Table of Contents

[Module 1 - C++ Fundamentals 4](#_Toc203507578)

[1.3) Object-Oriented Programming 4](#_Toc203507579)

[Encapsulation 4](#_Toc203507580)

[Polymorphism 5](#_Toc203507581)

[Inheritance 5](#_Toc203507582)

[1.4) A First Simple Program 6](#_Toc203507583)

[The First Sample Program Line by Line 6](#_Toc203507584)

[1.5) A Second Simple Program 9](#_Toc203507585)

[1.6 Using an Operator 10](#_Toc203507586)

[1.7 Reading Input from the Keyboard 11](#_Toc203507587)

[Some Output Options 12](#_Toc203507588)

[Another Data Type 13](#_Toc203507589)

[Project 1-1: Converting Feet to Meters 14](#_Toc203507590)

[1.8) Two Control Statements 14](#_Toc203507591)

[The if statement 14](#_Toc203507592)

[The for loop 16](#_Toc203507593)

[1.9) Using Blocks of Code 17](#_Toc203507594)

[Semicolons and Position 18](#_Toc203507595)

[Project 1-2: Generating a Table of Feet to Meter Conversions 19](#_Toc203507596)

[1.10) Introducing Functions 21](#_Toc203507597)

[1.11) The C++ Keywords 23](#_Toc203507598)

[1.12) Identifiers 23](#_Toc203507599)

[Module 2 - Introducing Data Types and Operators 24](#_Toc203507600)

[2.1) The C++ Data Types 24](#_Toc203507601)

[Integers 25](#_Toc203507602)

[Characters 27](#_Toc203507603)

[Floating-Point Types 28](#_Toc203507604)

[The bool type 29](#_Toc203507605)

[void 31](#_Toc203507606)

[Project 2-1) Talking to Mars 31](#_Toc203507607)

[2.2) Literals 32](#_Toc203507608)

[Hexadecimal and Octal Literals 33](#_Toc203507609)

[String Literals 33](#_Toc203507610)

[Character Escape Sequences 33](#_Toc203507611)

[2.3) A Closer Look at Variables 34](#_Toc203507612)

[Initializing a Variable 34](#_Toc203507613)

[Dynamic Initialization 35](#_Toc203507614)

[Operators 35](#_Toc203507615)

[2.4) Arithmetic Operators 35](#_Toc203507616)

[Increment and Decrement 36](#_Toc203507617)

[2.5) Relation and Logical Operators 38](#_Toc203507618)

[Project 2-2) Construct an XOR Logical Operation 40](#_Toc203507619)

[2.6) The Assignment Operator 41](#_Toc203507620)

[2.7) Compound Assignments 41](#_Toc203507621)

[2.8 Type Conversion in Assignments 42](#_Toc203507622)

[Expressions 43](#_Toc203507623)

[2.9) Type Conversions in Expressions 43](#_Toc203507624)

[Converting to and from bool 43](#_Toc203507625)

[2.10) Casts 43](#_Toc203507626)

[2.11) Spacing and Parentheses 45](#_Toc203507627)

[Project 2-3) Compute the Regular Payments on a Loan 45](#_Toc203507628)

[Module 3 - Program Control Statements 47](#_Toc203507629)

[3.1) The if Statement 47](#_Toc203507630)

[The Conditional Expression 48](#_Toc203507631)

[Nested ifs 49](#_Toc203507632)

[The if-else-if Ladder 50](#_Toc203507633)

[3.2) The switch Statement 52](#_Toc203507634)

[Nested switch Statements 55](#_Toc203507635)

[Project 3-1) Start Building a C++ Help System 56](#_Toc203507636)

[3.3) The for loop 58](#_Toc203507637)

[Some Variations on the for Loop 59](#_Toc203507638)

[Missing Pieces 61](#_Toc203507639)

Self-test answers can be found at [www.osborne.com](http://www.osborne.com)

# Module 1 - C++ Fundamentals

## 1.3) Object-Oriented Programming

In an object-oriented language, you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.

To support the principles of object-oriented programming, all OOP languages, including C++, have three traits in common: **encapsulation**, **polymorphism**, and **inheritance**. Let’s examine each.

### Encapsulation

***Encapsulation*** is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be bound together in such a way that a self-contained ***black-box*** is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an ***object*** is created. In other words, an object is a device that supports encapsulation.

