still work fine. We have chosen to use \n because it is more compact, but endl is equally acceptable. At this point, any program that you write that works properly, is neat, logical, and well commented, is an acceptable program.

Modification Exercises

1. Make the logo for the company a bridge.

2. Make a third table (below the second) with three columns: Date, Transaction Type, and Amount. The column widths should be 10, 12, and 12.

3. Make a fourth table (below the third) with four columns: First Name, Middle Name, Last Name, and Tax ID Number. The column widths should be 20, 20, 20, and 15.

APPLICATION EXAMPLE 2.2 DEBUGGING

This is an important example that you should read thoroughly. It illustrates a typical session a beginner goes through to get a simple program to run. By understanding this example, you will be more confident and composed as you work to get your own programs to execute correctly. Before we give the problem statement, we make some general comments about bugs and debugging.

Bug

An error in a program is generically called a bug. Since there is no formal definition, a bug can refer to almost anything that produces a difficulty in program execution or compilation. The term originated in 1945 when a moth flew into the Mark I computer at Harvard and caused a program’s execution to terminate. The logbook recorded the “first actual case of a bug being found.” Since then, many bugs have been found in many programs.

Debugging

In your source code, looking for and correcting errors or mistakes that cause your programs to behave unexpectedly is called debugging. In general, there are three types of errors in a C++ source code: syntax errors, run-time errors, and logic errors.

Syntax errors

Syntax errors are mistakes caused by violating the “grammar” rules of C++. They easily can be caused by typographical mistakes or a lack of knowledge of the forms of statements required by C++. These errors often can be diagnosed by the C++ compiler as it compiles the program. If your compiler indicates errors when you try to compile a program, it will not translate your code
into machine instructions. You must fix the errors before the compiler will translate your code. Therefore, when you have syntax errors you will not generate any output, even if your syntax errors are very minor and located in the very last lines of code.

Run-time errors

Run-time errors, also called semantic errors or smart errors, are caused by violation of the rules during execution of your program. The compiler does not recognize them during compilation. However, the computer displays a message during execution that something has gone wrong and (usually) that execution is terminated. If a run-time error occurs near the end of execution, you may get some of your results. The error message given by the computer may help you locate the source of error in your code.

Logic errors

Logic errors are the most difficult errors to recognize and correct because the computer does not indicate that there are errors in your program as it does with syntax and run-time errors. It is up to you to identify that there is a problem at all. It is up to you to look at your output and decide that it is incorrect. In other words, your program may have appeared to have executed successfully, perhaps giving very reasonable results. However, the answers may be completely wrong. You must recognize that they are wrong and correct the code in the program. (Be careful, though. We will see, as we go further in this book, that the problem may be your input data. Many hours have been spent looking for bugs in programs only to find out that the program is correct, but the input data is incorrect.)

**Defending Against Bugs**

To reduce the number of bugs in your programs, you need to make sure that you develop good programming habits. This includes such things as

- Writing your programs neatly.
- Adding blank lines at natural locations.
- Lining up your opening and closing braces.
- Adding comments properly.

Following these steps will get you started in avoiding bugs. In essence, you should try to work in an organized and structured manner. Remember, a computer is not forgiving. Any error that you make will not be ignored by the computer. Throughout this text, we will note common errors to keep you aware of certain issues so that you can focus on these issues and avoid bugs.

**First Steps in Debugging**

If your program does not run, do not get frustrated. Realize that debugging is a part of programming. When you are debugging at this stage in your programming career, look at the whole program, and ask yourself:
Did I type anything wrong? For example, was cout typed cut?
- Did I use and follow C++ punctuation properly? For example, int main () being typed int main ();.
- Are my parentheses and braces in pairs?

In other words, look for obvious things first— the common errors. Then, like looking for problems with your car when it is not working, identify the performance problem and use it as a guide to find the source. For instance, if your car’s windshield wipers do not work, you do not look for the source in the rear of the car. Instead, you look at the windshield wiper motor and wiring. With a computer program, if a certain calculation is not performing correctly you look at the portion of code where that calculation is performed and the portions of code that connect to that calculation.

Although the C++ compiler can help you find some errors, do not expect it to pinpoint all the errors and give you instructions on how to correct them. For example, you may just miss the closing "*/" in one of your multiline comments at the beginning of your program. If this is the case, the C++ compiler may think the rest of your program is simply a part of an unfinished comment. This minor error may generate 30 error messages. Do not be alarmed. Remember that the typical C++ compiler is not particularly sophisticated in identifying syntax errors. You may be able to eliminate 100 error messages by modifying a single character in your source code.

Problem Statement

This problem is meant to illustrate the debugging that is needed in the process of developing a required program. Here, the goal is to create a program that can print the following to the screen:

Envision that you have written the following program as a first attempt to print this statement (note that we have included line numbers for this program so that we can reference them in the description of the errors):

```
DEBUGGING EXAMPLE:

This is an example of debugging. By following the step-by-step procedure that we describe in this application example, you will begin to develop the skills you need for successful and efficient debugging.
```

Line number

```
1 #<include iostream>;
2 using namespace std;
3 int main ( );
4 {
5     cout << 'DEBUGGING EXAMPLE:';
6     cout << "This is an example of debugging.\n;
7     By following the step-by-step procedure";
8     cout << "that we describe in this application example,
```
you will begin to develop the;
cout << "skills you need for successful and efficient debugging.";
}

Use the compiler to help you modify the code. Follow the step-by-step procedure outlined in the following methodology.

**Methodology**

First, we outline the steps involved in debugging. Then, we use the steps to guide us to debug the preceding program. The procedure we describe is not the only one that can be used. Your instructor may have other suggestions that you will find helpful. However, at this point in your programming career, follow these steps to correct your program errors:

1. Compile your source code.

2. Look at the location in your source code that is indicated by the first error message. At this point, do not try to understand what the error message means. Only the location indicated is important to us right now.

3. Examine approximately five statements at this location—two or three statements above the indicated location, the statement at the location, and two or three statements below the indicated location.

4. Use this text and your reference books to check your syntax (that is, “grammar, notation, punctuation, and form”) for all the lines in the region indicated by step 3.

5. Correct the syntax error(s) you see. At this point, do not attempt to correct your program for every error the C++ compiler has printed out. Fix only the first one.

6. Repeat steps 1–5. Note that each time you compile your program you will likely get a set of error messages that are completely different from the ones the compiler printed previously. This is one reason why we fix only one error at a time. Remember, just one error may cause 100 error messages to be printed. There is no need to read and try to interpret all 100 messages! Fix only the first one and then recompile your program.

You may need to repeat steps 1–5, even 10 or 15 times. Do not get discouraged, this is not unusual. Eventually, you will fix all these errors (which are syntax errors) indicated by your compiler, and your program will begin executing when you try to run it. However, even after you have fixed all the syntax errors, your program still may have run-time or logic errors.

7. Now that all the syntax errors are corrected, execute your program again and look at the output. Does the output have statements printed that are not like any you have used in your cout statements? You may see such words as overflow or execution terminated. If you have these, then these are run-time errors. From this message, you may get an idea of the location in your program where the run-time error occurred. Also, your program may have printed some of your
cout statements before the run-time error is printed. This means that the run-time error is located after these cout statements in your program. Go back to your source code and look at the statements in the region indicated by the run-time error. Correcting run-time errors is somewhat similar to correcting logic errors. We describe correcting both of these in step 8.

8. You can identify logic errors in your program by noticing that the output is not what you want or expect. In other words, suppose you were expecting the output

This is my output.

but the program printed

This is my output.

From this you know which cout statement has an error, and you must go back to the code and modify it.

Sometimes the location and cause of the error is not as obvious as in this example. You often can get an idea of the location of the error, though, by looking at your output. For instance, suppose your program has 10 cout statements in it. If your program produces errors after the first five have been printed correctly, then the error is located in your source code after the fifth cout statement.

As we did for syntax errors, look at five or six statements in the region where you feel that the error has occurred. Do not look at just one statement, even if you are sure that execution stopped at that statement. At this point you should ask yourself, “Why is the program not doing what I am trying to tell it to do?” If you cannot see the necessary correction initially, you can make small adjustments to the statements in the error region and see the effect on the output. You should make adjustments and rerun the program repeatedly. This is how you get experience programming—change something and see its effect. It is a major part of the learning process. Do not get discouraged by the computer telling you that you have made errors. Do not make changes blindly, though. Think about what you are doing and what effect you expect the changes will have. Throughout this book, we discuss techniques for helping recognize the source of logic errors and the changes to correct them.

9. After running, making changes, and rerunning your program repeatedly you will have developed a working program. Congratulations! However, before you put this program in your program library, make sure that it is well commented so that you can understand it later or others can easily interpret what you have done. If you have not commented it well, go back at this point and put in comments before you forget what you have done, and then create whatever other documentation is necessary. Resist the temptation to quit and celebrate. You will thank yourself later for spending a relatively small amount of time at this point to properly document your program. A summary of debugging is presented in Fig. 2.3.
Debugging the Given Program

1. We begin by compiling the program in the problem statement. Each compiler gives slightly different error messages, so your compiler may not give exactly what we show here. When we compile the program, our compiler lists eight error messages. The first message points to line 1 of the code and says “Unknown preprocessor directive.” As stated in the methodology, we will not worry about what this means but simply look at the line indicated and a few lines above and below for an error. Since line 1 is the first line of code, we focus on this line. When we do this, we see that we typed:

```cpp
#include iostream;
```

when we should have typed:

```cpp
#include <iostream>
```

Note that we had two errors in this statement: the `<` was in the wrong location, and we had a semicolon at the end of the line. At this point we ignore the other seven errors, correct just this one line and recompile the program. Also, refer to Table 2.2, Table 2.3 and Fig. 2.4 for a summary of all the errors we describe here.
### TABLE 2.2 — Syntax error correction summary

<table>
<thead>
<tr>
<th>Step</th>
<th>Total number of errors indicated by compiler</th>
<th>First line to which the compiler points indicating a syntax error</th>
<th>First error message</th>
<th>First line with a syntax error</th>
<th>Action required to correct the syntax error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>1</td>
<td>Unknown preprocessor directive</td>
<td>1</td>
<td>change <code>&lt;include iostream&gt;;' to </code>#include &lt;iostream&gt;`</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5</td>
<td>Character constant too long</td>
<td>3</td>
<td>change int main( ) to int main( )</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>5</td>
<td>Comma expected</td>
<td>4</td>
<td>change ( to )</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>5</td>
<td>Character constant too long in function main</td>
<td>5</td>
<td>change <code>DEBUGGING EXAMPLE;' to </code>DEBUGGING EXAMPLE;' to debugging.</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>6</td>
<td>Unterminated string or character constant in function main</td>
<td>6</td>
<td>change debugging.; to debugging. ;</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>Unterminated string or character constant in function main</td>
<td>8</td>
<td>change example, to example, &quot;</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>8</td>
<td>Undefined symbol 'you' in function main</td>
<td>9</td>
<td>change you to 'you'</td>
</tr>
</tbody>
</table>

*Note: The error messages are dependent upon the IDE used.*

### TABLE 2.3 — Logic error correction summary

<table>
<thead>
<tr>
<th>Step</th>
<th>Indication of error</th>
<th>Action required to correct the logic error</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Output spacing is incorrect</td>
<td>add \n after EXAMPLE: after procedure after develop the</td>
</tr>
<tr>
<td>9</td>
<td>Output spacing is incorrect</td>
<td>1. add another \n after EXAMPLE: &lt;br&gt; 2. replace \ after debugging. \ with &quot; &lt;br&gt; 3. add &quot; before by</td>
</tr>
</tbody>
</table>

![Figure 2.4](#)  
Summary of syntax errors in Application Example 2.2.
2. On compiling the next time, we get an error indicated at the first cout statement (line 5) with the message, “) expected”, which is read “Right parenthesis expected.” Again, we do not try to determine why this message is being given. Instead, we look a few lines above our location and see that we typed:

    int main ( );

when we should not have put a semicolon at the end of this line. Thus, we remove the semicolon and recompile the program. Note that the compiler indicated the error was at line 5, but the error was really at line 3.

3. We now recompile the program and get one warning and seven errors. Warnings are not fatal errors. In other words, your program can execute correctly and completely with many warnings being given by the compiler. Warnings are exactly those: messages from the compiler to the programmer to indicate that something unusual is happening, and an error may be caused by this. While you are still trying to locate syntax errors in your programs, we recommend that you ignore warnings and focus on the errors. (It also may be helpful to read the first warning. If you can understand it and it indicates a simple problem, you can correct it quickly. Otherwise, pass it by. When your program begins to execute, read and evaluate all the warning messages given.)

    The first error message is “, expected”, which is read “Comma expected.” The location indicated is again the first cout statement in the program (line 5). Again, we look above this location and see that line 4 has a left parenthesis, {, instead of a left brace, {. So we change the parenthesis to a brace and recompile the program.

4. On recompiling we find that the number of errors is not reduced, instead it increases to 17 errors! We should not be alarmed by this. Instead, we simply work with the first message that reads, “Character constant too long in function main.” The location indicated by this message is once more the first cout statement (line 5). Since we thought we fixed the errors prior to this statement, we look at this statement and realize that we have used single quotes and not double quotes. We, therefore, change the single quotes to double quotes and recompile the program.

5. Our compiler now gives 16 error messages. The first one is indicated by the compiler to be on the second cout statement (line 6) in the program and says, “Unterminated string or character constant in function main.” We look at this cout statement and see that we should not have used a semicolon at the end of the line. So we remove it and recompile.

6. We now get seven error messages. The first one points to the third cout statement (line 8) and says, “Unterminated string or character constant in function main.” At this point we can describe some of the rationale behind the error messages. We note that the written description of the error for step 5 is the same as that for step 6. Our string constants are enclosed in double quotes. The error message says that we have an “unterminated string constant.” Remember, spacing is significant in string constants. The compiler says that because it is looking for a second
pair of double quotes or a continuation character, \, on the same line as our third cout statement and does not find it, it thinks the string is unterminated. Of course, we meant to terminate it on the next line, but this does not work. Therefore, we add " to the end of the third cout line (line 8) to terminate the string.

Note that step 5 gave us the same error message. However, in that case our action was to remove a semicolon rather than add double quotes. C++ interpreted the semicolon to mean the end of the line. Since no matching double quote was at the end of the line, C++ believed that the string was unterminated. As you get more experience, you will be able to make some use of the actual error messages. However, even experienced programmers often find the messages not particularly helpful because they do not necessarily point out the immediate problem. Rather than spend time trying to interpret the message, it is frequently more efficient to simply focus on the syntax in the region of the error.

We now recompile the program.

7. We get six errors with the first being, “Undefined symbol ‘you’ in function main” at line 9. We realize that this line was meant to be part of the cout string constant, and we put quotes at the start and end of this line. We recompile the program.

8. We get no errors and therefore attempt to run the program. The program executes completely! However, the output is

DEBUGGING EXAMPLE: This is an example of debugging a program. By following the step-by-step procedure that we describe in this application example, you will begin to develop the skills you need for successful and efficient debugging.

Our program executes but is incorrect because the spacing is wrong. Because the compiler sent no error message, we find that we do not have any run-time errors in this program. However, we have logic errors that now must be addressed. At this point, the compiler can no longer aid us in finding or correcting our errors. We must do it completely on our own by looking at the source code and the output and figuring out what is incorrect and changing it. In your own programs, if you are completely stumped at this point, before you ask for help, make small changes in your code and recompile and run the program. By doing this, you will understand the effect changes make, and it may inspire you to find the source of your error.

In this example, we must get the spacing correct. We add \n after “EXAMPLE:”, after procedure, and after develop the. We make these changes from the knowledge we developed reading this chapter.

9. After having made the changes indicated in step 8 and rerunning the program we find that the resulting output is

DEBUGGING EXAMPLE: This is an example of debugging. By following the step-by-step procedure that we describe in this application example, you will begin to develop the skills you need for successful and efficient debugging.
This is still not quite correct because we want double space after “DEBUGGING EXAMPLE:” and because we have too much space before the word By. So, we add another \n after “DEBUGGING EXAMPLE:”. Eliminating the space before By is more difficult. We have extra space because the continuation character, \, on the cout line above By causes the indentation in the source code to be interpreted as extra space. Therefore, we choose to eliminate the continuation character, replace it with double quotes, and put double quotes before By.

10. After having made these changes, we get the correct result when we execute the program. However, we are not finished because we must comment our program. Had our program been longer, we would have written comments earlier. However, for this short program now is a good time to write comments. Our end product is

```cpp
/******************
* Name: Application 2.2
* Purpose: Learning debugging skills.
* Date: Written on 3/7/2002
* Author: John Jones
* Reference: None
******************
#include <iostream>
using namespace std;
int main ()
{
  cout << "DEBUGGING EXAMPLE:

  This is an example of debugging.
  "By following the step-by-step procedure\n"
  "that we describe in this application example,"
  "you will begin to develop the\n"
  "skills you need for successful and efficient debugging.\n"
}
```

11. We run our program after commenting it to make sure that it still works. If this were a larger program, we would need to write supplementary documentation as well. However, for this program we are now finished.

Comments

You should notice the following from Tables 2.2 and 2.3:

1. The line indicated by the compiler may not be the line with the error. You must look above and below that line to find the error.

2. The error messages given by the compiler may or may not be helpful. As you gain experience, you will be able to use the error messages more effectively.

3. When you recompile the program after having corrected an error, the number of error messages indicated by the compiler may actually increase. This does not mean that the program is becoming more incorrect.
4. The compiler does not indicate logic errors. You must observe that logic errors have occurred on your own. Then you must correct them without help from the compiler.

Even though the procedure we have described here is valuable, we cannot guarantee that if you follow it you will definitely eliminate all your errors. For instance, the method does not work if you were to simply forget the terminating `/` on a multiline comment. Try doing this. You will find that the compiler indicates an error on the very last line of the program when the error is really located much earlier! Therefore, looking only a few lines above and below the indicated error location would not uncover the error. Although the method we presented does not work in this situation, we believe that you will find the method helpful for locating many of your programming errors. As we go further in this book, we describe other techniques for recognizing and correcting errors. The method described here simply is to get you started in the process of debugging.

We urge you to learn the debugging feature of your IDE. Right now you may have difficulty understanding it, but, by the end of Chapter 3, you should have made an effort at learning how to use it. See your IDE’s documentation or ask your instructor about how to use it. Learning to use the debugger will save you many hours of programming frustration. We will not discuss it more as each IDE’s debugger is different. We simply encourage you to learn this device on your own.

Also, be as independent as you can be, and be selective in your attempts at getting help. In general, try not to rely on others to debug your programs for you. Start by trying to work through any problems on your own. After you have made an effort to solve your own problem and still have not solved it, seek help. As painful as it may be, you will learn the most and become a better programmer by solving problems relying just on your books and computer.
Chapter 3 Variables and Arithmetic Operations

Chapter Topics
In this chapter, you will learn how to:
- Work with variables
- Perform arithmetic calculations
- Use mathematical functions

To get your programs to solve practical problems, you need to know how to create and manipulate variables. The variables can be used in math functions like sin(), cos(), and log() (that are built into C++) as well as with ordinary operators such as +. However, C++ has strict rules on how arithmetic operations are carried out. For instance, in an arithmetic statement involving addition and multiplication, the multiplication is performed first. This chapter describes these types of rules and illustrates how to create programs that solve some basic problems.

LESSON 3.1 VARIABLES (1): NAMING, DECLARING, ASSIGNING, AND PRINTING VALUES

TOPICS
- Naming variables
- Declaring variables
- Displaying variable values
- Elementary assignment statements

Variables are crucial to virtually all C++ programs. You have learned about variables in algebra, and you will find that in C++ variables are used in much the same manner. Suppose, for instance, you want to calculate the area of many triangles, all of different sizes. And suppose that the given information is

1. The length of each of the three sides.
2. The size of each of the three angles.

To write an algebraic equation to determine the area, you need to make up your own variable names. You might choose as variable names

1. Lengths: \(a, b, c\)
2. Angles: \(\alpha, \beta, \gamma\)

Or you could name the variables

1. Lengths: \(l_1, l_2, l_3\)
2. Angles: \(\theta_1, \theta_2, \theta_3\)

Or you could name the variables something completely different. It is entirely up to you what to name them, and you most likely would choose variable names that are descriptive to you and others or conventionally used in a particular context.
For programming in C++, the situation is quite similar. You choose the variable names, and it is best to choose names that are descriptive to you and others or conventionally used in a particular context. A major difference between typical C++ programs and typical algebraic expressions is that the variables in most algebraic expressions consist of just one or two characters, maybe with a subscript or superscript. Variables in C++ programs often consist of entire words rather than single characters. Why? Because, as you will find, programs can get quite long, and there simply are not enough single characters to represent all the necessary variables. Also, you will find that it will be easier for you and others to understand your programs if you have given very descriptive names to each variable.

For instance, for the triangle area program you may use the variable names

1. **Lengths:** length1, length2, length3
2. **Angles:** angle1, angle2, angle3

Or, if you wanted to be even more descriptive, you could name your variables

1. **Lengths:** side_length1, side_length2, side_length3
2. **Lengths:** SideLength1, SideLength2, SideLength3
3. **Angles:** angle_opposite_side1, angle_opposite_side2, angle_opposite_side3
4. **Angles:** AngleOppositeSide1, AngleOppositeSide2, AngleOppositeSide3

These variable names are much less ambiguous than their algebraic counterparts. Unfortunately, expressions using these variable names look much more cumbersome than the ones using simple algebraic notation. However, this is a disadvantage with which we simply must live. Check with your instructor or employer regarding the preferred form for naming variables. Some organizations use forms 1 and 3 while others use forms 2 and 4.

C++ has rules you must follow in dealing with variables. For instance, you must declare all your variable names in your programs, which means to list your variables and indicate what type they are. Variables can be integers or reals (as well as other types) and must be declared to be of the desired type in your programs. Also, older C++ compilers do not allow you to use more than 31 characters for one variable name, while current standard C++ compilers have no practical limit on the number of characters. These and other rules will be discussed in the lesson.

We can give numeric values to the variables using assignment statements. Assignment statements have a single variable name followed by an = sign and an expression. The = symbol is called the assignment operator. You may use multiple assignment operators in a single statement, for example

```
angle1 = angle2 = angle3 = 0;
```

The program for this lesson works with variables associated with an industrial facility. At this facility are a number of tanks. Each tank has an identification number (variable tank_id), diameter (variable diameter) and pressure (variable pressure). The program assigns numeric values to these variables and prints them out along with text. Read the source code and pay particular attention to the declarations, assignment statements, and the cout statements used to print the variable values.
**Source Code**

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

int main ()
{
    int tank_id;  // Declaring variable (tank_id) to be an integer using the keyword int.
    double diameter, pressure;  // Declaring variables (diameter and pressure) to be real using the keyword double.

    tank_id=12;  // Assignment statements using the = sign store numerical values in the memory cells for the declared variables.
    diameter = 111.1;
    pressure = 100.0;

    cout << "Tank_id:" << tank_id << " , Diameter:" << diameter << endl;  // Variable names (without double quotes) after insertion operators in cout statements cause the variable values to be printed.

    tank_id=11;  // Assignment statements cause the values previously stored in variable memory cells to be overwritten.
    diameter = 82.1;

    cout << "Tank_id:" << tank_id << " , Diameter:" << diameter << " , Pressure=" << pressure << "\n";  // This cout statement prints the values of the variables tank_id, diameter, and pressure after the last two assignment statements have been executed.
}
```

**Output**

```
Tank_id=12, Diameter=111.1
Tank_id=11, Diameter=82.1, Pressure=100
```

**Description**

**Actions in memory as a program executes.** As a program executes, conceptually, a table is created internally. This table contains variable names, types, addresses, and values. The names, types, and addresses are first established during compilation; then as execution takes place space is reserved in memory, and the variable values are put into the memory cells reserved for the variables at the locations indicated by the addresses. In this lesson’s program, we declared three variables and after assigning values to them, the table looks like:

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Variable type</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tank_id</td>
<td>int</td>
<td>12</td>
</tr>
<tr>
<td>diameter</td>
<td>double</td>
<td>111.1</td>
</tr>
<tr>
<td>pressure</td>
<td>double</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Later in the program, we assign new values to the variables and the table changes. It becomes

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Variable type</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tank_id</td>
<td>int</td>
<td>11</td>
</tr>
</tbody>
</table>
Comparing the two tables, the variable values do change. You will find that your programs continually change the variable values in the table. Next, we describe how to properly set up such a table by naming, declaring, and assigning values to variables.

**Naming variables.** Variables in C++ programs are identified by name. Remember, you choose the variable names for your programs. Variable names are classified as identifiers. Therefore, when naming variables you must obey the rules used for identifiers. For instance, the first character of an identifier cannot be numeric, and the other characters must be a–z, A–Z, _, or 0–9. Table 3.1 lists the constraints on creating valid identifiers. Read this table to understand the rules for identifiers.

### Table 3.1 — Some constraints on identifiers

<table>
<thead>
<tr>
<th>Topic</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>The first character in identifier</td>
<td>Must be letters (a–z, A–Z) or _</td>
</tr>
<tr>
<td>Other characters in identifier</td>
<td>Must be letters (a–z, A–Z) _, or digits 0–9</td>
</tr>
<tr>
<td>The maximum number of characters</td>
<td>Standard C++ does not restrict the number of characters; however older compilers have a limit of 31 (which is more than enough for most programs)</td>
</tr>
<tr>
<td>in an internal identifier (i.e., identifier within a function)</td>
<td></td>
</tr>
<tr>
<td>Use of all uppercase characters</td>
<td>Allowed; however, many programmers use all uppercase for constant names.</td>
</tr>
<tr>
<td>Use of mixed cases or underscores</td>
<td>Allowed and commonly used when variables need long descriptive names.</td>
</tr>
<tr>
<td>Use of C++ keywords (also called reserved words) and alternative representations of certain operators and punctuators as identifiers</td>
<td>Not allowed; do not use these as identifiers:</td>
</tr>
<tr>
<td></td>
<td><code>asm</code> <code>bitand</code> <code>bitor</code></td>
</tr>
<tr>
<td></td>
<td><code>auto</code> <code>break</code> <code>case</code></td>
</tr>
<tr>
<td></td>
<td><code>bool</code> <code>catch</code> <code>class</code></td>
</tr>
<tr>
<td></td>
<td><code>char</code> <code>const</code> <code>continue</code></td>
</tr>
<tr>
<td></td>
<td><code>compl</code> <code>not</code> <code>not_eq</code></td>
</tr>
<tr>
<td></td>
<td><code>const_cast</code> <code>delete</code> <code>dynamic_cast</code></td>
</tr>
<tr>
<td></td>
<td><code>default</code> <code>do</code> <code>double</code></td>
</tr>
<tr>
<td></td>
<td><code>else</code> <code>enum</code> <code>extern</code></td>
</tr>
<tr>
<td></td>
<td><code>explicit</code> <code>false</code> <code>friend</code></td>
</tr>
<tr>
<td></td>
<td><code>float</code> <code>for</code> <code>goto</code></td>
</tr>
<tr>
<td></td>
<td><code>if</code> <code>int</code> <code>long</code></td>
</tr>
<tr>
<td></td>
<td><code>inline</code> <code>mutable</code> <code>namespace</code></td>
</tr>
<tr>
<td></td>
<td><code>new</code> <code>operator</code> <code>private</code></td>
</tr>
<tr>
<td></td>
<td><code>or</code> <code>or_eq</code> <code>xor</code></td>
</tr>
<tr>
<td></td>
<td><code>protected</code> <code>public</code> <code>reinterpret_cast</code></td>
</tr>
<tr>
<td></td>
<td><code>register</code> <code>return</code> <code>short</code></td>
</tr>
<tr>
<td></td>
<td><code>signed</code> <code>sizeof</code> <code>static</code></td>
</tr>
<tr>
<td></td>
<td><code>static_cast</code> <code>template</code> <code>this</code></td>
</tr>
<tr>
<td></td>
<td><code>struct</code> <code>switch</code> <code>typedef</code></td>
</tr>
<tr>
<td></td>
<td><code>throw</code> <code>true</code> <code>try</code></td>
</tr>
<tr>
<td></td>
<td><code>typeid</code> <code>typename</code> <code>using</code></td>
</tr>
<tr>
<td></td>
<td><code>union</code> <code>unsigned</code> <code>void</code></td>
</tr>
<tr>
<td></td>
<td><code>virtual</code> <code>wchar_t</code> <code>and</code></td>
</tr>
<tr>
<td></td>
<td><code>volatile</code> <code>while</code> <code>and_eq</code></td>
</tr>
<tr>
<td>Use of standard identifiers such as <code>main</code></td>
<td>Standard identifiers, such as <code>main</code>, can be used as variable names.</td>
</tr>
<tr>
<td></td>
<td>However, their use is not recommended because it leads to confusion.</td>
</tr>
<tr>
<td>Use of blank within an identifier</td>
<td>Not allowed, because an identifier is a token.</td>
</tr>
</tbody>
</table>
Here are some examples of illegal variable names:

- 1apple (illegal because it begins with a number)
- interest_rate% (illegal because it has % in it)
- float (illegal because it is a keyword)
- Income (illegal because it has a space)
- one'two (illegal because it has a quote in it)

and examples of legal variable names:

- apple1
- interest_rate
- xfloat
- Income
- one_two

**Keywords.** A keyword is an identifier type token for which C++ has a defined purpose (Table 3.1). Keywords used in this lesson’s program are using, namespace, int, and double. Because these have special meanings in C++, we cannot use them as variable names.

You do not need to memorize the keywords in this table; however, it is very important that you recognize the keywords in a program you read. If you do not know the keywords, you may confuse a keyword with a variable name, function name, or other identifier. As you read this book you will learn how to use many of the keywords listed in Table 3.1, and they will naturally become part of your C++ vocabulary.

**Declaring variables.** Variable names in C++ must be declared. The statement

```cpp
int tank_id;
```

declares the variable `tank_id` to be of the int (which must be typed in lowercase letters and means integer) type.

Variables of the same type may be declared in the same statement. However, each must be separated from the others by a comma; for example, the statement

```cpp
double diameter, pressure;
```

declares the variables `diameter` and `pressure` to be of the double (which must be typed in lower case) type (see Fig. 3.1).

![Diagram](image-url)
The type double is one of C++’s real number data types. Variables declared to be double have their values stored using the binary format. Variables declared to be double can hold values that contain digits after the decimal point.

Declaring variables causes the C++ compiler to know that space is to be reserved in memory for storing the values of the variables. By stating a variable’s type, the C++ compiler knows how much space in memory is to be set aside. Although not explicitly set by the C++ standard, the standard implies (because C++ is a superset of C) the minimum number of bits to be used for each variable type. For instance, C++ requires that type int be capable of storing at least a range of values between -32768 and 32767. This requires 16 bits or 2 bytes of memory. Therefore, declaring the variable tank_id as int indicates that at least 16 bits or 2 bytes of memory is reserved for this variable’s value. However, for 32-bit computer systems, int variables could require 32 bits or 4 bytes of memory with a range between -2147483648 and 2147483647. On the other hand, a double type value occupies a minimum of 8 bytes or 64 bits with a range between $\pm 2.22507 \times 10^{-308}$ and $\pm 1.79769 \times 10^{308}$. Therefore, declaring a variable to be a double requires at least 8 bytes of memory to be reserved. You may use the sizeof operator to determine the size in bytes of a variable, for example:

```cpp
    cout << sizeof (double);
```

C++ uses different types of binary codes for integers and real numbers. This means that, for example, the bit pattern for 87 stored as an int is completely different from the bit pattern for storing 87.0 as a double. It is important that we keep this in mind. Forgetting this fact will lead to errors as we will see when we begin working more with variables.

**Assignment statements and using variables.** An assignment statement assigns a value to a variable, which means an assignment statement causes a value to be stored in the variable’s memory cell. For example, the statement

```cpp
    tank_id = 12;
```

assigns the integer value 12 to int type variable tank_id. It causes 12, written in two’s complement binary type of notation, to be stored in the tank_id memory cell (Fig. 3.2).

![Figure 3.2 Assignment statement.](image)

In C++, a simple assignment statement takes the form of

```cpp
    variable_name = value;
```

where this statement stores the value on the right side of the equal sign in the memory cell reserved for the variable_name on the left side of the equal sign. The binary representation of the value is stored in the variable’s memory cell after the assignment has taken place. The value can be a constant, a variable
with a known value, or other, such as a function or an expression that returns a value (see the next few lessons for more details). Note that the equal sign in an assignment statement does not really mean equal. It simply indicates that a value is to be stored in the variable’s memory cells.

The following forms are not allowed:

```cpp
value = variable_name;
```

or

```cpp
12 = tank_id;
```

Although you learned in algebra that x = 5 and 5 = x are equivalent, C++ does not allow a value to appear on the left side of an assignment statement. Also, a single variable name should appear on the left.

Variables must be declared before they are assigned values! In other words, we could not have written the beginning of this lesson’s program:

```cpp
#include <iostream>
using namespace std;
int main ()
{
    tank_id = 12 ;
    diameter = 111.1;
    pressure = 100. ;
}
```

because the variables were assigned values before being declared. Remember, C++ processes the statements one after another. If you try to assign a value to a variable before declaring it, no memory exists for that variable, and therefore no value can be placed in memory. An error will be indicated during compilation if you either do not declare a variable or declare a variable after it has been used.

C++ does not require that variable declarations be the first lines of the body of the program or function. In other words, we could have written the beginning of this lesson’s program as:

```cpp
#include <iostream>
using namespace std;
int main ()
{
    int tank_id;
    double diameter, pressure;
    tank_id = 12 ;
    diameter = 111.1;
    pressure = 100. ;
```
because each variable is declared before it is used. In most places in this textbook, we will declare our variables at the beginning of each function. You may see some programmers declaring their variables immediately before they are used. You should check with your instructor or employer on which method is preferred. We recommend that you follow a method consistently throughout your programs so that others can easily follow your code.

You may also see a line of the type:

```cpp
double diameter = 111.1;
```

This both declares the variable diameter to be type double and initializes it (that is, gives the variable its first value) to be 111.1. You may also use the function notation to initialize a variable:

```cpp
double diameter(111.1);
```

Under the C++11 standard, brace initializer could also be used:

```cpp
double diameter = {111.1};
```

or

```cpp
double diameter {111.1};
```

It is also possible to let the compiler to decide the type of variables, for example:

```cpp
auto diameter = 111.1;
```

The type of the variable diameter will be double as 111.1 is of the type double. If you forget to initialize a variable that is defined inside a function, the variable’s value is indeterminate. However, such a line does not need to appear at the beginning of a function. When we study functions more thoroughly in Chapter 7, we will see that there are other rules for when we declare variables outside the body of any function.

### Displaying the value of a variable or constant on the screen.

The cout object can be used to display the value of a variable or constant on the screen. The syntax is

```cpp
cout << variable_name;
```

where variable_name is a variable that has been declared and assigned a value. This simple form will display the variable value only and contain no explanatory text. To add explanatory text and advance the cursor to the next line, the following form can be used:

```cpp
cout << "explanatory_text" << variable_name << "explanatory_text" << endl;
```

where explanatory_text is any text information as described in Chapter 2. Only one variable_name can be used between stream insertion operators (<<).
LESSON 3.2  VARIABLES (2): CREATING CONSTANT QUALIFIED VARIABLES AND MORE ABOUT PRINTING VARIABLE VALUES

TOPICS

- Creating constant qualified variables with the const qualifier.
- Setting the after decimal point precision for printing.
- Controlling the space used for printing.
- Scientific notation.
- Left justifying values.

You will find sometimes that you need to use values in your programs that do not change. For instance, we know that \( \pi \) is approximately 3.14159. For a program that involves areas of circles, it is convenient to simply write the characters PI in the equations rather than the digits 3.14159. This can be done by using the const qualifier in the declaration. Once the const qualifier has been used in declaring a variable, the value of the variable cannot be modified later in the program.

Sometimes you will want to display your values in special ways rather than in the C++ default manner. For instance, to display a table neatly you may want to use 20 spaces for each value no matter how many digits a number may have. C++ allows you to set the width in your cout statements. Also, you may want to display fewer digits after the decimal point than a number actually has. This is particularly important when dealing with dollar calculations where only two digits after the decimal should be displayed. In this lesson’s program we show how to perform these manipulations with cout statements.

When working with very large or very small numbers, scientific notation is convenient. For example, to represent 57,650,000, the scientific notation would be \( 5.765 \times 10^7 \), which in C++ would be 5.765e+007 or 5.765E+007. In other words, we can use scientific notation in our source code. For instance, we could use the following declaration and assignment statement in a program:

```cpp
   double x;
   x = 5.63e+009;
```

C++ accepts and properly interprets this notation in source code. We can also display scientific notation in our output as we show in this program. By specifying scientific notation in our cout statements, C++ decides the value of the exponent, and thus it is possible to display an extremely large number in a small number of spaces. The programmer need decide only on the number of significant digits to display. This program creates one constant qualified variable, PI. It prints the numerical value of this variable in a number of formats. Read the program and notations and look at the output.
Constant qualified variables. We use the const qualifier (const is a keyword) and assign a value in the declaration (which is required) to create constant qualified variables. For example, the line

```
const double PI = 3.14159;
```

assigns the value of 3.14159 to PI and qualifies it so that it cannot be modified later in the program. An error would have been generated during compilation if we had used an assignment statement later in the program of the sort:

```
PI = 3.1415926;
```
because C++ does not allow us to use a constant qualified variable on the left side of an assignment statement after it has been initialized.

We will find a number of uses for the const qualifier. The const qualifier helps us write better code because it can assure us that we do not accidentally modify something that should not be modified. As we go through the text, we will point out the uses of the const qualifier when appropriate. For now, we will create constant qualified variables when we use such things as conversion factors (for instance, the number of centimeters per inch is 2.54) or fixed numbers (for instance, the number of days in a week is 7).

Many C++ programmers use all uppercase characters to name constant qualified variables (as in this lesson’s program with PI). Lowercase characters are used to name ordinary variables (with underscores and some uppercase characters possibly mixed in). This book follows this convention in most cases. The value of following this convention is that it is easier for others to understand your code and for you to understand others’ code. Check with your employer or instructor if another convention is to be used. For integer type, enum keyword could be used to create a list of named constants, for example:

```
enum {Zero, One, Two, Three};
```

where Zero is 0, One is 1, Two is 2 and Three is 3. The constants may not be in consecutive order, for example:

```
enum {Zero, Thousand = 1000, Hundred = 100};
```

You may find some books still using the #define preprocessor directive instead of the const keyword. However, it is not a good practice and should be avoided.

**Formatting output using manipulators.** In the program for Lesson 3.1, the C++ compiler decided how many digits would be displayed for a number on output and how much space a number would occupy. There will be times when you will want to decide how a number will be displayed rather than leaving it to the compiler (for instance, when you want to print a table neatly). To do this, we insert I/O manipulators called parameterized manipulators, into cout statements that we use to print variable values. The parameterized manipulators are declared in the header iomanip. You must include this header with the statement:

```
#include <iomanip>
```

to use parameterized manipulators in your cout statements. In its most basic form, a parameterized manipulator can be inserted into a cout statement like this:

```
cout << manipulator(parameter);
```

where manipulator is any standard C++ parameterized manipulator and parameter is what the manipulator uses to modify output. We can also use parameterized manipulators between << operators in our cout statements as we have done in this lesson’s program. Table 3.2 lists the C++ I/O
parameterized manipulators. The most important of these at this time are used in this lesson’s program and are described here.

**TABLE 3.2 — The C++ I/O parameterized manipulators**

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Action</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setfill(int f)</code></td>
<td>Set the fill character to <code>f</code></td>
<td><code>setfill('*')</code></td>
</tr>
<tr>
<td><code>setprecision(int p)</code></td>
<td>Set the precision of a floating-point number to <code>p</code></td>
<td><code>setprecision(2)</code></td>
</tr>
<tr>
<td><code>setw(int w)</code></td>
<td>Set the field width to <code>w</code></td>
<td><code>setw(20)</code></td>
</tr>
<tr>
<td><code>setiosflags(long f)</code></td>
<td>Set the format flag specified by <code>f</code></td>
<td><code>setiosflags(ios::left)</code></td>
</tr>
<tr>
<td><code>resetiosflags(long f)</code></td>
<td>Return to default for the flag specified by <code>f</code></td>
<td><code>resetiosflags(ios::left)</code></td>
</tr>
<tr>
<td><code>setbase(int b)</code></td>
<td>Set the base of numbers output to be <code>b</code> (must be 8, 10, or 16)</td>
<td><code>setbase(16)</code></td>
</tr>
</tbody>
</table>

- `setw`. We can make empty space appear on the left side of a displayed number by setting the field width (that is, the number of text spaces that a printed number occupies) much larger than the number needs by inserting the parameterized manipulator `setw` into the `cout` statement. For example, the statement

  ```cpp
cout << "2. PI=\[" << setw(15) << PI << "]\]" << endl;
```

uses the manipulator `setw` to change the field width to 15. The output for this lesson’s program shows that this statement prints:

```
2. PI=[[    3.14159]]
```

which illustrates 15 spaces between the double brackets. Observe that the number is right justified within the 15 spaces, meaning that the number is pushed to the right edge of the field. C++ does this automatically.

If the `cout` statement does not specify the field width, as we did in Lesson 3.1, C++ decides how large the field width should be. When integers are printed, C++ makes the field width equal to the number of digits in the integer, meaning that no empty text space is displayed on either side of the integer. However, if we use `setw(4)` to print the number 549382, C++ will auto-matically expand the width to accommodate all the digits.

For instance, if we have the statements:

```cpp
i = 549382;
cout << setw(4) << i;
```

the output will be:

```
549382
```

not 5493. In other words, you need not worry about accidentally specifying the width to be smaller than what is actually required.
- **setprecision.** The manipulator *setprecision()* sets the number of digits after the decimal point to be printed. For instance, in this lesson’s program, we use *setprecision* with the line:

```cpp
cout << "3. PI = [" << setprecision(2) << PI << "]" << endl ;
```

This causes the output to show only two digits after the decimal point for PI, making the output:

```
3. PI = [3.14]
```

Even though only two digits are displayed, all the digits are retained in memory so that later in the program one can use *setprecision(4)* and still have four digits correctly printed.

Once the precision is set in a *cout* statement, this precision may or may not be used until another statement changes it. It depends on your compiler as standard C++ does not require that all compilers work this way. Some compilers may need the precision to be specified with each *cout* statement (or the default will be used). Check the manual of your compiler to see how *setprecision* is implemented. For our compiler, in this lesson’s program, the statement:

```cpp
cout << setprecision(4);
```

sets the precision to 4 for all the *cout* statements after it in the program.

- **setfill.** The manipulator *setfill()* specifies the character for filling blank space in a field. For instance:

```cpp
cout << "4. PI=[" << setw(20) << setfill('*') << PI << "]" << endl;
```

creates a field width of 20 (with *setw(20)*) and fills the portion of the field not containing digits with the character * (with *setfill('*')).* Observe that single quotes (not double quotes) are required around the character enclosed in parentheses. The output from this statement is:

```
4. PI=[***************3.14]
```

Note that the precision is 2 (two digits after the decimal point) even though the precision is not specified in this *cout* statement. Because the precision was previously specified to be 2, *cout* (with the compiler we have used) retains that precision until a new precision is specified. Depending on your compiler, this filler may appear until changed by another *cout* statement.

- **setiosflags.** The manipulator *setiosflags* can perform a number of different actions. In this lesson’s program, we use it to left justify the value of PI by using *setiosflags(ios::left)* in the line:
cout <<"5. PI=[[" <<setiosflags(ios::left) <<setw(20) <<PI <<"]"] <<endl;

which produces the output:

5. PI=[[3.14***************]]

For the time being, simply remember to use ios followed by two colons and then left to left justify. (Actually, left is considered to be a flag of the ios class. For now, you do not need to know the details of what this means.) Observe that we used setw(20) to make the field width 20. Unlike setprecision(), we need setw() for each cout statement if the field width is to be something other than the default.

In this lesson’s program, we used setiosflags(ios::scientific) to display a number in scientific notation. Look at the output from the program to see the effect of this.

We can use setiosflags() with more than one flag by combining the flags with the symbol ‘|’. For example, the manipulator setiosflags() in the statement:

cout << "7. PI=[[" << setiosflags(ios::left | ios::scientific) << setw(20) << PI << "]"]" << endl;

uses the | symbol to combine the left and scientific flags to produce the output:

7. PI=[[3.1416e+00**********]]

You can use other flags to perform other tasks. Other flags and what they do are shown in Table 3.3. You do not need to understand all of this table at this time. We will highlight what is important as we go along. You can use this table later as a reference when you do more advanced programming.
Some stream manipulators do not require parameters. In fact, `endl` is classified as a standard manipulator that does not require parameters. Table 3.4 lists other standard manipulators and what they do. Again, you do not need to know and understand these at this time. We simply present them here for later reference.
Printing in “dollar” format. To print output in “dollar” format (meaning two digits after a decimal point), it is necessary to use I/O manipulators. For instance, the statement

\[
\text{cout} \ll \text{setprecision}(2) \ll \text{setiosflags(ios:fixed | ios:showpoint)} \ll \text{"Income=$"} \ll \text{income};
\]

uses the manipulator setprecision(2) to indicate two digits after the decimal point, and setiosflags(ios:fixed | ios:showpoint) to assure the decimal point will be displayed in fixed notation with the decimal point shown. For the variable income having the value 7842, the output produced by this cout statement is:

\[
\text{Income = $7842.00}
\]

Apart from using manipulators, you may use formatting member functions of cout to achieve the same objective. For example:

\[
\text{cout.precison}(2); \text{cout} \ll 1.23;
\]

could replace

\[
\text{cout} \ll \text{setprecision}(2) \ll 1.23;
\]

while “cout << setw(20) " could be replaced by “cout.width(20)".

LESSON 3.3 VARIABLES (3): ASSIGNING AND PRINTING SINGLE CHARACTER DATA

TOPICS
- The set of characters
- Single character output

The single character type, char, can be utilized somewhat similarly to the numeric types, int and double. The term character refers to more than just lowercase and uppercase letters. There are graphic characters such as ‘!’, ‘#’, and ‘^’. Even ‘space’ (hitting the space bar) is considered a character. Escape sequences (like ‘\n’ and ‘\r’) also are regarded as single characters. The numbers ‘0’–’9’ can be treated as characters as well. We may find this helpful when we want to store such things as telephone numbers. Telephone numbers have no arithmetic manipulations on them (it makes no sense to add two of them
Because of this, a telephone number may be more difficult to manipulate if it is stored as an integer. Therefore, we may find it convenient to store numbers that do not require arithmetic operations as characters.

This program assigns and prints characters. We use the static_cast operator to print the decimal integer equivalent of each of the characters and show the effect of using characters in addition statements. Read the code and annotations.
Declaring character variables. Character variables are declared with the keyword char. The form is