*Sidenote: The term method which is used in C# and Java is sometimes also used when referring to a C++ function*

Within an object, code or data or both may be ***private*** to that object or ***public***. Private code or data is known to an accessible by only another part of the object. That is, private code or data cannot be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program can access it even though it is defined within an object.

C++’s basic unit of encapsulation is the ***class***. A class defines the form of an object. It specifies both the data and the code that will operate on that data. C++ uses a class specification to construct ***objects***. Objects are instances of a class. Thus, a class is essentially a set of plans that specifies how to build an object.

The code and data that constitute a class are called ***members*** of the class. Specifically, ***member variables***, also called ***instance variables***, are the data defined by the class. ***Member functions*** are the code that operates on that data. ***Function*** is C++’s term for a subroutine.

### Polymorphism

***Polymorphism*** is the quality that allows one interface to access a general class of actions. A simple example of polymorphism is found in the steering wheel of an automobile. The steering wheel (the interface) is the same no matter whether your actual steering mechanism is used. That is, the steering wheel works the same whether your car has manual steering, power steering, or rack-and-pinion steering. Thus, turning the steering wheel left causes the car to go left no matter what type of steering is used. The benefit of the uniform interface is, of course, that once you know how to operate the steering wheel, you can drive any type of car.

The same principle can also apply to programming. For example, consider a ***stack*** (which is a first-in, last-out list). You might have a program that requires three different types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. In this case, the algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names.

More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods”. This means that it is possible to design a generic interface to a group of related activities. Polymorphism helps reduce complexity by allowing the same interface to specify a ***general class of action***. It is the compiler’s job to select the ***specific action*** (that is, method) as it applies to each situation.

### Inheritance

***Inheritance*** is the process by which one object can acquire the properties of another object. This is important because it supports the concept of hierarchical classification. Without the use of hierarchies, each object would have to explicitly define all of its characteristics. Using inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

## 1.4) A First Simple Program

Now it’s time to begin programming. Let’s start by compiling and running the short sample C++ program shown here:

1. #include <iostream>

2. using namespace std;

3.

4. int main(){

5.

6.     cout << "Hello world!";

7.

8.     return 0;

9. }

***Source code*** is the human-readable form of the code. It is stored in a text file. ***Object code*** is the executable form of the program created by the compiler.

### The First Sample Program Line by Line

Although **sample.cpp** is quite short, it includes several key features that are common to all C++ programs. Let’s closely examine each part of the program. The program begins with the lines:

1. /\*

2. This is a simple C++ program

3. Call this file sample.cpp

4. \*/

This is a *comment*, the contents of a comment are ignored by the compiler. The purpose of a comment is to describe or explain the operation of a program to anyone reading its source code. In the case of this comment, it identifies the program. In C++, there are two types of comments. First, a ***multiline comment*** which begins with a /\* and ends only when a \*/ is encountered. Anything between these two comments is completely ignored by the compiler. Multiline comments may be one or more lines long. Second, ***single-line comment*** which is a # symbol followed by the text, i.e.:

1. #include <iostream>

The C++ language defines several *headers*, which contain information that is either necessary or useful to your program. This program requires the header **iostream**, which supports the C++ I/O system. This header is provided with your compiler. A header is included in your program using the **#include**directive.

The next line in the program is:

1. using namespace std;

This tells the compiler to use the **std** *namespace*. A namespace creates a declarative region in which various program elements can be placed. Elements declared in one namespace are separate from elements declared in another. Namespaces help in the organization of large programs. The **using** statement informs the compiler that you want to use the **std** namespace. This is the namespace in which the entire Standard C++ library is declared. By using the **std** namespace, you simplify access to the standard library.

The next line in the program is:

1. // A C++ program begins at main().

This line shows you the second type of comment available in C++: the *single-line comment*. Single-line comments begin with // and stop at the end of the line.

The next line, as the preceding comment indicates, is where program execution begins.

1. int main()

All C++ programs are composed of one or more functions. Every C++ function must have a name, and the only function that any C++ program **must** include is the one shown here, called **main( )**. The **main( )** function is where program execution begins and (most commonly) ends. (Technically speaking, a C++ program begins with a call to **main( )** and, in most cases, ends when **main( )** returns.) The opening curly brace on the line that follows **main( )** marks the start of the **main( )** function code. The **int** that precedes **main( )** specifies the type of data returned by **main( )**.