```cpp
char variable1, variable2, variable3, ...;
```

For example, the declaration in this lesson’s program:

```cpp
char c1, c2, c3, c4, c5, c6;
```

declares c1, c2, c3, c4, c5, and c6 to be of character type.

Assigning characters to character variables. To assign a character constant to a character variable, it is necessary to enclose the constant in single quotes, "(not double quotes, "). For instance, the assignment statement

```cpp
c1 = 'g';
```

assigns the symbol g to variable c1. Using single quotes, we can assign a “less than” symbol with:

```cpp
c2 = '<';
```

and treat the number 7 as a symbol with:

```cpp
c4 = '7';
```

Because the escape sequences are treated as a single symbol, the assignment statement

```cpp
c3 = '\n';
```

makes c3 represent “newline.” When we print these out with cout statements, the 'g', '<', and '7' symbols are printed out directly. The “newline” character does not appear; it simply causes the cursor to advance to the next line. The output from this lesson’s program illustrates this. If we choose to print a different escape sequence such as tab ('\t'), a tab takes place with no symbol printed out. Other escape sequences behave similarly.

If we assign a number to a char variable without using single quotes as we did with the statement

```cpp
c5 = 63;
```

we are actually assigning the symbol with the American Standard Code for Information Interchange ASCII (for most personal computers) code decimal equivalent value of 63. From the list of ASCII code decimal equivalent values shown in Table 3.5, we see that this represents the symbol '?'. Therefore, when we print out c5 using cout as we did in this lesson’s program, we print the '?’ symbol.

Only computers that use the ASCII encoding scheme (which is most personal computers) will interpret 63 as meaning '?’. Another encoding scheme, EBCDIC (not in our table) interprets 63 as being a nonprintable control character. Thus, different computers give different results. To make your programs portable from computer to computer, we recommend you replace “c5 = 63;” with “c5 = '?';”. (Note: We shall restrict our discussion in this text to ASCII.)
If we add or subtract from a character, we get a different ASCII code value and a different symbol. For instance, the statement
\[ c6 = c1 - 3; \]
subtracts 3 from the ASCII code value for \( c1 \) (which is 103 for the symbol ‘g’) and arrives at 100. Thus, this statement is equivalent to
\[ c6 = 'd'; \]
When we print \( c6 \), we print ‘d’ as shown in this lesson’s output.

**Assigning characters to int variables.** If we assign a character to an int variable, C++ assigns the ASCII code value for the char to the int. For instance, the statement

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII decimal equivalent</th>
<th>Character</th>
<th>ASCII decimal equivalent</th>
<th>Character</th>
<th>ASCII decimal equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\a</code></td>
<td>7</td>
<td><code>&lt;</code></td>
<td>60</td>
<td><code>-</code></td>
<td>95</td>
</tr>
<tr>
<td><code>\b</code></td>
<td>8</td>
<td><code>&gt;</code></td>
<td>61</td>
<td><code>a</code></td>
<td>96</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>9</td>
<td><code>?</code></td>
<td>62</td>
<td><code>b</code></td>
<td>97</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>10</td>
<td><code>A</code></td>
<td>63</td>
<td><code>c</code></td>
<td>98</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>11</td>
<td><code>B</code></td>
<td>64</td>
<td><code>d</code></td>
<td>99</td>
</tr>
<tr>
<td><code>\f</code></td>
<td>12</td>
<td><code>C</code></td>
<td>65</td>
<td><code>e</code></td>
<td>100</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>13</td>
<td><code>D</code></td>
<td>66</td>
<td><code>f</code></td>
<td>101</td>
</tr>
<tr>
<td><code>space</code></td>
<td>32</td>
<td><code>E</code></td>
<td>67</td>
<td><code>g</code></td>
<td>102</td>
</tr>
<tr>
<td><code>i</code></td>
<td>33</td>
<td><code>F</code></td>
<td>68</td>
<td><code>h</code></td>
<td>103</td>
</tr>
<tr>
<td><code>*</code></td>
<td>34</td>
<td><code>G</code></td>
<td>69</td>
<td><code>i</code></td>
<td>104</td>
</tr>
<tr>
<td><code>#</code></td>
<td>35</td>
<td><code>H</code></td>
<td>70</td>
<td><code>j</code></td>
<td>105</td>
</tr>
<tr>
<td><code>%</code></td>
<td>36</td>
<td><code>I</code></td>
<td>71</td>
<td><code>k</code></td>
<td>106</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>37</td>
<td><code>J</code></td>
<td>72</td>
<td><code>l</code></td>
<td>107</td>
</tr>
<tr>
<td><code>;</code></td>
<td>38</td>
<td><code>K</code></td>
<td>73</td>
<td><code>m</code></td>
<td>108</td>
</tr>
<tr>
<td><code>{</code></td>
<td>39</td>
<td><code>L</code></td>
<td>74</td>
<td><code>n</code></td>
<td>109</td>
</tr>
<tr>
<td><code>}</code></td>
<td>40</td>
<td><code>M</code></td>
<td>75</td>
<td><code>o</code></td>
<td>110</td>
</tr>
<tr>
<td><code>+</code></td>
<td>41</td>
<td><code>N</code></td>
<td>76</td>
<td><code>p</code></td>
<td>111</td>
</tr>
<tr>
<td><code>,</code></td>
<td>42</td>
<td><code>O</code></td>
<td>77</td>
<td><code>q</code></td>
<td>112</td>
</tr>
<tr>
<td><code>-</code></td>
<td>43</td>
<td><code>P</code></td>
<td>78</td>
<td><code>r</code></td>
<td>113</td>
</tr>
<tr>
<td><code>.</code></td>
<td>44</td>
<td><code>Q</code></td>
<td>79</td>
<td><code>s</code></td>
<td>114</td>
</tr>
<tr>
<td><code>/</code></td>
<td>45</td>
<td><code>R</code></td>
<td>80</td>
<td><code>t</code></td>
<td>115</td>
</tr>
<tr>
<td><code>0</code></td>
<td>46</td>
<td><code>S</code></td>
<td>81</td>
<td><code>u</code></td>
<td>116</td>
</tr>
<tr>
<td><code>1</code></td>
<td>47</td>
<td><code>T</code></td>
<td>82</td>
<td><code>v</code></td>
<td>117</td>
</tr>
<tr>
<td><code>2</code></td>
<td>48</td>
<td><code>U</code></td>
<td>83</td>
<td><code>w</code></td>
<td>118</td>
</tr>
<tr>
<td><code>3</code></td>
<td>49</td>
<td><code>V</code></td>
<td>84</td>
<td><code>x</code></td>
<td>119</td>
</tr>
<tr>
<td><code>4</code></td>
<td>50</td>
<td><code>W</code></td>
<td>85</td>
<td><code>y</code></td>
<td>120</td>
</tr>
<tr>
<td><code>5</code></td>
<td>51</td>
<td><code>X</code></td>
<td>86</td>
<td><code>z</code></td>
<td>121</td>
</tr>
<tr>
<td><code>6</code></td>
<td>52</td>
<td><code>Y</code></td>
<td>87</td>
<td><code>{</code></td>
<td>122</td>
</tr>
<tr>
<td><code>7</code></td>
<td>53</td>
<td><code>Z</code></td>
<td>88</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>8</code></td>
<td>54</td>
<td><code>\</code></td>
<td>89</td>
<td><code>}</code></td>
<td>124</td>
</tr>
<tr>
<td><code>9</code></td>
<td>55</td>
<td><code>;</code></td>
<td>90</td>
<td><code>:</code></td>
<td>125</td>
</tr>
<tr>
<td><code>;</code></td>
<td>56</td>
<td><code>:</code></td>
<td>91</td>
<td><code>;</code></td>
<td>126</td>
</tr>
<tr>
<td><code>:</code></td>
<td>57</td>
<td><code>:</code></td>
<td>92</td>
<td><code>;</code></td>
<td>127</td>
</tr>
<tr>
<td><code>:</code></td>
<td>58</td>
<td><code>:</code></td>
<td>93</td>
<td><code>;</code></td>
<td>128</td>
</tr>
<tr>
<td><code>;</code></td>
<td>59</td>
<td><code>:</code></td>
<td>94</td>
<td><code>;</code></td>
<td>129</td>
</tr>
</tbody>
</table>
\( i = c4; \)

In this lesson's program assigns the ASCII code value for the symbol '7' (which is 00110111 in binary and this interprets to 55 in decimal as shown in Table 3.5) to \( i \). Note that it does not assign the value 7! In fact, if we take \( c4 + 2 \) as we did in the assignment statement

\( j = c4 + 2; \)

we get 57 not 9. This also is shown in this lesson's output. Remember, when we treat numbers as characters, C++ works with the ASCII code value not with the numeric value. One benefit of working with characters this way is that we can sort or alphabetize since ASCII code values increase by one for each letter progressed in the alphabet (see Table 3.5). Using techniques described later, we can arrange characters in alphabetic order using this property.

LESSON 3.4 ARITHMETIC OPERATIONS (1): ARITHMETIC OPERATORS AND EXPRESSIONS

TOPICS
- Utilizing variables
- Operands
- Arithmetic operators and their properties
- Arithmetic expressions
- Exceeding the integer range
- Overflow errors

Arithmetic expressions in C++ look much like algebraic expressions that you write. In this program we show how to use expressions to perform arithmetic operations, save the results in memory, and avoid errors. We show the different C++ arithmetic operators and what they mean. The program is separated into two sections with cout statements indicating the section. Different arithmetic operations are performed in each section.

To conserve memory, instead of using int type, “short int” or simply short type could be used. “short int” variable represents a range of values from -32768 to 32767. You might wonder what happens if we attempt to assign a value outside this range. In this program, we make such an assignment and print the assigned value in the output.

Sometimes you will attempt to execute your programs, and you will get an abnormal termination (meaning that the program terminates before each statement is executed properly). You may get a message that uses the word overflow when this happens. You will need to trace the source of the overflow error and modify your code if this occurs. In this program we do not show code that produces an overflow error, but in the description we discuss a common cause of overflow problems and how to correct them.
Read the program and annotations, and pay special attention to the operators used in the assignment statements. Note that in this and the next few lessons, we use one-letter variable names. We do this simply because we are only demonstrating arithmetic operations, not writing real programs. In the application examples, we use descriptive variable names. Remember, when writing your programs make sure your variable names are long enough to convey the meaning of what they represent.
Description

**Utilizing variables.** Before we utilize a variable by putting it on the right side of an assignment statement, we must initialize it. In this lesson’s program, the variables i, j, k, p, x, and y are initialized in the first six assignment statements.

\[
\begin{align*}
i &= 5; \\
j &= 5; \\
k &= 11; \\
p &= 3; \\
x &= 3.0; \\
y &= 4.0; \\
\end{align*}
\]

After this they are put on the right side of the next assignment statements, for example,

\[
\begin{align*}
a &= x + y; \\
i &= i + 1; \\
j &= j + 1; \\
\end{align*}
\]

Notice that you can take your hand calculator (or use your head) to calculate a numerical value for each arithmetic expression shown here on the right side of the = sign because all the variables have been initialized. For all your programs, no matter how complex, you can (although it may be very tedious and time consuming) use your hand calculator or do hand arithmetic and perform the same computations that your source code instructs the computer to do. In theory, you could go from the beginning to the end of the program line by line using your hand calculator and arrive at the same results as the program.

Remember, your programs execute one line at a time in order from the top of your code to the bottom (although we will see in Chapter 5 that we can modify this). You must make sure that each variable that appears on the right side of an assignment statement has been given the correct numerical value in a line of code that has been previously executed!

**Arithmetic expressions, operands, and operators.** An arithmetic expression is a formula for computing a value. For example, the expression \( x + y \) computes \( x \) plus \( y \), \( x - y \) computes \( x \) minus \( y \), \( x \times y \) computes \( x \) times \( y \), and \( x / y \) computes \( x \) divided by \( y \). Often, arithmetic expressions appear on the right side of assignment statements.
An arithmetic expression consists of a sequence of operand(s) and operator(s) that specify the computation of a value. For example, the expression, \(-x + y\), consists of two operands \(x\) and \(y\) and two operators \(+\) and \(-\).

An operand can be a variable, such as \(x\) or \(y\), or a constant, such as 3.1416, or anything that represents a value, such as a function (see Chapter 7 for details).

The operator ++ is an increment operator, which can be placed before or after (but not both) a variable. The operator increases the value of the variable by 1. For example, assuming a variable \(i\) is equal to 1, then after the statement

\[
i++;\]

or

\[
++i;
\]
is executed, the value of \(i\) is 2 (\(i++\) is not exactly the same as \(++i\), see Lesson 3.5 for details). Note that the above C++ statements can be understood as the statement

\[
i = i + 1;
\]

which also causes the value of the variable \(i\) to increase by 1. Similarly, the operator -- is a decrement operator, which decreases the value of a variable by 1. The statement

\[
i--;
\]
or

\[
--i;
\]
can be understood as the statement

\[
i = i - 1;
\]

The operator \(\%\) is a remainder or modulus operator, meaning that it calculates the remainder of a division operation. The \(\%\) operator must be placed between two integer variables or constants. Assuming \(k\) and \(p\) are two integer variables, the meaning of \(k \% p\) is the remainder of \(k\) divided by \(p\). For example, if \(k = 11\) and \(p = 3\), then \(k \% p\) is equivalent to 11 \(\%\) 3, which is equal to 2 (because 3 goes into 11 three times with a remainder of 2). The operator \(\%\) is pronounced “mod.” So this example would be \(k \mod p\). Standard C++ states that if either operand is negative, the sign of the result of the \(\%\) operation is implementation defined; that is, it is free for the C++ compiler designer to decide. For example, depending on the compiler you use, the results of \(-50 \% 6\) and \(50 \% (-6)\) may be 2 or -2.

**Arithmetic expressions and assignment statements.** An arithmetic expression is not a complete C++ statement, but only a component of a statement. The value obtained from the expression may be stored in a variable using an assignment statement. For example, the arithmetic expression \(x / y\) is part of the C++ assignment statement
\[ d = \frac{x}{y}; \]

The statement assigns the value obtained from the arithmetic expression on the right to the variable on the left. Thus, the assignment statement

\[ i = i + 1; \]

although not looking correct algebraically, is a valid C++ assignment statement. The arithmetic expression \( i + 1 \) creates a new value that is 1 greater than the current value of \( i \). The assignment statement then gives \( i \) this new value.

Note that we cannot write these two assignment statements as

\[
\begin{align*}
    x / y &= d; \\
i + 1 &= i;
\end{align*}
\]

because on the left side of assignment statements we can have only single variables, not expressions. Single variables are allowed to be lvalues (pronounced ell-values), meaning they are allowed to be on the left side of assignment statements. Expressions are rvalues (pronounced are-values) because they are allowed on the right side of assignment statements.

A single variable located alone on the right side of an assignment statement is considered an expression. We will see other times when a single variable is considered an expression.

**Uninitialized variables.** Note that if an uninitialized variable is used in an expression on the right side of an assignment statement no error may be indicated at compile time! The C++ compiler is incapable of recognizing this type of error. This means that if the programmer does not observe the error, the program will be executed with an error.

In fact, a program may execute completely without a run-time error even if uninitialized variables appear on the right side of assignment statements! However, the results will be incorrect and may lead to run-time error if you are using Microsoft Visual Studio. It is up to you, the programmer, to recognize that the results are incorrect and to change the program. Sometimes a program will be obviously incorrect and can be easily modified by a programmer. At other times it is not so obvious, and it takes an astute programmer to detect the error. The moral is that you must look closely at your results and test your programs thoroughly before concluding that they are correct.

Just to demonstrate the effect of not initializing a variable, in this lesson’s program we have not initialized the variable \( z \), used the assignment statement

\[ x = y + z; \]

and printed out the value of \( x \) with a succeeding cout statement. The value of \( x \) is shown in the output line

\[ \text{With } z \text{ not initialized, } x = y + z = 97.23903 \]
Observe that x makes no sense and has nothing to do with the values used in this program, yet the program ran without indicating an error! The variable z was assigned a value of 93.23903 by the computer since we did not assign a value to z ourselves. Without a close examination of the output, one might think that the printed value of x is correct.

Why was z assigned a value of 93.23903 by the computer? It was not deliberate. Remember, because z is declared a double, 8 bytes (minimum) is reserved in memory for z. The address of this memory is set during compilation, and during-execution this memory is reserved. However, when the memory is reserved, the bits are not set to specific values. When we use z on the right side of an assignment statement, the computer simply reads the memory cells at the address reserved for z. Whatever is there is read and interpreted using the binary scheme for double type variables. Other computer activities may have used the same memory cells, and what is left behind is being read when z is used. It just so happened on this particular execution of the program that bits that are interpreted to be 93.23903 were left in the memory cells for z. This program may have a different number for z each time it is executed since it is dependent upon activities of the computer before the program is run. In fact, if you run this program you will almost certainly not get $x = 97.23903$. You will get whatever is in the memory cells for z plus 4.0 (which is the value assigned to y). Clearly, this type of behavior for a program is unacceptable.

It is not unusual that a programmer forgets to initialize a variable before using it. You will likely do it during your programming. You must be able to recognize that you have not initialized a variable, so you can correct it. One hint that you may have made this mistake is that sometimes bits interpreted to be very large numbers are stored in the memory cells for uninitialized variables. Therefore, if you see numbers printed out that are displayed in scientific notation with large exponents, look for uninitialized variables in your program. Also, if you get different results with the same input each time you run your program, an uninitialized variable may be the problem. We cannot give a method that will always work for recognizing this type of error in your programs. You simply must be meticulous, look at your output carefully, and test your programs thoroughly to assure that your programs are reliable.

**Exceeding integer range.** We noted in our description of the “short int” type that the range of possible integer values is -32768 to 32767 due to the use of two bytes of memory for an integer (for the minimum allowable C++ implementation). In this lesson’s program we have shown what happens if an integer variable is assigned a value outside this range by assigning a value of 32770 to i. Without going into the details of the values of each bit, we show how you can determine the assigned value when the range is exceeded in Fig. 3.3. It operates somewhat like a clock. At the top of the clock is 0, the right side has the range of positive values, and the left side has the range of the negative values. The large negative and positive numbers meet at the bottom of the clock.
displays program, than incorrect. The range of overflow value is
4294967295. If you use a very large number, the computer may
close an error. The number of possible integer values is
18446744073709551615. But if you exceed the allowed integer value,
it is up to you to recognize that the program is faulty and to
correct it. The computer may not help you at all with this type of
error. Remember, you must be careful not to exceed the integer
range, and you need to look closely at your results in all cases
due to many types of errors that the computer will not detect.

Division by zero and overflow errors. If we accidentally attempt to divide a number by zero in a
program, in general, a run-time error and termination of execution will take place. The computer
displays an error message that commonly uses the word overflow. So, unlike the uninitialized variable
problem and exceeding the integer range, division by zero is recognized by the computer. The word
overflow is used because division by 0 or a number close to 0 produces a very large number. The
number, too large to store in the allocated memory, causes an interpretation of an overflow problem.

If you get an overflow error in your program, you need to use some of the debugging techniques
described in Chapter 2 to find the statement(s) that are the source of the problem, and this may not be
simple (Table 3.6 summarizes this process). For instance, say we have in a program:

\[ b = c / (x - y); \]
with x and y being equal values before this statement is executed. This statement will cause a run-time error on execution (because the denominator is zero). However, by simply looking at the code, it is not possible to immediately say that this statement is the source of the problem because nothing is inherently wrong with the statement itself.

Therefore, in your programs, if you get an overflow error message caused by division by zero, the first thing you need to do is to find the statement in your program at which the division by 0 has taken place. To do this, observe from the output which cout statements have already executed before the error message saying overflow is printed. This tells you that the division by 0 was caused by a statement executed after the cout statements were executed. This is a first step in finding the problem statement.

<table>
<thead>
<tr>
<th>TABLE 3.6 — Finding the source of an overflow error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Look at your output. You know the error has been caused by a statement after the last cout statement that produced the last line of the output.</td>
</tr>
<tr>
<td>2. If no cout statement has been executed prior to the overflow error, add cout statements to your program for the sole purpose of helping you find the location of the error.</td>
</tr>
<tr>
<td>3. Rerun the program (even though you know that the program will crash).</td>
</tr>
<tr>
<td>4. Use the cout statements executed to locate the error within a few lines.</td>
</tr>
<tr>
<td>5. Look at the source code in the region of the lines causing the error.</td>
</tr>
<tr>
<td>6. Add more cout statements to print out the values of the variables in these statements.</td>
</tr>
<tr>
<td>7. Rerun the program.</td>
</tr>
<tr>
<td>8. Use your hand calculator to check the equations used in your program, so you can see exactly which statement caused an overflow.</td>
</tr>
<tr>
<td>8. At this point, we cannot tell you exactly how to modify your program to correct the error. However, performing these steps should have given you enough insight into your problem to make it easier for you to recognize errors in your equations or other methods for avoiding overflow errors.</td>
</tr>
</tbody>
</table>

If no cout statements were executed prior to the error occurring, then one method for finding the statement with the error is simply to put cout statements in the source code at a number of locations for the sole purpose of finding the error causing location. For instance, statements such as

```cpp
    cout << " Execution has taken place to statement 5 \n";
    cout << " Execution has taken place to statement 10 \n";
```

can be repeated throughout your program. After writing a large number of these, you can execute your program again (even though you know that it will get an overflow error). On rerunning it, you see which cout statements were executed. If you have spread these throughout your program, then you know that the division by 0 has been caused by a statement shortly after the last cout statement executed.

Once the approximate location of the statement causing the error is known, examine your source code and focus on the statements in the region following the last correct cout statement. Look for divisions that have taken place, paying particular attention to such things as the variables in denominators. Take all the variables and write a cout statement to have their values printed out. Put that cout statement immediately after the last cout statement executed on the previous run.

Rerun your program again, even though you know that it will still encounter an overflow error. Look at the values of the variables printed out. Use your calculator to perform hand calculations with these variables. Look at your source code and the equations you wrote in the region of the error. By plugging
the numbers in your calculator, you should find that one of the denominators in this region calculates to be 0 or nearly 0. It may not work out to be exactly 0 because the computer may be working with numbers slightly different from those that are printed out (one reason is that the computer may be working with 30 digits when you have printed only 6 and use only 6 in your calculator). By doing this you have now narrowed the problem down to one statement, and you should understand exactly how the computer is beginning to divide by 0.

The method for correcting the problem depends on the overall purpose of your computer program. However, in many cases, you will find that the process you have gone through to find the source of the error has made you aware of an error you have made in programming. For instance, maybe you used a subtraction sign where you should have used an addition sign, or you put the wrong variable in the denominator. Recognizing these sorts of things and making the appropriate changes normally will solve your problem. However, you need to address this on a case-by-case basis. An easier way to locate the error is to learn how to use your compiler’s debugger (see the manual). It will save you from writing all the cout statements. However, if you do not learn the debugger, you should become proficient enough to write all the cout statements that you need quickly and easily.

LESSON 3.5 ARITHMETIC OPERATIONS (2): MIXED TYPE ARITHMETIC, COMPOUND ASSIGNMENT, OPERATOR PRECEDENCE, AND TYPE CASTING

TOPICS

- Precedence of arithmetic operations
- Pitfalls in arithmetic statements
- Mixing integers and reals in arithmetic expressions
- Type casting
- Side effects

When we use numbers in our source code, the way we write the numbers influences how C++ treats the numbers. For instance, if we write the number 4 (without a decimal point), C++ treats the number as an int. If we write the number 4.0 (with a decimal point) C++ treats the number as a double. As we see in this program, we must be aware of the types we use when performing arithmetic calculations. This program is broken into four sections separated by cout statements. They illustrate:

Section 1 The difference between using ++ before and after a variable.
Section 2 The effect of using both double and int variables in a single arithmetic expression, and using integers in division operations.
Section 3 The compound operators.
Section 4 The precedence of arithmetic operators (meaning which operator is executed first).

As you read the source code, use your calculator to compute the values of the variables on the right sides of the assignment statements.
```cpp
#include <iostream>
using namespace std;

int main()
{
    int i = 1, j = 1,
        k1 = 10, k2 = 20, k3 = 30, k4 = 40, k5 = 50,
        k, h, m, n;

double a = 7.0, b = 6.0, c = 5.0, d = 4.0,
        e, p, q, x, y, z;

cout << "************ Section 1 ****************" << endl;
    cout << "Before increment, i=" << i << ", j=" << j << endl;
    k = i++;       // Pre-increment operator.
    h = ++j;       // Post-increment operator.

    cout << "After increment, i=" << i << "", j=" << j
                 << ", k=" << k << "", h=" << h << endl << endl;

    m = 6.0 / 4.0;
    p = 6.0 / 4.0;
    n = 6.0 / 4.0;
    q = 6.0 / 4.0;

cout << "m=" << m << ", p=" << p << ", m=" << m << ", q=" << q << endl << endl;

cout << "************ Section 2 ****************" << endl;

cout << "Original k1=" << k1 << ", k2=" << k2 << ", k3=" << k3
                 << ", k4=" << k4 << ", k5=" << k5 << endl;

    k1 += 2;
    k2 -= 1;
    k3 *= (8.0/4.0);
    k4 /= 2.0;
    k5 %= 2;

cout << "New k1=" << k1 << ", k2=" << k2 << ", k3=" << k3 << ", k4=" << k4
                 << ", k5=" << k5 << endl << endl;

cout << "************ Section 4 ****************" << endl;

e = 3;
    x = a + b - c / d * e;
    y = a + (b - c) / d * e;
    z = ((a + b) - c / d) * e;

cout << "a=" << a << ", b=" << b << ", c=" << c << ", d=" << d << ", e=" << e
                 << endl << endl;

    cout << "x=" << a + b - c / d * e << endl
                 << "y=" << a + (b - c) / d * e << endl
                 << "z=" << ((a + b) - c / d) * e << endl;
}
```
Pre- and post-increment operators used in assignment statements. In this lesson’s program, we initialized both i and j to 1 and used the assignment statements:

\[ k = i++; \]
\[ h = ++j; \]

These statements illustrate the difference in the post-increment (as in i++) and pre-increment (as in ++j) operators when used in assignment statements. In the first statement, the value of i is assigned to the variable k before the value of i is incremented. After the assignment, the variable i is incremented by the post-increment operator ++ from 1 to 2. Therefore, after executing

\[ k = i++; \]
\[ i = 2 \text{ and } k = 1. \]

In other words, the statement “k = i++;” is equivalent to the statements

\[ k = i; \]
\[ i = i + 1; \]

However, for “h = ++j;”, the value of j is incremented by the pre-increment operator ++ from 1 to 2. After the increment, the new j value, which is 2, is assigned to the variable h. Therefore, after executing

\[ h = ++j; \]
\[ j = 2 \text{ and } h = 2. \]

In other words, the statement “h = ++j;” is equivalent to statements

\[ j = j + 1; \]
\[ h = j; \]

The decrement operator (--) works similarly. These are the rules for such operators:
1. If the increment or decrement operator precedes the variable, the variable is first incremented; then the assignment takes place.
2. If the increment or decrement operator follows the variable, the assignment takes place first; then the variable is incremented or decremented.

You must memorize these two rules. When more complex expressions are used with the ++ and -- operators, the same rules apply. For instance:

\[
\begin{align*}
i & = 2; \\
j & = 3 \times (i++) - 2;
\end{align*}
\]

gives \( j \) a value of 4, whereas

\[
\begin{align*}
i & = 2; \\
j & = 3 \times (++i) - 2;
\end{align*}
\]

gives \( j \) a value of 7 because in the first case the expression is evaluated with \( i = 2 \) (before the increment) and in the second case the expression is evaluated with \( i = 3 \) (after the increment).

In this text, we normally use the ++ and -- operators in lone standing statements because, as you can see, some confusion is caused when they are used in more complex statements. We recommend that you do the same. However, you still need to know the rules because you may see them used in complex statements in programs written by others.

**Same type and mixed type arithmetic.** The calculation of 6.0/4.0 is considered to be same type arithmetic since both these values are real (both 6.0 and 4.0 have a decimal point as shown). The result is 1.5. This operation is illustrated schematically at the top left of Fig. 3.4. Because both operands are real (like the double type), the result is real.

By contrast, when we calculate 6/4, we have an integer divided by an integer (because neither 6 nor 4 have a decimal point). When this occurs, if both operands are negative or positive, the fractional part of the quotient is discarded. Therefore, using C++, 6/4 is not equal to 1.5. It is equal to 1! This is illustrated at the top right of Fig. 3.4.

Using a negative integer in an integer division operation (such as -6/5) produces a result that is implementation defined. This means that standard C++ has no hard and fast rule; therefore, the result depends on the compiler you use. Check your compiler; for -6/5 it should give either -1 or -2 as a result. Note that the “correct” answer is -1.2. The compiler therefore has a choice of rounding up or down.

It is considered mixed type arithmetic when we calculate 6/4.0. For this operation, one operand is a real type and the other operand is an integer type. When this occurs, C++ converts the integer to a real type temporarily (meaning that 6 is -converted to 6.0), then performs the operation, and the result is a real type. Thus, 6/4.0 gives the result 1.5. This procedure is illustrated at the bottom of Fig. 3.4.
Assigning real values to int variables.  If we try to assign a real type value to a variable that has been declared to be int, C++ cuts off the fractional part of the real value, converts the remaining part to int, and stores the result in two’s complement binary form in the memory cells for the variable. For instance, the statement in this lesson’s program

\[ n = 6 / 4.0; \]

takes the real value (6=4.0, which is 1.5), cuts off the 0.5 to give 1.0, converts this to 1 (without a decimal point, meaning it is in two’s complement binary form), and stores this in the memory cell for the int variable n. The assignment statements for m, p, and q are described in Fig. 3.5.

Assigning int values to real variables.  When assigning int values to real variables, C++ puts a decimal point at the end, converts the method of storage to exponential binary form, and stores the result in the variable’s memory cell. Thus,

\[ p = 6 / 4; \]

takes the integer value (6/4, which becomes 1 due to the cutting off of the fractional part), converts this to 1.0 (with a decimal point meaning it is in exponential binary form), and stores this in the memory cell for the variable p.
Figure 3.5 Same type and mixed type arithmetic operations. Assigning real values to int variables and integer values to double variables. Read the right column of boxes first and the left column second.

Modifying the way C++ uses types in arithmetic operations. We have not shown it in this lesson’s program, but C++ has cast operators. Cast operators can change the type of an expression (recall that a single variable can be regarded as an expression). Thus, we can use cast operators on the right side of an assignment statement to modify the type of an arithmetic expression’s result. Here, we describe the static_cast operator. It consists of the keyword static_cast followed by a type enclosed in angle brackets. The operand should be enclosed in parentheses.

For instance, if we have declared the variables

```c
int aa = 5, bb = 2, cc;
double xx, yy = 12.3, zz = 18.8;
```

then the static_cast operator can be used in the following way:

```c
xx = static_cast < double > (aa) / static_cast < double > (bb);
cc = static_cast < int > (yy) + static_cast < int > (zz);
```
To understand how the operations in these statements are to take place, realize that in performing arithmetic operations C++ makes copies of the variable values and works with the copies. After the operation has been completed using the copies, a final result for the expression is obtained. If the expression is on the right side of an assignment statement, the assignment then takes place.

Thus, the static_cast operator causes the copy of aa to be 5.0, bb to be 2.0, yy to be 12, and zz to be 18. Table 3.7 summarizes these actions. Because the operations are performed with the copies, we can clearly see that the values stored for the variables on the left sides of the assignment statements are

\[ xx = 5.0 / 2.0 = 2.5 \]
\[ cc = 12 + 18 = 30 \]

Had the program statements been written without the cast operators, the results would have been

Thus, we can see that the static_cast operator has changed the results stored for the variables xx and cc.

The general form for use of the static_cast operator is

\[ \text{static_cast < type > (expression)} \]

where the keyword static_cast uses the underscore symbol, and expression is the expression for which a temporary copy is to be made of type type. The type can be any valid C++ data type. We will learn of more valid C++ data types later in this text.

Note that the use of static_cast will make such things as the right sides of assignment statements look very complicated. Do not be deterred from using static_cast because of this. Use static_cast whenever it is necessary to get the correct numerical results.

Also, C++ has older forms of cast operators that look less complicated. For instance

\[ (\text{double}) \text{ aa} \]

or

\[ \text{double (aa)} \]
While such forms are simpler, the static_cast form is recommended.

Compound assignment operators. The operators +=, -=, *=, /=, and %= are compound assignment operators. Each performs an arithmetic operation and an -assignment operation. These operators require two operands: The left operand must be a variable; the right one can be a constant, a variable, or an arithmetic expression. In general, the two operands can be of integer or real data type. However, the %= operator requires that its two operands be of integer type.

For instance, the meaning of

```
k1 += 2;
```

(not k1 += 2;) can be understood to be similar to the statement

```
k1 = k1 + 2;
```

If the original value of k1 is equal to 20, the new value will be 20 + 2 or 22. Similarly, these statements also are valid if we replace the arithmetic operator + with -operator -, *, /, or %. For example,

```
k1 *= 2;
```

is similar to

```
k1 = k1 * 2;
```

With k1 initially 20, the new value for k1 is 40.

Summary of arithmetic operators. Table 3.8 shows the operators along with their properties that can be used in an arithmetic expression. The number of operands is the number of operands required by an operator. A binary operator, such as /, requires two operands while a unary operator, such as ++, needs only one. Fig. 3.6 shows the concepts of these two types of operators.

![Figure 3.6 Unary and binary operators.](image)

The position is the location of an operator with respect to its operands. For a unary operator, its position is prefix if the operator is placed before its operand and postfix if it is placed after its operand; for a binary operator, the position is infix because it is always placed between its two operands. For example, the negation operator in -x is prefix, the post-increment operator in y++ is postfix, and the remainder operator in a % b is infix.
The associativity specifies the direction of evaluation of the operators with the same precedence. For example, the operators + and - have the same level of precedence and both associate from left to right, so 1 + 2 - 3 is evaluated in the order of (1 + 2) - 3 rather than 1 + (2 - 3). This concept is shown in Fig. 3.7.

![Figure 3.7 Concept of associativity.](image)

The precedence specifies the order of evaluation of operators with their operands. Operators with higher precedence are evaluated first. For example, the operator * has higher precedence than -, so 1 - 2 * 3 is evaluated as 1 - (2 * 3) rather than (1 - 2) * 3. Note that in this example the - indicates subtraction and is a binary operator with precedence 4. The - also can be used as a negative sign, which is a unary operator with precedence 2. For example, -2 + 3 * 4 is evaluated as (-2) + (3 * 4) rather than -(2 + (3 * 4)). This concept is shown in Fig. 3.8.

![Figure 3.8 Concept of precedence.](image)
Note from Table 3.8 that the assignment operator has the lowest precedence. This means that in an assignment statement, the other operators are executed and then the assignment takes place. This, of course, is the desired intent. Also, the -assignment operator associates from right to left. This means the expression on the right side of the assignment operator is assigned to the variable on the left (again, the desired action).

Using parentheses. Parentheses can control the order of operation in arithmetic expressions. Arithmetic operators located within the parentheses always are executed prior to any outside the parentheses. When an arithmetic expression contains more than one pair of parentheses, the operators located in the innermost pair of parentheses are executed first. For example, the + operator in the statement

\[ z = ((a + b) - c / d); \]

will be executed before the - or / operator and a + b will be evaluated first.

To illustrate how to use the precedence table in deciding where parentheses are important, consider the static_cast operator. Because the static_cast operator has higher precedence (4) than the arithmetic operators (5) parentheses may not be needed in some cases. For instance, with the declarations

```cpp
int a, b;
double x;
```

the statement

```
x = (static_cast<double>(a)) / (static_cast<double>(b));
```

is equivalent to

```
x = static_cast<double>(a) / static_cast<double>(b);
```

because the cast operation is performed before the division operation. In addition, the effect of C++’s arithmetic operation rules is that sometimes you may not need to use a large number of cast operators in an expression. For instance, the previous assignment statement could be written as

```
x = static_cast<double>(a) / b;
```

and give the same result. This works because (as we described for mixed type arithmetic) dividing a double by an int is done in C++ by copying the int in double form and dividing the two as double variables as illustrated in Fig. 3.4.

Side effects. The primary effect of evaluating an expression is arriving at a value for that expression. Anything else that occurs during the evaluation of the expression is considered a side effect. For instance, the primary effect of the C++ statement (assuming i originally is 7)

```
j = i++;
```
is that the expression on the right side of the assignment statement is found to have a value of 7 (and then the assignment is made). The side effect of this statement is that the value of i is incremented by 1 (to make i equal to 8).

Consider the following C++ statement:

\[ j = (i = 4) + (k = 3) - (m = 2); \]

Its primary effect is to arrive at the value of the expression on the right side of the assignment statement (which is 5 obtained from \(4 + 3 - 2\)). Three side effects occur during the evaluation of the expression:

1. i is set equal to 4.
2. k is set equal to 3.
3. m is set equal to 2.

At times, side effects can be confusing. For the statement

\[ k = (k = 4) \times (j = 3); \]

the result of k will be 12 instead of 4. It is best not to use side effects except in their simplest form, such as

\[ i = j++; \]

or

\[ i = j = k = 5; \]

Note that because the associativity of the assignment operator is from right to left, multiple assignment statements such as the preceding one can be written. The order of operation is

1. \(k = 5\)
2. \(j = k\)
3. \(i = j\)

Also, an expression

\[ i = j = k = 2 + n + l; \]

is evaluated in this order:

1. \(k = 2 + n + l;\)
2. \(j = k;\)
3. \(i = j;\)

because the addition operation has a higher precedence than the assignment operator.
Programming with C++ style arithmetic. All this may sound confusing, but when writing arithmetic statements in your programs you need only to keep in mind a few things to handle most cases:

- When you have a division operation, make sure it does not involve two integer type variables or constants unless you really want the fractional part cut off. Remember, check your variable types before you write a division operation.

- When you are writing your code and a double or floating type variable is on the left side of an assignment statement, to be safe, use decimal points for any constants on the right side of the assignment statement. You may get the correct result without using decimal points, but we recommend that you use decimal points until you feel comfortable with mixed type arithmetic.

- When an int type variable is on the left side of an assignment statement, you must make sure that the arithmetic expression on the right side of the assignment statement creates an integer value. If you observe that it creates a real value, you must realize that the fractional part will be lost when the assignment is made.

Programming with C++ precedence rules. The most important precedence rule for you to remember is that multiplication and division occur before addition and subtraction. Knowing and using this fact will make your arithmetic expressions easy to write without the need for many parentheses. If you feel confident using the precedence table and have it handy while you are programming, you can use it to write statements involving other operators. However, when you do not have the table available or are in doubt, use parentheses to write your equations and control the order of operation. As you develop experience, you will be able to avoid parentheses in some cases.

It is not necessary to memorize the precedence list in Table 3.8 (and as we learn more operators, the precedence list becomes even longer). In this text we will indicate the importance of parentheses when introducing new operators. For now though, to be safe, use parentheses whenever you are uncertain about the precedence rules.

LESSON 3.6 ARITHMETIC OPERATIONS (3): MATH LIBRARY FUNCTIONS AND DATA TYPES

TOPICS

- Using the standard math header
- Contrasting the double and float data types
- Other data types

Your calculator makes it very easy for you to perform such operations as sin, log, and square root by having single buttons for them. Similarly, the C++ compiler makes it easy for you to perform these operations by providing mathematical library functions that you can call from your program. This lesson illustrates the use of some of these library functions. Without going into unnecessary detail at this point, know that to call a function we need to write the function name followed by parentheses. Enclosed in
the parentheses may be variables or values which serve as arguments (meaning values that are transferred to the function being called). The functions in this lesson's program operate on the argument values and return other values.

In this lesson we also introduce other real and integer data types (including the real data type float). Read the program with your calculator in hand, and see if you can deduce what is returned by each function and the difference between the float and double data types.

**Source Code**

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

int main ()
{
    double x = 3.0, y = 4.0;
    double a, b, c, d, e, f;
    float g;
    
    a = sin(x);
    b = exp(x);
    c = log(x);
    
    d = sqrt(x);
    e = pow(x,y);
    f = sin(y) + exp(y) - log10(y) * sqrt(y) / pow(3.2,4.4);
    g = log(x);

    cout << setprecision(9) 
        << "x= " << x << " y= " << y << endl
        << "a= sin(x) = " << a
        << " b= exp(x) = " << b
        << " c= log(x) = " << c 
        << " d= sqrt(x) = " << d 
        << " e= pow(x,y) = " << e 
        << " f= sin(y) + exp(y) * log10(y) * sqrt(y) / pow(3.2,4.4) = " << f
        << " g= log(x) = " << g << endl;
}
```

**Output**

- \( x=3 \) \( y=4 \)
- \( a=\sin(x) = 0.141120008 \)
- \( b=\exp(x) = 20.085536923 \)
- \( c=\log(x) = 1.098612289 \)
- \( d=\sqrt{x} = 1.732050808 \)
- \( e=\text{pow}(x,y) = 81 \)
- \( f=\sin(y)+\exp(y)\log10(y)\times\sqrt{y}\div\text{pow}(3.2,4.4) = 53.834136299 \)
- \( g=\log(x) = 1.098612309 \)

Use your calculator to decide which of \( c \) or \( g \) is more reliable.
Description

**Real data types.** The float data type is one of C++’s floating-point (real) data types. The float type usually occupies half the memory (4 bytes) of the double type (8 bytes). The amount of memory used influences the range or the number of significant digits. Because of the extra precision given by double types, in this text we will use double rather than float to represent real numbers. However, we show you float because you may see it in other programs, and we want to illustrate the effect of using a smaller amount of memory to store numbers.

We already have observed that real data types like double and float types cause values to be stored in exponential binary form and that the double data type occupies more memory than the float data type. When is it a good idea to use the extra memory and make sure that we carry a greater number of digits? Carrying a greater number of digits may be important when a large number of calculations are to be done. The drawback in declaring all variables as being of the double data type is that more memory is required to store double type variables than float type variables.

Consider the following example, which illustrates the effect of the number of digits carried in a calculation. You should try this on your calculator. Suppose you are multiplying a number by \( \pi \), 100 times. You essentially will be computing \( \pi^{100} \). The influence on the number of significant digits used for \( \pi \) is the following. Using five significant digits (similar to float) for \( \pi \) gives

\[
(3.1416)^{100} = 5.189061599 \times 10^{49}
\]

while using eight significant digits (similar to double) for \( \pi \) gives

\[
(3.1415926)^{100} = 5.1897839464 \times 10^{49}
\]

Here, it can be seen that while the first estimate of \( \pi \) has five significant digits, \((3.1416)^{100}\) is accurate only for the first four digits. This illustrates that accuracy is reduced after numerous arithmetic operations. Since one computer program easily can do one million operations, one can begin to understand the need for initially carrying many digits.

We can even see a difference in the values calculated for \( \log(3) \) for this lesson’s program. With double we get the natural log to be

\[
c = \log(3) = 1.098612289
\]

whereas with float we get

\[
g = \log(3) = 1.098612309
\]

If you compare these with your calculator, you will find that the double result is more accurate. You should be aware that your calculator probably carries 12 or more digits, whereas float minimally carries only 6 digits. Therefore, you should not use float if you want to be at least as accurate as your calculator.

In addition to the float and double data types is long double. The “long double” type occupies even more memory than double. This means that “long double” carries more digits, is capable of storing
larger numbers, and is more accurate for calculations than float and double. However, double and “long
double” are the same under Microsoft Visual Studio.

The real data types, float, double, and long double, are compared in Table 3.9. The C++ standard does
not state absolutely how much memory each of the types should occupy. So the table lists just the
minimums. Check your compiler for the -actual amount of memory it uses for each of these data types.

<table>
<thead>
<tr>
<th>Item</th>
<th>float</th>
<th>double</th>
<th>long double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum memory used</td>
<td>4 bytes = 32 bits</td>
<td>8 bytes = 64 bits</td>
<td>10 bytes = 80 bits</td>
</tr>
<tr>
<td>Range of values</td>
<td>1.1754944E−38 to 3.4028235E+38</td>
<td>2.2250738E−308 to 1.7976935E+308</td>
<td>Approximately 1.0E+308 to 1.0E+4932</td>
</tr>
<tr>
<td>Precision (digits)</td>
<td>Almost 7</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>

**Integer data types.** The different integer data types are compared in Table 3.10, again with minimal
sizes that are sometimes used. Note that the range of possible values for the integer data types is
considerably smaller than the range for the real data types. Also note that C++ has a number of
synonyms for the same data type. In other words, int and signed int are all the same. Most programmers
simply use int for this type. However, you may see old programs that use one of the other names for int.