The next line in the program is:

1. cout << “C++ is power programming.”;

This is a **console output** statement. It causes the message **C++ is power programming.** to be displayed on the screen. It accomplishes this by using the output operator **<<**. The << operator causes whatever expression is on its right side to be output to the device specified on its left side. **cout** is a predefined identifier that stands for console output and generally refers to the computer’s screen. Thus, this statement causes the message to be output to the screen.

The next line in the program is:

1. return 0;

This line **terminates main( )** and causes it to return the value 0 to the calling process (which is typically in the operating system). For most operating systems, a return value of 0 signifies that the program is terminating normally. Other values indicate that the program is terminating because of some error. **Return** is one of C++’s keywords, and it uses to return a value from a function.

The closing curly brace at the end of the program formally concludes the program.

Error Message vs Warning Message:

* **Error messages** report things that are unequivocally wrong in your program, such as forgetting a semicolon.
* **Warning messages** point out suspicious but technically correct code.

It is up to you, the programmer, then to decide whether the suspicion is justified.

## 1.5) A Second Simple Program

A *variable* is a named memory location that can be assigned a value. Further, the value of a variable can be changed during the execution of a program. That is, the content of a variable is changeable, not fixed.

The following program creates a variable called **length**, gives it the value 7, and then displays the message **The length is 7** on the screen.

1. // Using a variable

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     int length;     // this declares a variable

9.

10.     length = 7;     // this assigns 7 to length

11.

12.     cout << "The length is ";

13.     cout << length;           // this displays 7

14.

15.     return 0;

16. }

This program introduces two new concepts. First, the statement:

8. int length;     // this declares a variable

declares a variable called **length** of type integer. In C ++, all variables must be declared before they are used. Further, the type of values that the variable can hold must also be specified. This is called the *type* of the variable. In this case, **length** may hold integer values. These are whole number values whose range will be at least -32,768 through 32,767.

The second new feature is found is the next line of code:

10.     length = 7;     // this assigns 7 to length

As the comment suggests, this assigns the value 7 to **length**. In C++, the assignment operator is the single equal sign. It copies the value on its right side onto the variable on its left. After the assignment, the variable **length** will contain the number 7.

The following statement displays the value of **length**:

13.     cout << length;           // this displays 7

In general, if you want to display the value of a variable, simply put it on the right side of << in a **cout** statement. In this specific case, because **length** contains the number 7, it is this number that is displayed on the screen.

## 1.6 Using an Operator

C++ supports a full range of arithmetic operators enable you to manipulate numeric values used in a program. They include + (addition), - (subtraction), \* (multiplication), and / (division).

These operators work in C++ just like they do in algebra. The following program uses the \* operator to compute the area of a rectangle given a length and width:

1. // Using an operator

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main (){

7.     int length;  // this declares a variable

8.     int width;   // this declares another variable

9.     int area;    // this does, too

10.

11.     length = 7;   // this assigns 7 to length

12.     width = 5;    // this assigns 5 to width

13.

14.     area = length \* width; // compute area

15.

16.     cout << "The area is ";

17.     cout << area; // this displays 35

18.

19.     return 0;

20. }

In this program, there is actually no need for the variable **area**. For example, the program can be rewritten like this:

1. // A simplified version of the area program.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main (){

7.     int length;  // this declares a variable

8.     int width;   // this declares another variable

9.     int area;    // this does, too

10.

11.     length = 7;   // this assigns 7 to length

12.     width = 5;    // this assigns 5 to width

13.

14.

15.     cout << "The area is ";

16.     cout << length \* width; // this displays 35

17.

18.     return 0;

19. }

In this version, the area is computed in the **cout** statement by multiplying **length** by **width**. The result is then output to the screen.

One more point before we move on: It is possible to declare two or more variables using the same declaration statement. Just separate their names by commas. For example, **length**, **width**, and **area** could have been declared like this:

int length, width, area; // all declared using one statement

## 1.7 Reading Input from the Keyboard

The previous example would be much more useful if it would prompt the user for the dimensions of the rectangle, allowing the user to enter them using the keyboard.