<table>
<thead>
<tr>
<th>Item</th>
<th>short int</th>
<th>signed short int</th>
<th>unsigned short int</th>
<th>int</th>
<th>signed int</th>
<th>unsigned unsigned int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory used</td>
<td>2 bytes = 16 bits</td>
<td>2 bytes = 16 bits</td>
<td>4 bytes = 32 bits</td>
<td>4 bytes = 32 bits</td>
<td>4 bytes = 32 bits</td>
<td>4 bytes = 32 bits</td>
</tr>
<tr>
<td>Range of values</td>
<td>−32768 to 32767</td>
<td>0 to 65535</td>
<td>−2147483648 to 2147483647</td>
<td>0 to 4294967295</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that “long long int” or “long long” and “unsigned long long” have a greater range than int. Using
one of these types may eliminate a problem with exceeding the integer range in one of your programs
as described in Lesson 3.4.

**Math functions.** The meanings of all the C++ mathematical library functions are shown in Table 3.11. It
is very important that you notice that the argument for the sin() function (as well as the tan(), cos(), and
other functions that use angles as input) must be in radians, not degrees! So, if you want to use the sin
of 30 degrees in one of your programs, you must manually write source code that converts degrees to
radians by multiplying by π/180. For instance, this code might be

```c
angle = 30.0;
x = angle * 3.141592654/180.0;
y = sin(x);
```

You should carry a large number of digits for π to maintain accuracy.

You should read Table 3.11 to get an idea of what is available. You do not need to memorize the table.
You can simply refer to it later when you need to use any of the functions. Another important point is
that the log function calculates the natural logarithm. To calculate the base 10 logarithm use the log10
function.
To use C++ math functions, you need the cmath or cstdlib headers. Make sure your programs have:

```c++
#include <cmath>
```

or

```c++
#include <cstdlib>
```

for using the abs() function.

### APPLICATION EXAMPLE 3.1  TEMPERATURE UNITS CONVERSION

#### Problem Statement

Write a program that creates a table of degrees Celsius with the corresponding degrees Fahrenheit. Begin at 0°C and proceed to 100°C in 20°C increments. Use no more than two variables in your program.
Solution

First, assemble the relevant equations. The equation converting degrees Celsius to degrees Fahrenheit is

\[ F = \left(\frac{9}{5}\right)C + 32 \]

where \( C \) = degrees Celsius  
\( F \) = degrees Fahrenheit

SPECIFIC EXAMPLE

Once again, for this simple program, all the calculations can be done by hand.

\[
\begin{align*}
C &= 0 \\
F &= C \left(\frac{9}{5}\right) + 32 = 32 \\
C &= 20 \\
F &= C \left(\frac{9}{5}\right) + 32 = 68 \\
C &= 40 \\
F &= C \left(\frac{9}{5}\right) + 32 = 104 \\
C &= 60 \\
F &= C \left(\frac{9}{5}\right) + 32 = 140 \\
C &= 80 \\
F &= C \left(\frac{9}{5}\right) + 32 = 176 \\
C &= 100 \\
F &= C \left(\frac{9}{5}\right) + 32 = 212
\end{align*}
\]

ALGORITHM

We use the specific example to guide us in writing the algorithm. We add to it the printing of the headings and the results.

Begin
Declare variables
Print headings of table
Set \( C = 0 \)
Calculate \( F \)
Print \( C \) and \( F \)
Set \( C = 20 \)
Calculate \( F \)
Print \( C \) and \( F \)
Set \( C = 40 \)
Calculate \( F \)
Print C and F
Set C = 60
Calculate F
Print C and F
Set C = 80
Calculate F
Print C and F
Set C = 100
Calculate F
Print C and F
End

Source Code

This source code has been written from the algorithm. Note that this code has used the fact that the values of degrees Celsius are in increments of 20. Again, read this code line by line and make sure that you understand exactly how the program operates.

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

int main ()
{
    double degC, degF;
    cout << "Table of Celsius and Fahrenheit degrees\n";
    cout << "         Degrees    Degrees\n";
    cout << "      Celsius    Fahrenheit\n";

    degC = 0.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;

    degC += 20.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;

    degC += 20.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;

    degC += 20.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;

    degC += 20.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;

    degC += 20.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;

    degC += 20.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;

    degC += 20.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;

    degC += 20.;
    degF = degC * 9./5. + 32.;
    cout << setw(16) << degC << setw(20) << degF << endl;
}
```
Comments

First, we can see immediately that this program has the same three statements written repeatedly. Had we wanted to display the results for every single degree between 0 and 100 instead of every 20th degree, the program would have been extremely long but with the same three statements written over and over again. Chapter 6 has more advanced programming techniques to allow us to write a program that can accomplish the same task but with many fewer statements.

Second, we could have used the programming technique illustrated in the previous application program, which had a single cout statement at the end of the program instead of one immediately after each calculation of degF. However, this would have necessitated the use of more variables.

For instance, the program could have been

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main ()
{
    double degC1, degC2, degC3, degC4, degC5, degC6,
           degF1, degF2, degF3, degF4, degF5, degF6;
    cout << "Table of Celsius and Fahrenheit degrees\n\n";
    cout << "Degrees Degrees \n";
    cout << "Celsius Fahrenheit \n";
    // Your code here...
}
```
With this program 12 variables have been used instead of just 2. Variables take up space in the memory of the computer, so the program with 12 variables would occupy more memory than the program with just 2 variables. Efficient programming, in part, means to write a program that takes as little memory as possible. For this very small program, either programming technique could be used on today’s computers. However, for very large programs, the memory needed by the program may be very important. So, it is good to develop efficient programming habits while you are learning programming. Reducing memory size is only a part of developing efficient programs. Comments on other ways to make your programs efficient will be made throughout this book.

Also note that it is necessary to make your program understandable to someone other than you, the reason being that it is common for programs to be developed by teams of people and then undergo several revisions. This means that someone who has never seen a particular program may be responsible for modifying it. Your program is more valuable if it is easily understood.

Sometimes you will find a conflict between understandability and efficiency. In other words, efficient programs may not be understandable and understandable programs may not be efficient. Should you write code that is efficient but confusing, make sure that you comment it extremely well. Such comments can go a long way in making the code both efficient and understandable. If you run into a conflict, though, you should consult your employer or your course instructor for guidance in determining the more important characteristic for your particular program.

Note that the division is specified to be 9.0/5.0 (using decimal points) not 9/5 (no decimal points). The decimal points are required to prevent integer division, which causes a truncation of the decimal portion.
CHAPTER 5 DECISION MAKING

Chapter Topics
In this chapter, you will learn how to:

- Create if and if-else structures
- Use logical operators and relational expressions
- Make switch structures
- Use the bool data type
- Write programs with decision making

You will find as you continue learning to program that you will want your programs to make decisions regarding which calculations to perform. For instance, suppose you want to write a program that computes your income tax. Suppose that the percentage of tax is based on your income in the following way:

<table>
<thead>
<tr>
<th>Income</th>
<th>Percent tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–$50,000</td>
<td>20%</td>
</tr>
<tr>
<td>$50,000–100,000</td>
<td>30%</td>
</tr>
<tr>
<td>&gt;$100,000</td>
<td>40%</td>
</tr>
</tbody>
</table>

For your program to correctly compute your tax, it must be able to decide which percent tax applies to your income. In this chapter you will learn how to get your programs to make decisions of this sort.

LESSON 5.1 if CONTROL STRUCTURE (1)—basics

TOPICS
- Simple if statements
- Block if statements
- Controlling program flow
- Relational operators
- Relational expressions

The if statement or if control structure is capable of making decisions in C++. The form of the if statement is fairly simple. A relational expression (meaning an expression that compares the values of two variables, for instance) is contained in the if statement; if the relational expression is true, then the statements within the “true” group are executed. If the relational expression is false, then the statements are not executed.

The program for this lesson has four if statements. This program checks a pass code entered by a user. If the pass code is correct, the user must enter it one more time to verify it. If the pass code is incorrect, the user is given a second opportunity to enter it. If it is incorrect a second time, the user is notified that
access is denied. The correct pass code is built into the program to be 8765. Because a user of the program does not normally see the source code, the pass code is unknown to an invalid user.

Unlike the other programs we have studied, not all the statements in this program are executed (because the if control structure causes some of them to be skipped over). Do your best to trace the actions of the code line by line and see which statements are executed and which are not. Look at the keyboard input to help you follow the flow.

The purpose of this program is to show basic if structures and different operators commonly used with them.

```
#include <iostream>
using namespace std;
int main ()
{
    int i, pcode_entered;
    const int pcode = 8765;
    cout << "Enter your pass code." << endl;
    cin >> pcode_entered;
    if (pcode_entered < pcode) cout << "Incorrect code "
        "(too small). Enter it again." << endl;
    if (pcode_entered > pcode) cout << "Incorrect code "
        "(too large). Enter it again." << endl;
    if (pcode == pcode_entered) cout << "Verify your "
        "code by entering it again." << endl;
    cin >> pcode_entered;
    if (pcode_entered == pcode) {
        cout << "Access approved." << endl;
        cout << "Welcome!" << endl;
    }
    if (pcode_entered != pcode) cout << "Access denied." << endl;
}
```

**Output**

<table>
<thead>
<tr>
<th>Keyboard input</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter your pass code.</td>
<td>8766</td>
</tr>
<tr>
<td>Incorrect code (too large). Enter it again.</td>
<td></td>
</tr>
<tr>
<td>Keyboard input</td>
<td>8765</td>
</tr>
<tr>
<td>Access approved.</td>
<td></td>
</tr>
<tr>
<td>Welcome!</td>
<td></td>
</tr>
</tbody>
</table>
Description

Relational expressions. The expression

\[ \text{pcode-entered} < \text{pcode} \]

is a relational expression that compares the values of two arithmetic expressions. A relational expression is a type of logical expression and produces a result of either true or false. Here, it checks whether the value of the variable pcode-entered is less than the value of the variable pcode. Its general syntax is

\[ \text{left_operand \ relational_operator \ right_operand} \]

where the left_operand and right_operand can be variables, such as pcode-entered in this lesson’s program or any arithmetic expression. The relational_operator is used to compare the values of two operands. C++ contains six relational operators:

<table>
<thead>
<tr>
<th>Relational operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

Observe in the table the meaning of !=. No ≠ symbol is found on the keyboard, so C++ uses !=.  

Simple if statements. A simple if statement can be generalized as

\[ \text{if (expression) \ statement;} \]

where expression, represents a logical expression and statement, is an executable statement. For instance

\[ \text{if (pcode-entered < pcode) cout} \ll \text{"Incorrect code } \text{(too small). Enter it again."} \ll \text{endl;} \]

uses pcode-entered < pcode as the expression and cout "Incorrect code (too small). Enter it again." << endl; as the statement. Note that any executable statement can be used as the statement, including if and other control statements. A logical expression produces a result of either true or false. If the logical expression is true, the statement is executed. If the logical expression is false, the statement is not executed. The logical expression within an if statement is called a condition or test expression.

Make sure that you do not put a semicolon immediately after the parentheses enclosing the expression. In other words, if you write

\[ \text{if (expression);} \text{ \ statement;} \]
then you will have created two independent statements. The first uses an if conditional that does nothing whether the expression is true or false. The second is a statement that will always be executed (because the if conditional has no effect on it. This is clearly not what you want. Remember, do not put a semicolon separating the test expression from the executable statements.

Block if statements. The statement

```c++
if (pcode_entered == pcode) {statements...}
```

is called a block if statement. If the logical expression (between the two parentheses) is true, the statements in the “true” block (between the two braces) are executed. Otherwise, the entire true block of statements is ignored. The general form of a block if statement is

```c++
if (expression)
{
    executable statement 1;
    executable statement 2;
    ...
}
```

The code block should be indented one tab (at least three spaces) in from the keyword, if. We will see as we go further in this book that indentation is an important way of making your program understandable to others. Although there are no absolute rules on indentation (it is not required by standard C++), it has become common practice in programming.

Comparing the equality of two integer values. The equality relational operator `==` (with 2 not 1 = signs) is capable of comparing two values. If the two values are equal, the relational expression using `==` has the value true. If the two values are not equal, the relational expression using `==` has the value false. The equality relational operator is commonly used in test expressions for if control structures.

The `==` operator works very well for integer type comparisons (with integer values on both sides of `==`). However, a major error can occur if you mistakenly use `=` rather than `==`. For instance, in this lesson’s program, we could have mistakenly written just one equal sign and the expression in the form:

```c++
if (pcode_entered = pcode) cout << "Verify your "
    "code by entering it again." << endl;
```

The result is that `pcode_entered` is assigned the value of `pcode`, and the cout statement is executed (for a reason we will see in the next lesson)! This type of error is very difficult to find in a large program. The compiler does not help because it accepts the single equal sign. Consequently, you must find this error on your own.

If the comparison is to a constant, then this form of writing the comparison works well:

```c++
if (2 == x)
```

because the compiler will not accept

```c++
if (2 = x)
```
if we accidentally type just one instead of two equal signs. Consequently, the compiler has helped us find the single equal sign problem. In this lesson’s program, the constant is on the left side of the expression in

```c++
if (pcode == pcode_entered) cout << "Verify your "
   "code by entering it again." << endl;
```

because pcode is declared to be const int. If we had accidentally typed (with =)

```c++
if (pcode = pcode_entered) cout << "Verify your "
   "code by entering it again." << endl;
```

the compiler would have indicated an error for us. In general, be very careful when you use ==. Make sure you do not use just one =.

**Comparing the equality of two real values.** It is not recommended to use == to compare the values of double, long double, or float type variables in most cases, although the C++ compiler will not indicate an error if it is done.

The reason why it is not recommended for real number comparisons is that a typical C++ compiler carries a large number of significant digits for the real variable types (the number depends on the number of bytes used to store the type). If two doubles are compared with == and they differ in only the last significant digit, then the result of a comparison with == is false. For instance, if `a = 12.3456789123456789` and `b = 12.3456789123456788`, then the result of `a == b` is false although they differ by only $10^{-16}$. Often in real programs, we perform calculations using slightly approximate values and compare them with other approximate values. In many cases, we are interested only if the numbers are nearly or very nearly equal. Therefore, we want a comparison to evaluate to true when the values are very nearly equal. Because == does not do this, we do not use it.

Also, the binary representation of decimal numbers often requires an approximation. For instance, we need many binary bits to represent the simple decimal number 5.3 exactly. Because of this characteristic, numbers that we calculate to be exact with decimal arithmetic may not be exact using binary arithmetic with a limited number of bits. In other words, a calculation that you believe is exact using a hand calculation in decimal may not be exact using binary with a limited number of bits. We avoid the comparison problems caused by this effect by not using == for real type variables.

If we do not use == to compare real numbers, what do we do? One way to compare real values is to use the fabs function and <. For instance, to compare the double values of `a` and `b` as listed previously, the following statement would evaluate to true:

```c++
if (fabs (a - b) < 1.0e-10)
```

Here, we have somewhat arbitrarily selected the constant 1.0e – 10 as a very small number. In writing your own programs, you must decide how small that number should be based on what you require for your problem.
Comment. The following cout statements were not executed in this lesson’s program for the given input data:

cout << "Incorrect code (too small). Enter it again." << endl;
cout << "Verify your code by entering it again." << endl;
cout << "Access denied." << endl;

because the relational expressions preceding these cout statements all evaluated to false.

Note that an inefficiency in this lesson’s program is that five relational expressions are executed. Fewer relational expressions (and therefore, fewer comparisons, which is more efficient) are needed when we use if-else control structures. We cover these next, in Lesson 5.2.

LESSON 5.2 if CONTROL STRUCTURE (2)—SIMPLE if-else

TOPICS
- Simple if-else control structures
- The conditional ?: operator

Another form of the if statement is the if-else form. It is used when a group of statements is to be executed if the logical expression is false.

The program in this lesson computes whether or not a company is profitable. The program prompts for revenue and expenses and computes either the profit or loss. An if-else control structure is used to decide which calculations to perform.

In this lesson we introduce the ?: operator. It is the only operator in C++ that is a ternary operator, meaning that three operands are needed for it to be used properly. In this program, we have used all three operands on the right side of an assignment statement. A colon separates two of the operands. We use the ?: operator to help determine the interest on the company’s debt if it is operating at a loss.

Read the source code line by line along with the keyboard input portion of the output section. Determine which statements are executed and which are not executed.
Simple if-else control structure. The syntax of a simple C++ if-else control structure is

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

int main() {
    double revenue = 0, expenses = 0, profit = 0, loss = 0, interest = 0;

    cout << setprecision(2) << setiosflags(ios::showpoint);
cout << "Enter the company's revenue and expenses:" << endl;
    cin >> revenue >> expenses;
    cout << endl << endl;

    if (revenue > expenses) {
        profit = revenue - expenses;
        cout << "The company is profitable.\n" << "The company's profit for this month "
            "is: $" << profit << endl;
    }
    else {
        loss = expenses - revenue;
        cout << "The company is running a loss.\n" << "The company's loss for this month "
            "is: $" << loss << endl;
    }

    interest = (loss > 0.0) ? (0.05 * loss) : (0.0);
    cout << "The interest the company owes on its debt is $" << interest << endl;
}
```
if (expression)
{
    executable statement 1a;
    executable statement 1b;
    ...
}
else
{
    executable statement 2a;
    executable statement 2b;
    ...
}

Executable statements 1a, 1b, ... are part of the “true” block, whereas executable statements 2a, 2b, ... are part of the “false” block. If the expression is true, statements in the true block are executed. If the expression is false, control is transferred to the false block. If the statement block (either true or false) contains more than one statement, the block must be bounded by a pair of braces; otherwise, braces are optional. For example,

    if (test >= 0)
    {
        true block statements...
    }
    else
    { a single statement;}

where the true block contains more than one statement and therefore must be bounded by a pair of braces. However, the false block contains only one statement, so the braces are optional.

If no statements are to be executed in the false block, then the false block may be omitted. The syntax without the false block is as given in Lesson 5.1.

    if (expression)
    {
        executable statement 1a;
        executable statement 1b;
        ...
    }

Note that the impact of using such a control structure is to cause the execution of one block while bypassing another.

The ? : conditional operator. The ? : operator requires three operands. Its form is the following:

    expression1 ? expression2 : expression3

If expression1 is true, expression2 is evaluated. If expression1 is false, expression3 is evaluated. The value of the entire ? : expression becomes equal to the value of the expression evaluated (either expression2 or expression3).
For this lesson’s program, expression1, loss > 0.0 is true. Therefore, expression2, 0.05 * loss is evaluated, and the value of the right side of the assignment becomes equal to the value of expression2. Thus, interest becomes equal to 0.05 * loss.

As another example, the ? : operator can be used to find the smaller of two numbers. The statement

\[ x = (y < z) \ ? \ y \ : \ z; \]

assigns x the value of the smaller of y and z.

Statements using the ? : operator are good shorthand for longer if-else type control structures. Note that for the unevaluated expression no side effects occur.

LESSON 5.3 if CONTROL STRUCTURE (3)—NESTED if-else

TOPIC
- Nested if-else control structures

if-else control structures can be nested, meaning an if-else control structure can be contained within another if-else control structure.

Suppose you are a civil engineer managing a construction project. You may be interested in writing a program that tells you what should be done during a phase of a project during the week at a particular time.

Say that the following 24-hour, 7-day schedule is needed to complete the phase of the project in a timely manner:

**Weekdays**
(Days 1–5)
0:00–9:00 Drive piles
9:00–24:00 Construct formwork

**Weekends**
(Days 6–7)
0:00–8:00 Maintain equipment
8:00–24:00 Pour concrete

This schedule can be used to produce the following algorithm, which illustrates the logic that you would use to respond to a question about what would be done on a particular day at a particular time:

If day = weekday (1-5), then
   if time = 0:00-9:00 Drive piles
   if time = 9:00-24:00 Construct formwork
If day = weekend (6-7), then
if time = 0:00-8:00 Maintain equipment
if time = 8:00-24:00 Pour concrete

A computer program written in C++ can duplicate the logic of this algorithm. Examine the source code to see how if-else statements can be used to mimic this. Note that these if-else statements are said to be nested.

Source Code