To enable the user to enter data into a program from the keyboard, you will use the **>>** operator. This is like the C++ input operator. To read from the keyboard, use this general form:

cin >> var;

Here, **cin** is another predetermined identifier. It sands for *console input* and is automatically supplied by C++. By default, **cin** is linked to the keyboard, although it can be redirected to other devices. The variable that receives the input is specified by *var*.

Here is the program rewritten to allow the user enter the dimensions of the rectangle:

1. /\*

2.         An interactive program that computes the area of a rectangle

3. \*/

4.

5. #include <iostream>

6. using namespace std;

7.

8. int main(){

9.     int length; // this declares a variable

10.     int width;  // this declares another variable

11.

12.     cout << "Enter the length: ";

13.     cin >> length; // input the length

14.

15.     cout << "Enter the width: ";

16.     cin >> width; // input the width

17.

18.     cout << "The area is: ";

19.     cout << length \* width; // display the area

20.

21.     return 0;

22. }

Here is a sample run:

A black background with white text

AI-generated content may be incorrect.

### Some Output Options

So far, we have been using the simplest type of **cout** statements. However, **cout** allows much more sophisticated output statements. The first of two useful techniques is using multiple output operators within the same **cout** statement to output more than one piece of information:

1. cout << “The area is “ << length \* width;

Second, up to this point, there has been no occasion to advance output to the next line – that is, to execute a carriage return – linefeed sequence. In C++, the carriage return—linefeed is generated using the *newline* character. To put a newline character into a string, use this code: \**n** (a backslash followed by a lowercase *n*). i.e.:

1. /\*

2.     This program demonstrates the \n code, which generates a new line.

3. \*/

4.

5. #include <iostream>

6. using namespace std;

7.

8. int main(){

9.     cout << "one\n";

10.     cout << "two\n";

11.     cout << "three";

12.     cout << "four";

13.

14.     return 0;

15. }

The program produces the following output:

one

two

threefour

### Another Data Type

In the preceding programs, variables of type **int** were used. However, a variable of type **int** can only hold whole numbers. Thus, it cannot be used when a fractional component is required. For example, an **int** variable can hold the value 18, but not the value 18.3. To allow numbers with fractional components, C++ defines two main flavors of floating-point types: **float** and **double**, which represent single- and double-precision values, respectively. Of the two, **double** is probably the most commonly used. To declare a variable of type **double**, use the statement similar to that shown here:

double result;

Here, **result** is the name of the variable, which is of type **double**. Because **result** has a floating-point type, it can hold values such as 88.56, 0.033, or -107.03. To better understand the difference between **int** and **double**, try the following program:

1. /\*

2.     This program illustrates the differences between int and double.

3. \*/

4.

5. #include <iostream>

6. using namespace std;

7.

8. int main(){

9.     int ivar;   // this declares an int variable

10.     double dvar; // this declares a floating-point variable

11.

12.     ivar = 100; // assigns ivar the value 100

13.

14.     dvar = 100.0; // assigns dvar the value 100.0

15.

16.     cout << "Original value of ivar: " << ivar << "\n";

17.     cout << "Original value of dvar: " << dvar << "\n";

18.

19.     cout << "\n"; // print a blank line

20.

21.     // now, divide both by 3

22.     ivar = ivar / 3;

23.     dvar = dvar / 3.0;

24.

25.     cout << "ivar after division: " << ivar << "\n";

26.     cout << "dvar after division: " << dvar << "\n";

27.

28.

29.     return 0;

30. }

The output from this program is shown here:

Original value of ivar: 100

Original value of dvar: 100

ivar after division: 33

dvar after division: 33.3333

As you can see, when **ivar** is divided by 3, a whole-number division is performed and the outcome is 33—the fractional component is lost. However, when **dvar** is divided by 3, the fractional component is preserved.

### Project 1-1: Converting Feet to Meters

In this project, we will create a program that converts feet to meters. The program prompts the user for the number of feet. It then displays the value converted into meters.

A meter is equal to approximately 3.28 feet. Thus, we need to use floating-point data. To perform the conversion, the program declares two **double** variables. One will hold the number of feet, and the second will hold the conversion to meters.

## 1.8) Two Control Statements

Inside a function, execution proceeds from one statement to the next, top to bottom. It is possible, however to alter this flow through the use of the various *program control statements* supported by C++.