```cpp
#include <iostream>
using namespace std;
int main ( )
{
    int day;
    double time;

    cout << "Type the day and time of interest" << endl;
    cin >> day >> time;

    if (day <= 5)
    {
        if (time <= 9.00)
            cout << "Drive piles" << endl;
        else
            cout << "Construct formwork" << endl;
    }
    else
    {
        if (time <= 8.00)
            cout << "Maintain equipment" << endl;
        else
            cout << "Pour concrete" << endl;
    }
    cout << "End of program" << endl;
} 
```

Output

Type the day and time of interest

Keyboard input 3 10.00

Construct formwork

Description

**Nested if-else control structures.** In C++, different levels of if-else control structures can be nested. One can trace which statement blocks are executed by closely following the code. For example, follow these nested structures:
Each nested loop that you create or examine in someone else’s code must be evaluated on a case-by-case basis. A conceptual illustration of the nested if-else control structure used in this lesson is given in Fig. 5.1. Note the branching that is produced by the nesting.

In nested if-else statements, the total number of “ifs” can be greater than or equal to, but not less than the total number of “elses.” By default, an else clause is associated with the closest previous if statement that has no other else statement. Hand drawn arrows can be used to clarify the pairing of “if” and “else” clauses as shown at the beginning of this lesson’s description. The arrows should not cross in properly written nested if-else statements. If arrows do cross, you must rewrite your if statements so they do not cross.
As if-else control structures become nested, the flow of programs becomes more difficult to trace. Making the program more readable can improve the ease at which it can be followed. The traditional way to make a program readable is to use indentation. We recommend that you indent each pair of if-else statements so that the inner else is paired with the inner if and the outer else is paired with the outer if. An example of this is shown in the indentation of this lesson’s program.

LESSON 5.4 LOGICAL OPERATIONS (1)—LOGICAL OPERATORS

TOPICS
- Using logical operators
- Using logical expressions

Logical operators can be used to connect two relational expressions. For example, the following

\[
\text{if (x == 0 && y == 0)}
\]

is read, “If \(x\) is equal to 0 and \(y\) is equal to 0.”

Here, the logical operator is `&&` which is read “and.” It connects the two relational expressions, \(x == 0\) and \(y == 0\). The other two logical operators in C++ are `||` and `!`. These are used in this lesson’s program. Read the source code and follow the flow of the program line by line. This program performs no useful task; it simply illustrates logical operators.
Logical operators. C++ has three logical operators, &&, ||, and !. The && and || operators are binary operators because they appear between two relational expression operands. The ! operator is unary because it precedes a single relational operand. The formal meanings of these operators follow:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Operation</th>
<th>Operator type</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>Negation</td>
<td>Unary</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>Conjunction</td>
<td>Binary</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
</tbody>
</table>

The logical NOT operator reverses the result of a relational expression. For instance, if \( x \) is 5 and \( y \) is 0, \( x \) is not equal to \( y \), and the expression \( x == y \) is false. The logical NOT operator can be used to reverse the false value. Therefore,

\[
!(x == y)
\]

is true.

The logical AND and logical OR operators perform much the same way the words “and” and “or” were used in your very early mathematics classes. The C++ statement

\[
if (x > 0 && y >= 0)
\]

is read, “If \( x \) is greater than 0 and \( y \) is greater than or equal to 0,” meaning that the logical expression enclosed in parentheses is true when both \( x > 0 \) and \( y >= 0 \) are true. On the other hand,

\[
if (x == 0 || y == 0)
\]

is read, “If \( x \) equals 0 or \( y \) equals 0.” For the logical expression between the parentheses to be true in this case, only one of

\[
x == 0 \quad y == 0
\]

need be true.

In the table that follows, we have indicated relational expressions with the symbols A and B. You can use this table to determine the result of a logical expression that includes the two relational expressions. For example, suppose again that \( x \) is 5 and \( y \) is 0. For a relational expression A being \( x > 0 \) and a relational expression B being \( y >= 0 \), the relational expression A is true, and the relational expression B is true. This means that we use line 1 in the table. Therefore, A && B is equivalent to true && true and the logical result is true. Looking again at line 1, we can determine the logical result of A || B, !A, and !B.

Similarly, for the relational expressions A being \( x == 0 \) and B being \( y == 0 \), we have A is false and B is true. This leads us to line 3 of the table. Reading across this line, we get A || B being true. We also can look at it as false || true, which results in true.
Finally, consider relational expression A being \( x = y \) (which is false) and relational expression B being \(! (x = y)\) (which is equivalent to \(! (\text{false})\), which results in true). This also leads us again to line 3 of the table. This line indicates A && B is false. Otherwise, looking at this, A && B is true && false, which is false.

The results of other combinations of relational expressions within logical expressions can be discerned from the table.

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

One way to remember the table is to realize that for two expressions being operated on,

1. For &&, if one of the relational expressions is false then the result is false.
2. For ||, if one of the relational expressions is true then the result is true.

LESSON 5.5 LOGICAL OPERATIONS (2)—VALUES OF RELATIONAL EXPRESSIONS AND PRECEDENCE OF RELATIONAL AND LOGICAL OPERATORS

TOPICS
- Precedence of logical operators
- Finding the result of a logical expression

To this point, we have not mentioned that C++ gives relational expressions numerical values just as it gives arithmetic expressions numerical values. If a relational expression is false, C++ gives it a value of 0. If it is true, C++ gives it a value of 1 (which you should recognize as being nonzero). The C++ compiler also operates in the reverse manner. If the value of a relational expression is 0, then it knows the result is false; and if the value of a relational expression is not 0, then the result is true.

Something similar can be done with variables. If the value of a variable is 0, then it can be treated as false; and if the value of a variable is not 0, then it can be treated as true. This works this way because (you will recall) that a single variable can be considered to be an expression.

We saw earlier that C++ has an established order of precedence for arithmetic operators. Similarly, C++ has an established order of precedence for relational and logical operators. In this lesson’s program, we illustrate the precedence of relational and logical operators. Read and follow the source code and annotations. Use the output to help you determine how the program operates.
Description

**Precedence and associativity of logical, relational, and arithmetic operators.** The precedence and associativity of the logical, relational, and arithmetic operators follow:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Parentheses</td>
<td>L to R</td>
<td>1 (highest)</td>
</tr>
<tr>
<td>++, --</td>
<td>Postincrement</td>
<td>L to R</td>
<td>2</td>
</tr>
<tr>
<td>++, --</td>
<td>Preincrement</td>
<td>R to L</td>
<td>3</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>L to R</td>
<td>3</td>
</tr>
<tr>
<td>+, -</td>
<td>Positive, negative sign</td>
<td>L to R</td>
<td>3</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Multiplication, division</td>
<td>L to R</td>
<td>4</td>
</tr>
<tr>
<td>+=, -=</td>
<td>Addition, subtraction</td>
<td>L to R</td>
<td>5</td>
</tr>
<tr>
<td>&lt;=, &gt;=, &gt;, &lt;</td>
<td>Relational operator</td>
<td>L to R</td>
<td>6</td>
</tr>
<tr>
<td>==, !=</td>
<td>Relational operator</td>
<td>L to R</td>
<td>7</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND</td>
<td>L to R</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
<tr>
<td>+=, -=, *=, /=, %=</td>
<td>Compound assignment</td>
<td>R to L</td>
<td>10</td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>R to L</td>
<td>10 (lowest)</td>
</tr>
</tbody>
</table>
This table shows that parentheses have the highest order of precedence, followed by unary increment or decrement and logical NOT operators. In general, arithmetic operators, including multiplication, division, addition, and subtraction, have a higher order of precedence than any relational operator. Then come the binary logical operators, in which the logical AND has higher precedence than the logical OR. The assignment operator has the lowest precedence (meaning the assignment takes place after the other operations). The operators work (associativity) left to right—except the pre-increment or pre-decrement, compound assignment, and assignment operators.

For example, assuming $a = 4$, $b = -2$, and $c = 0$, the expression

$$x = (a > b || b > c && a == b)$$

is equivalent to the following expressions (note that the sequence of evaluation is based on the precedence level of each operator; we add parentheses at appropriate locations so that the expression can be grouped and evaluated):

$$x = (a > b || b > c && a == b)$$
$$x = ((a > b) || (b > c) && (a == b))$$
$$x = ((4 > -2) || (-2 > 0) && (4 == -2))$$
$$x = (TRUE || FALSE && FALSE)$$
$$x = (TRUE || FALSE)$$
$$x = (TRUE)$$

which results in true. The value of $x$ becomes 1 (true). If $x$ is true, !$x$ is false, and !$x$ is printed as 0 in this lesson’s program.

The effect of the different precedences of the relational and logical operators on the relational expressions in this lesson’s program is illustrated in Fig. 5.2.

![Figure 5.2](image)

Figure 5.2 Operation of compound logical expression $a > b || b > c && a == b$, from this lesson’s program.
Although not shown in this lesson’s program, a relational expression of the sort \( a > b == c \) is evaluated from left to right since all the operators have equal precedence. For instance, if \( a = 4 \), \( b = -2 \), and \( c = 5 \), this expression evaluates to false. The steps in evaluation are

\[
\begin{align*}
a > b & \text{ is true, giving this expression a value of 1} \\
1 == c & \text{ is false.}
\end{align*}
\]

Logical value of a single variable. The logical value of a single variable is false if the variable has a value of 0 and true if the value is nonzero. This is illustrated in Fig. 5.3a. For example, in this lesson, the logical value of \( c \) is false, since \( c \) is equal to 0, but the logical values of \( a \) and \( b \) are true, since \( a \) (which is 4) and \( b \) (which is -2) are nonzero. Also, the values of \( !a \) and \( !b \) are false and therefore printed out as 0, whereas the value of \( !c \) is printed out as 1. This is shown in Fig. 5.3b.

![Figure 5.3](image)

(a) True or false result of integer values. (b) Integer values for \( !\text{true} \) and \( !\text{false} \).

LESSON 5.6 if-else-if AND switch CONTROL STRUCTURES

TOPICS
- Using if-else-if control structures
- Using switch statements
- Comparing if-else-if and switch control structures

The if-else control structure executes one of two statement blocks. Frequently in programming, though, we want to execute one of a number (three, four, or more) of statement blocks. This is usually most conveniently done with an if-else-if or a switch control structure. Both these structures contain multiple statement blocks and have the feature that when one of the blocks is executed, the others are bypassed.

Two source codes are given for this lesson. They perform the same tasks but in different ways. The first source code uses an if-else-if control structure, and the second source code uses a switch control structure. Read and follow the flow of both codes using the output as a guide.
Source Code 1

```cpp
#include <iostream>
using namespace std;
int main() {
    int option;
    cout << "Please type 1, 2, or 3\n";
    cin >> option;
    if (option == 1) {
        cout << "Attend meeting\n";
    } else if (option == 2) {
        cout << "Debug program\n";
    } else if (option == 3) {
        cout << "Write documentation\n";
    } else {
        cout << "Do nothing\n";
    }
}
```

The value of `option` is read from the keyboard.

If-else-if control structure. Only one of the statement blocks (enclosed in braces) is executed.

Source Code 2

```cpp
#include <iostream>
using namespace std;
int main() {
    int option;
    cout << "Please type 1, 2, or 3\n";
    cin >> option;
    switch (option) {
    case 1: cout << "Attend meeting\n";
             break;
    case 2: cout << "Debug program\n";
             break;
    case 3: cout << "Write documentation\n";
             break;
    default: cout << "Do nothing\n";
    }
}
```

The value of `option` is read from the keyboard.

The keyword `switch` is followed by an integer constant and colon. If `option` is 1, control passes to this line. If `option` is 2 or 3, control passes to one of the other lines with the keyword `case`.

The keyword `break` causes exiting of the `switch` control structure.

Switch control structure.

The keyword `default`. If `option` is not 1, 2, or 3, control passes to this line.

Output from Both Source Codes

Please type 1, 2, or 3

**Keyboard input**

2

Debug program
Description

**if-else-if control structure.** An if-else-if control structure shifts program control, step by step, through a series of statement blocks. Control stops at the relational expression that is true and executes the corresponding statement block. After execution of that statement block, control shifts to the end of the control structure. If none of the relational expressions is true, the final statement block is executed. In this lesson’s program the value of option was read in to be 2, so the first statement block was not executed. Because the relational expression option == 2 was true, the second statement block was executed. The third and fourth statement blocks were bypassed, and control transferred to the end of the control structure.

The form of the if-else-if control structure is

```c
if (relational_expression_1)
{
    statement_block_1
}
else if (relational_expression_2)
{
    statement_block_2
}
.
.
.
.
.
else if (relational_expression_n-1)
{
    statement_block_n-1
}
else
{
    statement_block n
}
```

Figure 5.4 illustrates the if-else-if control structure for this lesson’s program. Note the branching that occurs due to the different values of option.

**switch control structure.** A switch statement or switch control structure commonly is constructed like the if-else-if control structure. It also is used to transfer control. Its syntax is

```c
switch (expression)
{
    case constant1:
        statement1a
        statement1b
        ...
    case constant2:
        statement2a
        statement2b
        ...
```
...  
    default:  
        statements  
    )  

where the expression must be enclosed in a pair of parentheses and must result in an integer type value when the program flow enters the switch block. A switch block must be bounded by a pair of braces. The terms constant1, constant2, and so on are integer type constant expressions. Note that all constant expressions are followed by colons. All the constant expressions must be unique, meaning that none can have the same value as another constant expression. Although not required, it is common that the last case type line is the keyword default. If no constant matches the value of the expression, the statements in the default case are executed. The default case is optional. If no default case is given and no constant expression matches the expression value, the entire switch block is ignored.

![Control Flow Diagram](image)

**Figure 5.4** if-else-if control structure for Lesson 5.6’s program. Compare to switch (Fig. 5.5) and nested if-else (Fig. 5.1) control structure illustrations.

Figure 5.5 shows the switch control structure for this lesson’s program. Compare this figure to Figs. 5.1 and 5.4. Note the similarities between the if-else-if, nested if-else, and switch control structures. In all the cases illustrated, the control structure has chosen a single block of code to execute and bypassed the others.

The keyword case can be used only in a switch control structure. It is used to form a label called a case label. A case label is a constant followed by a colon. The label does not affect the execution of the statement that follows it. In switch control structures, C++ looks for a match between the switch expression and the expression in a case label. C++ then executes the statement sequence following the
matching case label. For instance, for the form shown previously, if the value of the switch expression matches constant1, then the program flow is transferred to case constant1 and statement1a, statement1b, and so forth are executed. Because the switch control structure can search only for equality, it differs from the if-else-if control structure, which can use other relational operators.

A break statement in a switch control structure terminates execution of the smallest enclosing switch statement. The keyword break terminates (which means to send control to the point of the closing brace) the switch structure. We will see that break statements have other uses, which operate similarly in that they cause control to pass to a closing brace.

Often the last statement for each case is the break statement because it terminates the process and exits switch. If no break statement is used, then the statements in the next case are executed. For example, in the following code,

```c++
switch (option)
{
    case 1: cout << "Entering case 1\n";
            break;
    case 2: cout << "Entering case 2\n";
    case 3: cout << "Entering case 3\n";
            break;
}
```

Figure 5.5 switch control structure for Lesson 5.6's program. Note the importance of the break statement in controlling program flow for the switch control structure. Compare to if-else-if (Fig. 5.4) and nested if-else (Fig. 5.1) control structure illustrations.
if option is 1, then "Entering case 1" will be displayed on the screen. If option is 3, "Entering case 3" will be displayed. However, if option is 2, both "Entering case 2" and "Entering case 3" will be displayed because C++ first finds a match between the switch expression and a case label. Execution then continues, line by line, until a break or the end of the block (indicated by a closing brace) is encountered. This is because a statement label has no effect on the statement that follows it. A case statement label serves only as a marker to which control can be sent. The program flow when a break statement is missing is shown in Fig. 5.6.

![Diagram of switch control structure]

Figure 5.6 Flow of program control for the switch control structure of Lesson 5.6's program if no break statement were given for the case 2 block. Observe that the break statement causes control to exit the switch structure. With no break statement, control passes directly to the next block.

The keyword default is a special label used only for switch control structures. In the event that none of the case label constants agrees with the switch expression, control passes to the default labeled statement sequence. Because the label default is a keyword, it is not considered to be a user-defined label.

**Nested switch control structures.** We can nest switch control structures. A nested switch control structure could take the following form:

```cpp
switch (outer_expression)
{
    case constant_outer1:
    ...
```

```cpp```
switch (inner_expression)
{
    case constant_inner1:
        statement inner_1a
        statement inner_1b
        ...
    case constant_inner2:
        statement inner_2a
        statement inner_2b
        ...

    case constant_outer2:
        statement outer_2a
        statement outer_2b
        ...
    case constant_outer3:
        statement outer_3a
        statement outer_3b
        ...
}

An illustration of a nested switch structure is given in Fig. 5.7.

Figure 5.7 Nested switch control structure with break statements.
LESSON 5.7   THE bool DATA TYPE

TOPICS

- Using the bool data type
- Input and output for bool data

C++ supports another data type that we have not yet discussed— the bool data type, named after mathematician George Boole who worked in the area of logic. Unlike an int variable that can hold any integer value (within memory limits), a variable declared to be bool can contain one of only two values which are most conveniently regarded as 0 or 1. The 0 or 1 value of a bool variable can be interpreted to mean false or true, fail or pass, off or on, or any two states we are interested in. This type of representation can be useful for storing such information as the results of material testing among other things. For instance, if a steel piece has met the requirements for use in a machine a bool variable can be used to represent pass/fail for the test results.

In this lesson’s program, we illustrate how bool variables can help represent water quality. In this case, we want to indicate whether the water is salty, hard, acidic, has a good taste, and whether the user receives home service for this water. A bool type variable is useful because there are only two states for each of these. In other words, we classify the water as being salty or not salty, hard or not hard, acidic or not acidic, having a good taste or not a good taste, and a user receives the water or does not receive it. In the program, we create bool variables salty, hard, acidic, good_taste, and have_service. For instance, if we set the value of hard to be 0 (meaning false) then the water is not hard; if we set hard to 1 (meaning true) then the water is hard.

For the variables salty, hard, and acidic there are scientific tests that determine which state the water is in. If the sodium level is greater than 4000 mg/l, the water can be considered salty. If the calcium is greater than 40 mg/l and the magnesium is greater than 20 mg/l, the water can be considered hard. If the pH is less than 7, the water is acidic. As a result, we have double type variables in the program that represent the sodium, calcium, magnesium, and pH levels of the water. By checking the values of these variables, we can determine in which state each of the bool variables should be. This means we can assign the result of a relational expression to a bool type variable, and the bool variable indicates the state. Read the program to see how we do this.
Description

**bool data type.** The bool data type is a C++ data type that can be used to represent quantities that can have one of two states. To declare variables to be type bool, we use the keyword bool followed by a comma separated list of variable names in a manner similar to declaring int or double variables. For instance, to declare the variables salty, hard, and acidic to be bool, the following declaration is used:

```
bool salty, hard, acidic;
```

The states of a bool variable are most conveniently thought of as being 1 or 0, where 1 is equivalent to true and 0 is equivalent to false. Commonly, bool type variables occupy one byte of memory.

**Uses for bool variables.**

- As we showed in this lesson’s program, bool variables can represent characteristics of materials. We have declared bool variables salty, hard, acidic, and good_taste to represent characteristics of water. Two states are appropriate for each of these variables because the water can be thought to be salty or not salty, hard or soft, acidic
or basic, good tasting or bad tasting. This means that the variables are ideally suited to being bool type.

- The current status of events can be represented with bool type variables when a state can be considered to be either existent or not existent. For instance, in this lesson’s program, we have used the bool variable have_service. Having service is an event that either exists or not. When this variable has a value of true, the user has service. When it is false, the user does not have service. Other events that either exist or not, such as a switch being on or off, can also be represented by bool variables.

- In general, bool variables can be used for any quantity that can be described by one of two states.

Assigning values to bool variables.

- One way to assign a value to a bool variable is to write a logical expression on the right side of an assignment statement. For instance, in this lesson’s program we used:

```cpp
salty = (sodium > 4000);
hard = (Ca > 40 && Mg > 20);
acidic = (pH < 7);
```

In each case, if the logical expression is true, the value 1 is assigned to the bool variable. On the other hand, if the logical expression is false, the value 0 is assigned to the bool variable. This is illustrated in Fig. 5.8.

- We can also assign integer values to bool variables. Although not shown in this lesson’s program, we could have used the assignment statements:

```cpp
good_taste = 15;  // (the nonzero value, 15, indicates true and causes the value 1 to be assigned to good_taste)
good_taste = 0;   // (the zero value indicates false)
```

In the first of these, a nonzero integer has been assigned to a bool variable, which is equivalent to assigning a true logical expression. Any nonzero integer (negative or positive) produces the result of assigning a value of 1 to the bool variable even if the nonzero integer is not 1! In the second of these, zero has been assigned to a bool variable, which is equivalent to assigning a false logical expression. Only zero assigns a value of 0 to a bool variable.

In addition to using assignment statements to accomplish this, we can use cin statements as we have done in this lesson’s program with

```cpp
cin >> good_taste;
```
The user in our example entered the integer 1 (although any nonzero value would have been equivalent) for the value of good_taste. Again, this is the same as assigning a true logical expression to the good_taste variable. Had the user entered 0, it would have been the same as assigning a false logical expression to the variable good_taste. This is also illustrated in Fig. 5.8.

- We can also assign the keywords true or false to bool variables. Although not shown in this lesson’s program, we could have used the assignment statement:

  ```
  good_taste = true;
  ```

  or

  ```
  good_taste = false;
  ```

  Similarly a user can enter the words true or false in response to a cin statement as we did in this lesson’s program in response to

  ```
  cin >> boolalpha >> have_service;
  ```

to assign a value to the variable have_service. Note that the boolalpha manipulator is needed for C++ to properly interpret the true or false word input as shown in Fig. 5.8.

---

**Assigning bools to int variables.** We could have used in this lesson’s program code of the sort:

```c++
int i;
i = salty;
```
after the variable salty had been initialized. Because salty is false (sodium < 4000) it has a value of 0. The
int variable i then gets the value 0. Had salty been true, i would have been given a value of 1. This is
illustrated in Fig. 5.9.

![Diagram of boolean variable values](image)

**Figure 5.9** Assigning bool variable values to int types or printing bool variables.

**Printing the values of bool variables.** In using cout to print bool variable values, we can choose to use
the boolalpha manipulator or not. If we do not use bool-alpha, bool variable values are printed as 1 (for
true) or 0 (for false). For instance,

```cpp
cout << "Water composition" << " " << salty << "  hard " << hard << "  acidic " << acidic;
```

prints the line

```
Water composition  salty 0  hard 1  acidic 0
```

If we use the boolalpha manipulator, the values are printed as “true” or “false.” For instance,

```cpp
cout << boolalpha << "Water composition"
    << "  salty " << salty << "  hard " << hard << "  acidic " << acidic;
```

prints the line

```
Water composition  salty false  hard true  acidic false
```

These are illustrated in Fig. 5.9.

**Using bool variables with if statements.** Although not shown in this lesson’s program, we could have
used a line such as:

```cpp
if (salty) cout << "The water is salty";
```

The output is printed only if the bool variable salty is 1 (true). This is another common way to use bool
variables.
APPLICATION EXAMPLE 5.1  SOLVING A QUADRATIC EQUATION

Problem Statement

Write a computer program capable of solving the quadratic equation

\[ ax^2 + bx + c = 0 \]

The input data is to consist of the values of a, b, and c and is to come from the keyboard. The output is to consist of the values of x and go to the screen.

Solution

RELEVANT EQUATIONS

The quadratic equation has two solutions:

\[ x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]  \hspace{1cm} (5.1)

and

\[ x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \]  \hspace{1cm} (5.2)

Having been assigned to write a computer program, you must do a thorough and correct job. Consider all the possibilities. In the case of the quadratic equation, no real solution may exist, and your computer program must account for this possibility. If \( b^2 - 4ac \) is positive, then Eqns. 5.1 and 5.2 can be used directly to find the solutions \( x_1 \) and \( x_2 \). However, if \( b^2 - 4ac \) is negative, the solutions become:

\[ x_1 = \frac{-b + \sqrt{-(b^2 - 4ac)}}{2a} i \]  \hspace{1cm} (5.3)

and

\[ x_2 = \frac{-b - \sqrt{-(b^2 - 4ac)}}{2a} i \]  \hspace{1cm} (5.4)

where \( i = \sqrt{-1} \)

SPECIFIC EXAMPLE

Consider the following equation:

\[ 2x^2 + 8x + 3 = 0 \]

For this case

\[ a = 2 \]
\[ b = 8 \\
\text{c} = 3 \]

and \( b^2 - 4ac = 40 \), which is positive. The two solutions are:

\[ x_1 = \frac{-8 + \sqrt{8^2 - 4(2)(3)}}{2(2)} = -0.4186 \]

and

\[ x_2 = \frac{-8 - \sqrt{8^2 - 4(2)(3)}}{2(2)} = -3.5814 \]

Consider also the equation:

\[ 15x^2 - 2x + 3 = 0 \]

For this case

\[ a = 15 \\
b = -2 \\
c = 3 \]

and \( b^2 - 4ac = -176 \), which is negative. The two solutions from Eqns. 5.3 and 5.4 are:

\[ x_1 = \frac{-(-2)}{2(15)} + \frac{\sqrt{-((-2)^2 - 4(15)(3))}}{2(15)}i = -0.06667 + 0.44222i \]

and

\[ x_2 = \frac{-(-2)}{2(15)} - \frac{\sqrt{-((-2)^2 - 4(15)(3))}}{2(15)}i = -0.06667 - 0.44222i \]

Just like your calculator, the computer indicates an error and stops executing when it tries to take the square root of a negative number. For this program to execute properly, it should calculate the real and imaginary parts (in this example the real part is \(-0.06667\) and the imaginary part is \(0.44222\)) separately. To calculate the imaginary part, it is necessary to reverse the negative number under the square root to a positive one and then take the square root.

ALGORITHM

Equations 5.1 through 5.4 have been written such that only a single variable appears on the left-hand side of the equations. This form is useful because it fits the form of assignment statements in C++ code. As you write equations for programs, you should get your equations into this form so that you can easily write the source code.
The algorithm (including equations) and a check for taking the square roots of negative numbers is given below.

1. Read the values of a, b, and c from the keyboard.

2. Compute the value of

3. If $b^2 - 4ac$ is positive then

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

and

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

4. Print $x_1$ and $x_2$ to the screen.

5. If $b^2 - 4ac$ is negative then the real part is

$$\text{real} = \frac{-b}{2a}$$

and

$$\text{imaginary} = \frac{-\sqrt{-b^2 - 4ac}}{2a}$$

6. Print the real and imaginary parts in the form real + imaginary * i and real – imaginary * i.

Comments

One can see that the quadratic equation itself never appears in the source code, only the solution to the quadratic equation. In general, you will need to solve your equation or equations before you can begin writing your algorithm or source code. This is considered part of the programming process and is integral to developing a reliable, efficient program. If you solve your equations incorrectly, then your program will give incorrect results even though it is capable of executing without terminating abnormally.

Your program also must be able to handle all possibilities. In this program it was necessary to handle cases where the result is imaginary. Your responsibility as a programmer is to envision all the possibilities and write a program to handle them.

Note that the variable test was used in the source code. This variable was used only for convenience and to simplify the look of the program. It was not necessary for this variable to be used. However, we recommend that you also use variables for convenience and to simplify the look of your programs.
Note also that the directive `#include <cmath>` is necessary for this program because the function `sqrt` is used.

**Source Code**

The source code below has been written from the preceding algorithm.

```cpp
#include <cmath>
#include <iostream>
using namespace std;
int main ()
{
    double i, a, b, c, x1, x2, test, real, imag;
    cout << "Enter the values of a, b, and c (each separated by a space) then press return\n";
    cin >> a >> b >> c;
    test = b * b - 4 * a * c;
    if (test >= 0)
    {
        x1 = (-b + sqrt(test)) / (2 * a);
        x2 = (-b - sqrt(test)) / (2 * a);
        cout << "Real result: \n x1= " << x1 << "\n x2= " << x2 << "\n\n";
    }
    else
    {
        real = -b / (2 * a);
        imag = sqrt(-test) / (2 * a);
        cout << "Imaginary result:\n";
        "x1= " << real << "+" << imag << "i\n";
        "x2= " << real << "-" << imag << "i\n";
    }
}
```

**Output**

Enter the values of a, b, and c (each separated by a space) then press return

Keyboard input: 15 2 3

Imaginary result:

x1 = 0.06667 + 0.44222 i
x2 = 0.06667 - 0.44222 i

**Modification Exercises**

Modify the program to

1. Handle the input of five different equations. Have the user type in five lines when prompted. Each line should contain three coefficients.
2. Read the input from a data file.
3. Read five equations from a data file and print the results to a data file.
CHAPTER 5  DECISION MAKING

Chapter Topics
In this chapter, you will learn how to:
- Create if and if-else structures
- Use logical operators and relational expressions
- Make switch structures
- Use the bool data type
- Write programs with decision making

You will find as you continue learning to program that you will want your programs to make decisions regarding which calculations to perform. For instance, suppose you want to write a program that computes your income tax. Suppose that the percentage of tax is based on your income in the following way:

<table>
<thead>
<tr>
<th>Income</th>
<th>Percent tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–$50,000</td>
<td>20%</td>
</tr>
<tr>
<td>$50,000–100,000</td>
<td>30%</td>
</tr>
<tr>
<td>&gt;$100,000</td>
<td>40%</td>
</tr>
</tbody>
</table>

For your program to correctly compute your tax, it must be able to decide which percent tax applies to your income. In this chapter you will learn how to get your programs to make decisions of this sort.

LESSON 5.1  if CONTROL STRUCTURE (1)—basics

TOPICS
- Simple if statements
- Block if statements
- Controlling program flow
- Relational operators
- Relational expressions

The if statement or if control structure is capable of making decisions in C++. The form of the if statement is fairly simple. A relational expression (meaning an expression that compares the values of two variables, for instance) is contained in the if statement; if the relational expression is true, then the statements within the “true” group are executed. If the relational expression is false, then the statements are not executed.

The program for this lesson has four if statements. This program checks a pass code entered by a user. If the pass code is correct, the user must enter it one more time to verify it. If the pass code is incorrect, the user is given a second opportunity to enter it. If it is incorrect a second time, the user is notified that
access is denied. The correct pass code is built into the program to be 8765. Because a user of the program does not normally see the source code, the pass code is unknown to an invalid user.

Unlike the other programs we have studied, not all the statements in this program are executed (because the if control structure causes some of them to be skipped over). Do your best to trace the actions of the code line by line and see which statements are executed and which are not. Look at the keyboard input to help you follow the flow.

The purpose of this program is to show basic if structures and different operators commonly used with them.

---

### Source Code

```cpp
#include <iostream>
using namespace std;
int main ( )
{
    int i, pcode_entered;
    const int int pcode = 8765;
    cout << "Enter your pass code." << endl;
    cin >> pcode_entered;
    // Relational expression. The result of a relational expression can be regarded as being only true or false. Each relational expression in this program compares the values of pcode_entered and pcode.
    if (pcode_entered < pcode) cout << "Incorrect code " "(too small). Enter it again." << endl;
    if (pcode_entered > pcode) cout << "Incorrect code " "(too large). Enter it again." << endl;
    if (pcode == pcode_entered) cout << "Verify your " "code by entering it again." << endl;
    cin >> pcode_entered;
    if (pcode_entered == pcode)
    {
        cout << "Access approved." << endl;
        cout << "Welcome!" << endl;
    }
    if (pcode_entered != pcode) cout << "Access denied." << endl;
}
```

---

### Output

<table>
<thead>
<tr>
<th>Keyboard input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter your pass code.</td>
<td>8766</td>
</tr>
<tr>
<td>Incorrect code (too large). Enter it again.</td>
<td></td>
</tr>
<tr>
<td>Keyboard input</td>
<td>8765</td>
</tr>
<tr>
<td>Access approved.</td>
<td>Welcome!</td>
</tr>
</tbody>
</table>
Relational expressions. The expression

\[ \text{pcode}_{\text{entered}} < \text{pcode} \]

is a relational expression that compares the values of two arithmetic expressions. A relational expression is a type of logical expression and produces a result of either true or false. Here, it checks whether the value of the variable \( \text{pcode}_{\text{entered}} \) is less than the value of the variable \( \text{pcode} \). Its general syntax is

\[
\text{left_operand} \ \text{relational_operator} \ \text{right_operand}
\]

where the \text{left_operand} and \text{right_operand} can be variables, such as \( \text{pcode}_{\text{entered}} \) in this lesson’s program or any arithmetic expression. The \text{relational_operator} is used to compare the values of two operands. C++ contains six relational operators:

<table>
<thead>
<tr>
<th>Relational operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>(\leq)</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>(\geq)</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

Observe in the table the meaning of \(\neq\). No \(\neq\) symbol is found on the keyboard, so C++ uses \(\neq\).

Simple if statements. A simple if statement can be generalized as

\[
\text{if (expression)} \ \text{statement};
\]

where \text{expression}, represents a logical expression and \text{statement}, is an executable statement. For instance

\[
\text{if (pcode}_{\text{entered}} < \text{pcode}) \ \text{cout} \ << \ "\text{Incorrect code }" \\
\ "(too small). Enter it again." \ << \ \text{endl};
\]

uses \(\text{pcode}_{\text{entered}} < \text{pcode}\) as the expression and \text{cout} \(\ll\) "Incorrect code (too small). Enter it again." \(\ll\) \text{endl}; as the statement. Note that any executable statement can be used as the statement, including if and other control statements. A logical expression produces a result of either true or false. If the logical expression is true, the statement is executed. If the logical expression is false, the statement is not executed. The logical expression within an if statement is called a condition or test expression.

Make sure that you do not put a semicolon immediately after the parentheses enclosing the expression. In other words, if you write

\[
\text{if (expression); \ statement;}
\]
then you will have created two independent statements. The first uses an if conditional that does nothing whether the expression is true or false. The second is a statement that will always be executed (because the if conditional has no effect on it. This is clearly not what you want. Remember, do not put a semicolon separating the test expression from the executable statements.

**Block if statements.** The statement

```cpp
if (pcode_entered == pcode) {statements...}
```

is called a block if statement. If the logical expression (between the two parentheses) is true, the statements in the “true” block (between the two braces) are executed. Otherwise, the entire true block of statements is ignored. The general form of a block if statement is

```cpp
if (expression) {
    executable statement 1;
    executable statement 2;
    ...
}
```

The code block should be indented one tab (at least three spaces) in from the keyword, if. We will see as we go further in this book that indentation is an important way of making your program understandable to others. Although there are no absolute rules on indentation (it is not required by standard C++), it has become common practice in programming.

**Comparing the equality of two integer values.** The equality relational operator `==` (with 2 not 1 = signs) is capable of comparing two values. If the two values are equal, the relational expression using `==` has the value true. If the two values are not equal, the relational expression using `==` has the value false. The equality relational operator is commonly used in test expressions for if control structures.

The `==` operator works very well for integer type comparisons (with integer values on both sides of `==`). However, a major error can occur if you mistakenly use `=` rather than `==`. For instance, in this lesson’s program, we could have mistakenly written just one equal sign and the expression in the form:

```cpp
if (pcode_entered = pcode) cout << "Verify your "
    "code by entering it again." << endl;
```

The result is that pcode_entered is assigned the value of pcode, and the cout statement is executed (for a reason we will see in the next lesson)! This type of error is very difficult to find in a large program. The compiler does not help because it accepts the single equal sign. Consequently, you must find this error on your own.

If the comparison is to a constant, then this form of writing the comparison works well:

```cpp
if (2 == x)
```

because the compiler will not accept

```cpp
if (2 = x)
```
if we accidentally type just one instead of two equal signs. Consequently, the compiler has helped us find the single equal sign problem. In this lesson’s program, the constant is on the left side of the expression in

```c
if (pcode == pcode_entered) cout << "Verify your "
  "code by entering it again." << endl;
```

because pcode is declared to be const int. If we had accidentally typed (with =)

```c
if (pcode = pcode_entered) cout << "Verify your "
  "code by entering it again." << endl;
```

the compiler would have indicated an error for us. In general, be very careful when you use ==. Make sure you do not use just one =.

**Comparing the equality of two real values.** It is not recommended to use == to compare the values of double, long double, or float type variables in most cases, although the C++ compiler will not indicate an error if it is done.

The reason why it is not recommended for real number comparisons is that a typical C++ compiler carries a large number of significant digits for the real variable types (the number depends on the number of bytes used to store the type). If two doubles are compared with == and they differ in only the last significant digit, then the result of a comparison with == is false. For instance, if a = 12.3456789123456789 and b = 12.3456789123456788, then the result of a == b is false although they differ by only $10^{-16}$. Often in real programs, we perform calculations using slightly approximate values and compare them with other approximate values. In many cases, we are interested only if the numbers are nearly or very nearly equal. Therefore, we want a comparison to evaluate to true when the values are very nearly equal. Because == does not do this, we do not use it.

Also, the binary representation of decimal numbers often requires an approximation. For instance, we need many binary bits to represent the simple decimal number 5.3 exactly. Because of this characteristic, numbers that we calculate to be exact with decimal arithmetic may not be exact using binary arithmetic with a limited number of bits. In other words, a calculation that you believe is exact using a hand calculation in decimal may not be exact using binary with a limited number of bits. We avoid the comparison problems caused by this effect by not using == for real type variables.

If we do not use == to compare real numbers, what do we do? One way to compare real values is to use the fabs function and <. For instance, to compare the double values of a and b as listed previously, the following statement would evaluate to true:

```c
if (fabs (a - b) < 1.0e-10)
```

Here, we have somewhat arbitrarily selected the constant $1.0e-10$ as a very small number. In writing your own programs, you must decide how small that number should be based on what you require for your problem.
Comment. The following cout statements were not executed in this lesson’s program for the given input data:

```
cout << "Incorrect code (too small). Enter it again." << endl;
cout << "Verify your code by entering it again." << endl;
cout << "Access denied." << endl;
```
because the relational expressions preceding these cout statements all evaluated to false.

Note that an inefficiency in this lesson’s program is that five relational expressions are executed. Fewer relational expressions (and therefore, fewer comparisons, which is more efficient) are needed when we use if-else control structures. We cover these next, in Lesson 5.2.

LESSON 5.2  if CONTROL STRUCTURE (2)—SIMPLE if-else

TOPICS
- Simple if-else control structures
- The conditional ? : operator

Another form of the if statement is the if-else form. It is used when a group of statements is to be executed if the logical expression is false.

The program in this lesson computes whether or not a company is profitable. The program prompts for revenue and expenses and computes either the profit or loss. An if-else control structure is used to decide which calculations to perform.

In this lesson we introduce the ? : operator. It is the only operator in C++ that is a ternary operator, meaning that three operands are needed for it to be used properly. In this program, we have used all three operands on the right side of an assignment statement. A colon separates two of the operands. We use the ? : operator to help determine the interest on the company’s debt if it is operating at a loss.

Read the source code line by line along with the keyboard input portion of the output section. Determine which statements are executed and which are not executed.
**Description**

**Simple if-else control structure.** The syntax of a simple C++ if-else control structure is
if (expression) {
    executable statement 1a;
    executable statement 1b;
    ...
} else {
    executable statement 2a;
    executable statement 2b;
    ...
}

Executable statements 1a, 1b, ... are part of the “true” block, whereas executable statements 2a, 2b, ... are part of the “false” block. If the expression is true, statements in the true block are executed. If the expression is false, control is transferred to the false block. If the statement block (either true or false) contains more than one statement, the block must be bounded by a pair of braces; otherwise, braces are optional. For example,

```plaintext
if (test >= 0) {
    true block statements...
} else {
    a single statement;
}
```

where the true block contains more than one statement and therefore must be bounded by a pair of braces. However, the false block contains only one statement, so the braces are optional.

If no statements are to be executed in the false block, then the false block may be omitted. The syntax without the false block is as given in Lesson 5.1.

```plaintext
if (expression) {
    executable statement 1a;
    executable statement 1b;
    ...
}
```

Note that the impact of using such a control structure is to cause the execution of one block while bypassing another.

**The ? : conditional operator.** The ? : operator requires three operands. Its form is the following:

```plaintext
expression1 ? expression2 : expression3
```

If expression1 is true, expression2 is evaluated. If expression1 is false, expression3 is evaluated. The value of the entire ? : expression becomes equal to the value of the expression evaluated (either expression2 or expression3).
For this lesson’s program, expression1, loss > 0.0 is true. Therefore, expression2, 0.05 * loss is evaluated, and the value of the right side of the assignment becomes equal to the value of expression2. Thus, interest becomes equal to 0.05 * loss.

As another example, the ? : operator can be used to find the smaller of two numbers. The statement

\[ x = (y < z) \ ? \ y : z; \]

assigns x the value of the smaller of y and z.

Statements using the ? : operator are good shorthand for longer if-else type control structures. Note that for the unevaluated expression no side effects occur.

LESSON 5.3 if CONTROL STRUCTURE (3)—NESTED if-else

TOPIC
- Nested if-else control structures

if-else control structures can be nested, meaning an if-else control structure can be contained within another if-else control structure.

Suppose you are a civil engineer managing a construction project. You may be interested in writing a program that tells you what should be done during a phase of a project during the week at a particular time.

Say that the following 24-hour, 7-day schedule is needed to complete the phase of the project in a timely manner:

Weekdays  
(Days 1–5)  
0:00–9:00 Drive piles  
9:00–24:00 Construct formwork  

Weekends  
(Days 6–7)  
0:00–8:00 Maintain equipment  
8:00–24:00 Pour concrete

This schedule can be used to produce the following algorithm, which illustrates the logic that you would use to respond to a question about what would be done on a particular day at a particular time:

If day = weekday (1-5), then  
if time = 0:00-9:00 Drive piles  
if time = 9:00-24:00 Construct formwork  
If day = weekend (6-7), then
if time = 0:00-8:00 Maintain equipment
if time = 8:00-24:00 Pour concrete

A computer program written in C++ can duplicate the logic of this algorithm. Examine the source code to see how if-else statements can be used to mimic this. Note that these if-else statements are said to be nested.

**Source Code**

```cpp
#include <iostream>
using namespace std;
int main ()
{
    int day;
    double time;
    cout<< "Type the day and time of interest"<< endl;
    cin>> day >> time;

    if (day <= 5)
    {
        if (time <= 9.00)
            cout << "Drive piles" << endl;
        else
            cout << "Construct formwork" << endl;
    }
    else
    {
        if (time <= 8.00)
            cout << "Maintain equipment" << endl;
        else
            cout << "Pour concrete" << endl;
    }
}
```

**Output**

```
Type the day and time of interest
Keyboard input 3 10.00
Construct formwork
```

**Description**

**Nested if-else control structures.** In C++, different levels of if-else control structures can be nested. One can trace which statement blocks are executed by closely following the code. For example, follow these nested structures:
Each nested loop that you create or examine in someone else’s code must be evaluated on a case-by-case basis. A conceptual illustration of the nested if-else control structure used in this lesson is given in Fig. 5.1. Note the branching that is produced by the nesting.

In nested if-else statements, the total number of “ifs” can be greater than or equal to, but not less than the total number of “eloses.” By default, an else clause is associated with the closest previous if statement that has no other else statement. Hand drawn arrows can be used to clarify the pairing of “if” and “else” clauses as shown at the beginning of this lesson’s description. The arrows should not cross in properly written nested if-else statements. If arrows do cross, you must rewrite your if statements so they do not cross.
As if-else control structures become nested, the flow of programs becomes more difficult to trace. Making the program more readable can improve the ease at which it can be followed. The traditional way to make a program readable is to use indentation. We recommend that you indent each pair of if-else statements so that the inner else is paired with the inner if and the outer else is paired with the outer if. An example of this is shown in the indentation of this lesson’s program.

LESSON 5.4 LOGICAL OPERATIONS (1)—LOGICAL OPERATORS

TOPICS
- Using logical operators
- Using logical expressions

Logical operators can be used to connect two relational expressions. For example, the following

```
if (x == 0 && y == 0)
```

is read, “If x is equal to 0 and y is equal to 0.”

Here, the logical operator is && which is read “and.” It connects the two relational expressions, x == 0 and y == 0. The other two logical operators in C++ are || and !. These are used in this lesson’s program. Read the source code and follow the flow of the program line by line. This program performs no useful task; it simply illustrates logical operators.
Logical operators. C++ has three logical operators, &&, ||, and !. The && and || operators are binary operators because they appear between two relational expression operands. The ! operator is unary because it precedes a single relational operand. The formal meanings of these operators follow:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Operation</th>
<th>Operator type</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>Negation</td>
<td>Unary</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>Conjunction</td>
<td>Binary</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
</tbody>
</table>

The logical NOT operator reverses the result of a relational expression. For instance, if x is 5 and y is 0, x is not equal to y, and the expression x == y is false. The logical NOT operator can be used to reverse the false value. Therefore,

! (x == y)

is true.

The logical AND and logical OR operators perform much the same way the words “and” and “or” were used in your very early mathematics classes. The C++ statement

```cpp
if (x > 0 && y >= 0)
```

is read, “If x is greater than 0 and y is greater than or equal to 0,” meaning that the logical expression enclosed in parentheses is true when both x > 0 and y >= 0 are true. On the other hand,

```cpp
if (x == 0 || y == 0)
```

is read, “If x equals 0 or y equals 0.” For the logical expression between the parentheses to be true in this case, only one of

```cpp
x == 0    y == 0
```

need be true.

In the table that follows, we have indicated relational expressions with the symbols A and B. You can use this table to determine the result of a logical expression that includes the two relational expressions. For example, suppose again that x is 5 and y is 0. For a relational expression A being x > 0 and a relational expression B being y >= 0, the relational expression A is true, and the relational expression B is true. This means that we use line 1 in the table. Therefore, A && B is equivalent to true && true and the logical result is true. Looking again at line 1, we can determine the logical result of A || B, IA, and !B.

Similarly, for the relational expressions A being x == 0 and B being y == 0, we have A is false and B is true. This leads us to line 3 of the table. Reading across this line, we get A k B being true. We also can look at it as false k true, which results in true.
Finally, consider relational expression $A$ being $x == y$ (which is false) and relational expression $B$ being $!(x == y)$ (which is equivalent to $!(false)$, which results in true). This also leads us again to line 3 of the table. This line indicates $A && B$ is false. Otherwise, looking at this, $A && B$ is true && false, which is false.

The results of other combinations of relational expressions within logical expressions can be discerned from the table.

|    |    | $A && B$ | $A || B$ | $!A$ | $!B$ |
|----|----|----------|----------|------|------|
| true | true | true     | true     | false| false|
| true | false| false    | true     | false| true |
| false| true | false    | true     | true | true |
| false| false| false    | false    | true | true |

One way to remember the table is to realize that for two expressions being operated on,

1. For $&&$, if one of the relational expressions is false then the result is false.
2. For $||$, if one of the relational expressions is true then the result is true.

LESSON 5.5 LOGICAL OPERATIONS (2)—VALUES OF RELATIONAL EXPRESSIONS AND PRECEDENCE OF RELATIONAL AND LOGICAL OPERATORS

TOPICS
- Precedence of logical operators
- Finding the result of a logical expression

To this point, we have not mentioned that C++ gives relational expressions numerical values just as it gives arithmetic expressions numerical values. If a relational expression is false, C++ gives it a value of 0. If it is true, C++ gives it a value of 1 (which you should recognize as being nonzero). The C++ compiler also operates in the reverse manner. If the value of a relational expression is 0, then it knows the result is false; and if the value of a relational expression is not 0, then the result is true.

Something similar can be done with variables. If the value of a variable is 0, then it can be treated as false; and if the value of a variable is not 0, then it can be treated as true. This works this way because (you will recall) that a single variable can be considered to be an expression.