### The if statement

You can selectively execute part of a program through the use of C++’s conditional statement: the **if**. Its simplest form is shown here:

if(condition) statement;

where *condition* is an expression that is evaluated to be either true or false. In C++, true is nonzero and false is zero. If the condition is true, then the statement will execute. If it is false, then the statement will not execute. For example, the following fragment displays the phrase **10 is less than 11** on the screen because 10 is less than 11.

if(10 < 11) cout << “10 is less than 11”;

However, consider the following:

if(10 > 11) cout << “this does not display”;

In this case, 10 is not greater than 11, so the **cout** statement is not executed. Of course, the operands inside the **if** statement need not be constants. They can also be variables.

C++ defines a full complement of relation operators that can be used in a conditional expression. They are shown here

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| < | Less than |
| <= | Less than or equal |
| > | Greater than |
| >= | Greater than or equal |
| == | Equal to |
| != | Not equal |

Notice that the test for equality is the double equal sign.

Here is a program that illustrates the **if** statement:

1. // Demonstrate the if.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.     int a, b, c;

8.

9.     a = 2;

10.     b = 3;

11.

12.     if(a < b){

13.         cout << "a is less than b\n";

14.     }

15.

16.     // this won't display anything

17.     if(a == b){

18.         cout << "you won't see this\n";

19.     }

20.

21.     cout << "\n";

22.

23.     c = a - b; // c contains -1

24.     cout << "c contains -1\n";

25.     if(c >= 0 ){

26.         cout << "c is non-negative\n";

27.     }

28.     if(c < 0){

29.         cout << "c is negative\n";

30.     }

31.

32.     cout << "\n";

33.

34.     c = b - a; // c now contains 1

35.     cout << "c contains 1\n";

36.     if( c >= 0){

37.         cout << "c is non-negative";

38.     }

39.     if(c < 0){

40.         cout << "c is negative\n";

41.     }

42.

43.     return 0;

44. }

The output generated by this program is shown here:

a is less than b

c contains -1

c is negative

c contains 1

c is non-negative

### The for loop

You can repeatedly execute a sequence of code by creating a *loop*. C++ supplies a powerful assortment of loop constructs. The one we will look at here is the **for** loop. The simplest form of the **for** loop is shown here:

for(initialization; condition; increment) statement;

Here, *initialization* sets a loop control variable to an initial value. *condition* is an expression that is tested each time the loop repeats. As long as *condition* is true (nonzero), the loop keeps running. The *increment* is an expression that determines how the loop control variable incremented each time the loop repeats.

The following program demonstrates the **for**. It prints the numbers 1 through 100 on the screen:

1. // A program that illustratse the for loop.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.     int count;

8.

9.     for(count = 1; count <= 100; count++){

10.         cout << count << " ";

11.     }

12.

13.     return 0;

14. }

In the loop, **count** is initialized to 1. Each time the loop repeats, the condition

count <= 100

is tested. If it is true, the value is output and **count** is increased by one. When **count** reaches a value greater than 100, the condition becomes false, and the loop starts running.

## 1.9) Using Blocks of Code

A *code block* is a grouping of two or more statements. This is done by enclosing the statements between opening and closing curly braces. Once a block of code is created, it becomes a logical unit that can be used any place that single statement can. For example, a block can be a target of the **if** and **for** statements. Consider this **if** statement:

1. if( w < h){

2. v = w \* h;

3. w = 0;

4. }

The key point here is that whenever you need to logically link two or more statements, you do so by creating a block. Code blocks allow many algorithms to be implemented with greater clarity and efficiency.

Here is a program that uses a block of code to prevent division by zero:

1. // Demonstrate a block of code

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.     double result, n, d;

8.

9.     cout << "Enter value: ";

10.     cin >> n;

11.

12.     cout << "Enter divisor: ";

13.     cin >> d;

14.

15.     // the target of this if is a block

16.     if(d !=0 ){

17.         cout << "d does not equal zero so division is OK" << "\n";

18.         result = n/d;

19.         cout << n << " / " << d << " is " << result;

20.     }

21.

22.     return 0;

23. }

Here is a sample run:

A black background with white text

AI-generated content may be incorrect.

### Semicolons and Position

In C++, the semicolon signals the end of a statement. A block is *not* terminated with a semicolon. Since a block is a group of statements, with a semicolon after each statement, it makes sense that a block is not terminated by a semicolon; instead, the end of the block is indicated by the closing brace.

C++ does not recognize the end of the line as the end of a statement—only a semicolon terminates a statement. For this reason, it does not matter where on a line you put a statement. For example, to C++

x = y;

y = y + 1;

cout << x << “ “ << y;

is the same as

x = y; y = y + 1; cout << x << “ “ << y;

Furthermore, the individual elements of a statement can also be put on separate lines. For example, the following is perfectly acceptable:

cout << “This is a long line. The sum is : “ << a + b + c +

d + e +f;

Breaking long lines in this fashion is often used to make programs more readable. It can also help prevent excessively long lines from wrapping.

### Project 1-2: Generating a Table of Feet to Meter Conversions

This project demonstrates the **for** loop, the **if** statement, and code blocks to create a program that displays a table of feet-to-meters conversions. The table begins with 1 foot and ends at 100 feet. After every 10 feet, a blank line is output. This is accomplished through the use of a variable called **counter** that counts the number of lines that have been output. Pay special attention to its use.

1. /\*

2.     Project 1-2

3.

4.     This program displays a conversion table of feet to meters.abort

5.

6.     Call this program FtoMTable.cpp

7. \*/

8.

9. #include <iostream>

10. using namespace std;

11.

12. int main(){

13.     double f; // holds the length in feet

14.     double m; // holds the conversion to meters

15.     int counter;

16.

17.     counter = 0;

18.

19.     for(f = 1.0; f <= 100.0; f++){

20.         m = f / 3.28; // convert to meters

21.         cout << f << " feet is " << m << " meters.\n";

22.

23.         counter++;

24.

25.         // every 10th line, print a blank line

26.         if(counter == 10){

27.             cout << "\n"; // output a blank line

28.             counter = 0; // reset the line counter

29.         }

30.     }

31.

32.     return 0;

33. }

1 feet is 0.304878 meters.

2 feet is 0.609756 meters.

3 feet is 0.914634 meters.

4 feet is 1.21951 meters.

5 feet is 1.52439 meters.

6 feet is 1.82927 meters.

7 feet is 2.13415 meters.

8 feet is 2.43902 meters.

9 feet is 2.7439 meters.

10 feet is 3.04878 meters.

11 feet is 3.35366 meters.

12 feet is 3.65854 meters.

13 feet is 3.96341 meters.

14 feet is 4.26829 meters.

15 feet is 4.57317 meters.

16 feet is 4.87805 meters.

17 feet is 5.18293 meters.

18 feet is 5.4878 meters.

19 feet is 5.79268 meters.

20 feet is 6.09756 meters.

21 feet is 6.40244 meters.

22 feet is 6.70732 meters.

23 feet is 7.0122 meters.

24 feet is 7.31707 meters.

25 feet is 7.62195 meters.

26 feet is 7.92683 meters.

27 feet is 8.23171 meters.

28 feet is 8.53659 meters.

29 feet is 8.84146 meters.

30 feet is 9.14634 meters.

On your own, try changing the program so that it prints a blank line every 25 lines.

1. /\*

2.     Project 1-2

3.

4.     This program displays a conversion table of feet to meters.abort

5.

6.     Call this program FtoMTable.cpp

7. \*/

8.

9. #include <iostream>

10. using namespace std;

11.

12. int main(){

13.     double f; // holds the length in feet

14.     double m; // holds the conversion to meters

15.     int counter;

16.

17.     counter = 0;

18.

19.     for(f = 1.0; f <= 100.0; f++){

20.         m = f / 3.28; // convert to meters

21.         cout << f << " feet is " << m << " meters.\n";

22.

23.         counter++;

24.

25.         // every 10th line, print a blank line

26.         if(counter == 25){

27.             cout << "\n"; // output a blank line

28.             counter = 0; // reset the line counter

29.         }

30.     }

31.

32.     return 0;

33. }

1 feet is 0.304878 meters.

2 feet is 0.609756 meters.

3 feet is 0.914634 meters.

4 feet is 1.21951 meters.

5 feet is 1.52439 meters.

6 feet is 1.82927 meters.

7 feet is 2.13415 meters.

8 feet is 2.43902 meters.

9 feet is 2.7439 meters.

10 feet is 3.04878 meters.

11 feet is 3.35366 meters.

12 feet is 3.65854 meters.

13 feet is 3.96341 meters.