We saw earlier that C++ has an established order of precedence for arithmetic operators. Similarly, C++ has an established order of precedence for relational and logical operators. In this lesson’s program, we illustrate the precedence of relational and logical operators. Read and follow the source code and annotations. Use the output to help you determine how the program operates.
Description

Precedence and associativity of logical, relational, and arithmetic operators. The precedence and associativity of the logical, relational, and arithmetic operators follow:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Parentheses</td>
<td>L to R</td>
<td>1 (highest)</td>
</tr>
<tr>
<td>++, --</td>
<td>Postincrement</td>
<td>L to R</td>
<td>2</td>
</tr>
<tr>
<td>++, -=</td>
<td>Preincrement</td>
<td>R to L</td>
<td>3</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>L to R</td>
<td>3</td>
</tr>
<tr>
<td>+, -</td>
<td>Positive, negative sign</td>
<td>L to R</td>
<td>3</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Multiplication, division</td>
<td>L to R</td>
<td>4</td>
</tr>
<tr>
<td>+=, -=, *=, /=, %=</td>
<td>Compound assignment</td>
<td>R to L</td>
<td>10</td>
</tr>
<tr>
<td>==, !=, &lt;=, &gt;=</td>
<td>Relational operator</td>
<td>L to R</td>
<td>7</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND</td>
<td>L to R</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>R to L</td>
<td>10 (lowest)</td>
</tr>
</tbody>
</table>

Output

a=4, !a=0
b=-2, !b=0
c=0, !c=1

Answer is TRUE
x=1, !x=0
This table shows that parentheses have the highest order of precedence, followed by unary increment or decrement and logical NOT operators. In general, arithmetic operators, including multiplication, division, addition, and subtraction, have a higher order of precedence than any relational operator. Then come the binary logical operators, in which the logical AND has higher precedence than the logical OR. The assignment operator has the lowest precedence (meaning the assignment takes place after the other operations). The operators work (associativity) left to right—except the pre-increment or pre-decrement, compound assignment, and assignment operators.

For example, assuming $a = 4$, $b = -2$, and $c = 0$, the expression

$$x = (a > b || b > c && a == b)$$

is equivalent to the following expressions (note that the sequence of evaluation is based on the precedence level of each operator; we add parentheses at appropriate locations so that the expression can be grouped and evaluated):

$$x = (a > b || b > c && a == b)$$
$$x = ((a > b) || (b > c) && (a == b))$$
$$x = ((4 > -2) || (-2 > 0) && (4 == -2))$$
$$x = (TRUE || FALSE && FALSE)$$
$$x = (TRUE || FALSE)$$
$$x = (TRUE)$$

which results in true. The value of $x$ becomes 1 (true). If $x$ is true, $!x$ is false, and $!x$ is printed as 0 in this lesson’s program.

The effect of the different precedence of the relational and logical operators on the relational expressions in this lesson’s program is illustrated in Fig. 5.2.

![Figure 5.2](image_url)  
Figure 5.2 Operation of compound logical expression $a > b || b > c && a == b$, from this lesson’s program.
Although not shown in this lesson’s program, a relational expression of the sort \( a > b == c \) is evaluated from left to right since all the operators have equal precedence. For instance, if \( a = 4 \), \( b = -2 \), and \( c = 5 \), this expression evaluates to false. The steps in evaluation are

\[
\begin{align*}
a > b & \text{ is true, giving this expression a value of 1} \\
1 == c & \text{ is false.}
\end{align*}
\]

**Logical value of a single variable.** The logical value of a single variable is false if the variable has a value of 0 and true if the value is nonzero. This is illustrated in Fig. 5.3a. For example, in this lesson, the logical value of \( c \) is false, since \( c \) is equal to 0, but the logical values of \( a \) and \( b \) are true, since \( a \) (which is 4) and \( b \) (which is -2) are nonzero. Also, the values of \( !a \) and \( !b \) are false and therefore printed out as 0, whereas the value of \( !c \) is printed out as 1. This is shown in Fig. 5.3b.

![Figure 5.3](image)

**(a)** True or false result of integer values. (b) Integer values for \(!true\) and \(!false\).

**LESSON 5.6** if-else-if AND switch CONTROL STRUCTURES

**TOPICS**

- Using if-else-if control structures
- Using switch statements
- Comparing if-else-if and switch control structures

The if-else control structure executes one of two statement blocks. Frequently in programming, though, we want to execute one of a number (three, four, or more) of statement blocks. This is usually most conveniently done with an if-else-if or a switch control structure. Both these structures contain multiple statement blocks and have the feature that when one of the blocks is executed, the others are bypassed.

Two source codes are given for this lesson. They perform the same tasks but in different ways. The first source code uses an if-else-if control structure, and the second source code uses a switch control structure. Read and follow the flow of both codes using the output as a guide.
Source Code 1

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

int main() {
    int option;
    cin >> option;
    if (option == 1) {
        cout << "Attend meeting\n";
    } else if (option == 2) {
        cout << "Debug program\n";
    } else if (option == 3) {
        cout << "Write documentation\n";
    } else {
        cout << "Do nothing\n";
    }
}
```

Source Code 2

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

int main() {
    int option;
    cin >> option;
    switch (option) {
    case 1: cout << "Attend meeting\n"; break;
    case 2: cout << "Debug program\n"; break;
    case 3: cout << "Write documentation\n"; break;
    default: cout << "Do nothing\n";
    }
}
```

Output from Both Source Codes

Please type 1, 2, or 3

Keyboard input 2
Debug program
if-else-if control structure. An if-else-if control structure shifts program control, step by step, through a series of statement blocks. Control stops at the relational expression that is true and executes the corresponding statement block. After execution of that statement block, control shifts to the end of the control structure. If none of the relational expressions is true, the final statement block is executed. In this lesson’s program the value of option was read in to be 2, so the first statement block was not executed. Because the relational expression option == 2 was true, the second statement block was executed. The third and fourth statement blocks were bypassed, and control transferred to the end of the control structure.

The form of the if-else-if control structure is

```c
if (relational_expression_1)
{
    statement_block_1
}
else if (relational_expression_2)
{
    statement_block_2
}
.
.
.
.
else if (relational_expression_n-1)
{
    statement_block_n-1
}
else
{
    statement_block n
}
```

Figure 5.4 illustrates the if-else-if control structure for this lesson’s program. Note the branching that occurs due to the different values of option.

switch control structure. A switch statement or switch control structure commonly is constructed like the if-else-if control structure. It also is used to transfer control. Its syntax is

```c
switch (expression)
{
    case constant1:
        statement1a
        statement1b
        ...
    case constant2:
        statement2a
        statement2b
        ...
```
where the expression must be enclosed in a pair of parentheses and must result in an integer type value when the program flow enters the switch block. A switch block must be bounded by a pair of braces. The terms constant1, constant2, and so on are integer type constant expressions. Note that all constant expressions are followed by colons. All the constant expressions must be unique, meaning that none can have the same value as another constant expression. Although not required, it is common that the last case type line is the keyword default. If no constant matches the value of the expression, the statements in the default case are executed. The default case is optional. If no default case is given and no constant expression matches the expression value, the entire switch block is ignored.

Figure 5.4 if-else-if control structure for Lesson 5.6’s program. Compare to switch (Fig. 5.5) and nested if-else (Fig. 5.1) control structure illustrations.

Figure 5.5 shows the switch control structure for this lesson’s program. Compare this figure to Figs. 5.1 and 5.4. Note the similarities between the if-else-if, nested if-else, and switch control structures. In all the cases illustrated, the control structure has chosen a single block of code to execute and bypassed the others.

The keyword case can be used only in a switch control structure. It is used to form a label called a case label. A case label is a constant followed by a colon. The label does not affect the execution of the statement that follows it. In switch control structures, C++ looks for a match between the switch expression and the expression in a case label. C++ then executes the statement sequence following the
matching case label. For instance, for the form shown previously, if the value of the switch expression matches constant1, then the program flow is transferred to case constant1 and statement1a, statement1b, and so forth are executed. Because the switch control structure can search only for equality, it differs from the if-else-if control structure, which can use other relational operators.

![Diagram showing switch control structure for Lesson 5.6's program. Note the importance of the break statement in controlling program flow for the switch control structure. Compare to if-else-if (Fig. 5.4) and nested if-else (Fig. 5.1) control structure illustrations.](image)

A break statement in a switch control structure terminates execution of the smallest enclosing switch statement. The keyword break terminates (which means to send control to the point of the closing brace) the switch structure. We will see that break statements have other uses, which operate similarly in that they cause control to pass to a closing brace.

Often the last statement for each case is the break statement because it terminates the process and exits switch. If no break statement is used, then the statements in the next case are executed. For example, in the following code,

```c++
switch (option)
{
    case 1: cout << "Entering case 1\n";
            break;
    case 2: cout << "Entering case 2\n";
    case 3: cout << "Entering case 3\n";
            break;
}
```
if option is 1, then "Entering case 1" will be displayed on the screen. If option is 3, "Entering case 3" will be displayed. However, if option is 2, both "Entering case 2" and "Entering case 3" will be displayed because C++ first finds a match between the switch expression and a case label. Execution then continues, line by line, until a break or the end of the block (indicated by a closing brace) is encountered. This is because a statement label has no effect on the statement that follows it. A case statement label serves only as a marker to which control can be sent. The program flow when a break statement is missing is shown in Fig. 5.6.

![Flow of program control for the switch control structure of Lesson 5.6's program if no break statement were given for the case 2 block.](image)

The keyword default is a special label used only for switch control structures. In the event that none of the case label constants agrees with the switch expression, control passes to the default labeled statement sequence. Because the label default is a keyword, it is not considered to be a user-defined label.

**Nested switch control structures.** We can nest switch control structures. A nested switch control structure could take the following form:

```
switch (outer_expression)
{
    case constant_outer1:
```
switch (inner_expression)
{
    case constant_inner1:
        statement inner_1a
        statement inner_1b
        ...
        ...
    case constant_inner2:
        statement inner_2a
        statement inner_2b
        ...
        ...
}

case constant_outer2:
    statement outer_2a
    statement outer_2b
    ...
    ...

case constant_outer3:
    statement outer_3a
    statement outer_3b
    ...
    ...
}

An illustration of a nested switch structure is given in Fig. 5.7.

Figure 5.7  Nested switch control structure with break statements.
LES3ON 5.7  THE bool DATA TYPE

TOPICS

• Using the bool data type
• Input and output for bool data

C++ supports another data type that we have not yet discussed— the bool data type, named after
mathematician George Boole who worked in the area of logic. Unlike an int variable that can hold any
integer value (within memory limits), a variable declared to be bool can contain one of only two values
which are most conveniently regarded as 0 or 1. The 0 or 1 value of a bool variable can be interpreted to
mean false or true, fail or pass, off or on, or any two states we are interested in. This type of
representation can be useful for storing such information as the results of material testing among other
things. For instance, if a steel piece has met the requirements for use in a machine a bool variable can
be used to represent pass/fail for the test results.

In this lesson’s program, we illustrate how bool variables can help represent water quality. In this case,
we want to indicate whether the water is salty, hard, acidic, has a good taste, and whether the user
receives home service for this water. A bool type variable is useful because there are only two states for
each of these. In other words, we classify the water as being salty or not salty, hard or not hard, acidic or
not acidic, having a good taste or not a good taste, and a user receives the water or does not receive it.
In the program, we create bool variables salty, hard, acidic, good_taste, and have_service. For instance,
if we set the value of hard to be 0 (meaning false) then the water is not hard; if we set hard to 1
(meaning true) then the water is hard.

For the variables salty, hard, and acidic there are scientific tests that determine which state the water is
in. If the sodium level is greater than 4000 mg/l, the water can be considered salty. If the calcium is
greater than 40 mg/l and the magnesium is greater than 20 mg/l, the water can be considered hard. If
the pH is less than 7, the water is acidic. As a result, we have double type variables in the program that
represent the sodium, calcium, magnesium, and pH levels of the water. By checking the values of these
variables, we can determine in which state each of the bool variables should be. This means we can
assign the result of a relational expression to a bool type variable, and the bool variable indicates the
state. Read the program to see how we do this.

Source Code
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

int main ( )
{
    bool salty, hard, acidic, good_taste, have_service;
    double sodium, Ca, Mg, pH;

    ifstream infile["C:\\water.dat"];
    infile >> sodium >> Ca >> Mg >> pH;
    salty = (sodium > 4000);
    hard = (Ca > 40 && Mg > 20);
    acidic = (pH < 7);
bool data type. The bool data type is a C++ data type that can be used to represent quantities that can have one of two states. To declare variables to be type bool, we use the keyword bool followed by a comma separated list of variable names in a manner similar to declaring int or double variables. For instance, to declare the variables salty, hard, and acidic to be bool, the following declaration is used:

```cpp
bool salty, hard, acidic;
```

The states of a bool variable are most conveniently thought of as being 1 or 0, where 1 is equivalent to true and 0 is equivalent to false. Commonly, bool type variables occupy one byte of memory.

Uses for bool variables.

- As we showed in this lesson’s program, bool variables can represent characteristics of materials. We have declared bool variables salty, hard, acidic, and good_taste to represent characteristics of water. Two states are appropriate for each of these variables because the water can be thought to be salty or not salty, hard or soft, acidic
or basic, good tasting or bad tasting. This means that the variables are ideally suited to being bool type.

- The current status of events can be represented with bool type variables when a state can be considered to be either existent or not existent. For instance, in this lesson’s program, we have used the bool variable have_service. Having service is an event that either exists or not. When this variable has a value of true, the user has service. When it is false, the user does not have service. Other events that either exist or not, such as a switch being on or off, can also be represented by bool variables.

- In general, bool variables can be used for any quantity that can be described by one of two states.

### Assigning values to bool variables.

- One way to assign a value to a bool variable is to write a logical expression on the right side of an assignment statement. For instance, in this lesson’s program we used:

  ```
  salty = (sodium > 4000);
  hard = (Ca > 40 && Mg > 20);
  acidic = (pH < 7);
  ```

  In each case, if the logical expression is true, the value 1 is assigned to the bool variable. On the other hand, if the logical expression is false, the value 0 is assigned to the bool variable. This is illustrated in Fig. 5.8.

- We can also assign integer values to bool variables. Although not shown in this lesson’s program, we could have used the assignment statements:

  ```
  good_taste = 15;  // (the nonzero value, 15, indicates true and causes the value 1 to be assigned to good_taste)
  ```

  or

  ```
  good_taste = 0;  // (the zero value indicates false)
  ```

  In the first of these, a nonzero integer has been assigned to a bool variable, which is equivalent to assigning a true logical expression. Any nonzero integer (negative or positive) produces the result of assigning a value of 1 to the bool variable even if the nonzero integer is not 1! In the second of these, zero has been assigned to a bool variable, which is equivalent to assigning a false logical expression. Only zero assigns a value of 0 to a bool variable.

  In addition to using assignment statements to accomplish this, we can use cin statements as we have done in this lesson’s program with

  ```
  cin >> good_taste;
  ```
The user in our example entered the integer 1 (although any nonzero value would have been equivalent) for the value of good_taste. Again, this is the same as assigning a true logical expression to the good_taste variable. Had the user entered 0, it would have been the same as assigning a false logical expression to the variable good_taste. This is also illustrated in Fig. 5.8.

- We can also assign the keywords true or false to bool variables. Although not shown in this lesson’s program, we could have used the assignment statement:

```cpp
good_taste = true;
```

or

```cpp
good_taste = false;
```

Similarly a user can enter the words true or false in response to a cin statement as we did in this lesson’s program in response to

```cpp
cin >> boolalpha >> have_service;
```

to assign a value to the variable have_service. Note that the boolalpha manipulator is needed for C++ to properly interpret the true or false word input as shown in Fig. 5.8.

![Diagram showing how bool variables are assigned values](image)

**Figure 5.8 Assigning values to bool variables.**

**Assigning bools to int variables.** We could have used in this lesson’s program code of the sort:

```cpp
int i;
i = salty;
```
after the variable salty had been initialized. Because salty is false (sodium < 4000) it has a value of 0. The int variable i then gets the value 0. Had salty been true, i would have been given a value of 1. This is illustrated in Fig. 5.9.

Printing the values of bool variables. In using cout to print bool variable values, we can choose to use the boolalpha manipulator or not. If we do not use bool-alpha, bool variable values are printed as 1 (for true) or 0 (for false). For instance,

```cpp
cout << "Water composition" << " salty " << salty << " hard " << hard << " acidic " << acidic;
```

prints the line

```
Water composition  salty 0  hard 1  acidic 0
```

If we use the boolalpha manipulator, the values are printed as “true” or “false.” For instance,

```cpp
cout << boolalpha << "Water composition" << " salty " << salty << " hard " << hard << " acidic " << acidic;
```

prints the line

```
Water composition  salty false  hard true  acidic false
```

These are illustrated in Fig. 5.9.

Using bool variables with if statements. Although not shown in this lesson’s program, we could have used a line such as:

```cpp
if (salty) cout << "The water is salty";
```

The output is printed only if the bool variable salty is 1 (true). This is another common way to use bool variables.
APPLICATION EXAMPLE 5.1  SOLVING A QUADRATIC EQUATION

Problem Statement

Write a computer program capable of solving the quadratic equation

\[ ax^2 + bx + c = 0 \]

The input data is to consist of the values of a, b, and c and is to come from the keyboard. The output is to consist of the values of \( x \) and go to the screen.

Solution

RELEVANT EQUATIONS

The quadratic equation has two solutions:

\[ x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]  \hspace{1cm} (5.1)

and

\[ x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \]  \hspace{1cm} (5.2)

Having been assigned to write a computer program, you must do a thorough and correct job. Consider all the possibilities. In the case of the quadratic equation, no real solution may exist, and your computer program must account for this possibility. If \( b^2 - 4ac \) is positive, then Eqns. 5.1 and 5.2 can be used directly to find the solutions \( x_1 \) and \( x_2 \). However, if \( b^2 - 4ac \) is negative, the solutions become:

\[ x_1 = \frac{-b}{2a} + \frac{\sqrt{-(b^2 - 4ac)}}{2a}i \] \hspace{1cm} (5.3)

and

\[ x_2 = \frac{-b}{2a} - \frac{\sqrt{-(b^2 - 4ac)}}{2a}i \] \hspace{1cm} (5.4)

where \( i = \sqrt{-1} \)

SPECIFIC EXAMPLE

Consider the following equation:

\[ 2x^2 + 8x + 3 = 0 \]

For this case

\[ a = 2 \]
\[ b = 8 \]
\[ c = 3 \]
and \( b^2 - 4ac = 40 \), which is positive. The two solutions are:

\[
x_1 = \frac{-8 + \sqrt{8^2 - 4(2)(3)}}{2(2)} = -0.41886
\]

and

\[
x_2 = \frac{-8 - \sqrt{8^2 - 4(2)(3)}}{2(2)} = -3.58114
\]

Consider also the equation:

\[
15x^2 - 2x + 3 = 0
\]

For this case

\[
a = 15 \]
\[
b = -2 \]
\[
c = 3
\]
and \( b^2 - 4ac = -176 \), which is negative. The two solutions from Eqns. 5.3 and 5.4 are:

\[
x_1 = \frac{-(-2)}{2(15)} + \frac{\sqrt{\left((-2)^2 - 4(15)(3)\right)}}{2(15)}i = -0.06667 + 0.44222i
\]

and

\[
x_2 = \frac{-(-2)}{2(15)} - \frac{\sqrt{\left((-2)^2 - 4(15)(3)\right)}}{2(15)}i = -0.06667 - 0.44222i
\]

Just like your calculator, the computer indicates an error and stops executing when it tries to take the square root of a negative number. For this program to execute properly, it should calculate the real and imaginary parts (in this example the real part is \(0.06667\) and the imaginary part is \(0.44222\)) separately. To calculate the imaginary part, it is necessary to reverse the negative number under the square root to a positive one and then take the square root.

**ALGORITHM**

Equations 5.1 through 5.4 have been written such that only a single variable appears on the left-hand side of the equations. This form is useful because it fits the form of assignment statements in C++ code. As you write equations for programs, you should get your equations into this form so that you can easily write the source code.
The algorithm (including equations) and a check for taking the square roots of negative numbers is given below.

1. Read the values of \( a \), \( b \), and \( c \) from the keyboard.

2. Compute the value of

3. If \( b^2 - 4ac \) is positive then

\[
x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}
\]

and

\[
x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]

4. Print \( x_1 \) and \( x_2 \) to the screen.

5. If \( b^2 - 4ac \) is negative then the real part is

\[
real = -\frac{b}{2a}
\]

and

\[
imaginary = \frac{\sqrt{-(b^2 - 4ac)}}{2a}
\]

6. Print the real and imaginary parts in the form \( \text{real} + \text{imaginary} \ast i \) and \( \text{real} - \text{imaginary} \ast i \).

Comments

One can see that the quadratic equation itself never appears in the source code, only the solution to the quadratic equation. In general, you will need to solve your equation or equations before you can begin writing your algorithm or source code. This is considered part of the programming process and is integral to developing a reliable, efficient program. If you solve your equations incorrectly, then your program will give incorrect results even though it is capable of executing without terminating abnormally.

Your program also must be able to handle all possibilities. In this program it was necessary to handle cases where the result is imaginary. Your responsibility as a programmer is to envision all the possibilities and write a program to handle them.

Note that the variable test was used in the source code. This variable was used only for convenience and to simplify the look of the program. It was not necessary for this variable to be used. However, we recommend that you also use variables for convenience and to simplify the look of your programs.
Note also that the directive `#include <cmath>` is necessary for this program because the function `sqrt` is used.

**Source Code**

The source code below has been written from the preceding algorithm.

```cpp
#include <cmath>
#include <iostream>
using namespace std;
int main()
{
    double i, a, b, c, x1, x2, test, real, imag;
    cout << "Enter the values of a, b, and c (each separated by a space) then press return\n";
    cin >> a >> b >> c;
    test = b * b - 4 * a * c;
    if (test >= 0)
    {
        x1 = (-b + sqrt(test)) / (2 * a);
        x2 = (-b - sqrt(test)) / (2 * a);
        cout << "Real result: \nx1= " << x1 << "\nx2= " << x2 << "\n";
    }
    else
    {
        real = -b / (2 * a);
        imag = sqrt(-test) / (2 * a);
        cout << "Imaginary result: \n" "x1= " << real << '+' << imag << 'i ' "nx2= " << real
            "-imag " << imag << 'i ";
    }
}
```

**Output**

Enter the values of a, b, and c (each separated by a space) then press return

Keyboard input 15 -2 3
Imaginary result:

x1 = 0.06667 + 0.44222 i
x2 = 0.06667 - 0.44222 i

**Modification Exercises**

Modify the program to

1. Handle the input of five different equations. Have the user type in five lines when prompted. Each line should contain three coefficients.
2. Read the input from a data file.
3. Read five equations from a data file and print the results to a data file.