14 feet is 4.26829 meters.

15 feet is 4.57317 meters.

16 feet is 4.87805 meters.

17 feet is 5.18293 meters.

18 feet is 5.4878 meters.

19 feet is 5.79268 meters.

20 feet is 6.09756 meters.

21 feet is 6.40244 meters.

22 feet is 6.70732 meters.

23 feet is 7.0122 meters.

24 feet is 7.31707 meters.

25 feet is 7.62195 meters.

26 feet is 7.92683 meters.

27 feet is 8.23171 meters.

28 feet is 8.53659 meters.

29 feet is 8.84146 meters.

30 feet is 9.14634 meters.

## 1.10) Introducing Functions

A C++ program is constructed from building blocks called *functions*. A function is a subroutine that contains one or more C++ statements.

Each function has a name, and this name is used to call the function. To call a function, simply specify its name in the source code of your program, followed by parentheses. For example, assume some function named **MyFunc**. To call **MyFunc**, you would write

MyFunc();

When a function is called, program control is transferred to that function, and the code contained within the function is executed. When the function’s code ends, control is transferred back to the caller. Thus, a function performs a task for other parts of a program.

Some functions require one or more *arguments*, which you pass when the function is called. Thus, an argument is a value passed to a function. Arguments are specified between the opening and closing parentheses when a function is called. For example, if **MyFunc()** requires an integer argument, then the following calls **MyFunc()** with the value 2:

MyFunc(2)

When there are two or more arguments, they are separated by commas. In this book, the term *argument* *list* will refer to comma-separated arguments. Remember, not all functions require arguments. When no argument is needed, the parentheses are empty.

A function can return a value to the calling code. Not all functions return values, but many do. The value returned by a function can be assigned to a variable in the calling code by placing the call to the function on the right side of an assignment statement. For example, if **MyFunc()** returned a value, it could be called as shown here:

x = MyFunc()

The statement works as follows. First, **MyFunc()** is called. When it returns, its return value is assigned to x. You can also use a call to a function in an expression. For example,

x = MyFunc(2) + 10;

In this case, the return value from **MyFunc()** is added to 10, and the result is assigned to x. In general, whenever a function’s name is encountered in a statement, it is automatically called so that its return value can be obtained.

To review: an *argument* is a value passed into a function. A *return value* is data that is passed back to the calling code.

Here is a short program that demonstrates how to call a function. It uses one of C++’s built-in functions, called **abs()**, to display the absolute value of a number. The **abs()** function takes one argument, converts it into its absolute value, and returns the result.

1. // Use the abs() function.

2.

3. #include <iostream>

4. #include <cstdlib>

5. using namespace std;

6.

7. int main(){

8.     int result;

9.

10.     result = abs(-10);

11.

12.     cout << result;

13.

14.     return 0;

15. }

Notice one other thing about the preceding program; it includes the header **cstdlib**. This is the header required by **abs()**. Whenever you use a built-in function, you must include its header.

In general, there are two types of functions that will be used by your programs. The first type is written by you, and **main()** is an example of this type of function.

The second type of function is provided by the compiler. The **abs()** function used by the preceding program is an example. Programs that you write will generally contain a mix of functions that you create and those supplied by the compiler.

## 1.11) The C++ Keywords

## 1.12) Identifiers

In C++, an *identifier* is a name assigned to a function, variable, or other user-defined item. Identifiers can be from one to several characters long. Variable names can start with any letter of the alphabet or an underscore. Next comes a letter, a digit, or an underscore. The underscore can be used to enhance the readability of a variable name, as in **line\_count**. Uppercase and lowercase are seen as different; that is, to C++, **myvar** and **MyVar** are separate names. There is one important identifier restriction: you cannot use any of the C++ keywords as identifier names. In addition, predefined identifiers such as **cout** are also off limits.

Here are some examples of valid identifiers:

|  |  |  |  |
| --- | --- | --- | --- |
| Test | x | y2 | Maxlncr |
| up | \_top | my\_var | simpleInterest23 |

# Module 2 - Introducing Data Types and Operators

## 2.1) The C++ Data Types

At the core of the C++ type system are the seven basic data types shown here:

|  |  |
| --- | --- |
| **Type** | **Meaning** |
| char | Character |
| wchar\_t | Wide character |
| int | Integer |
| float | Floating point |
| double | Double floating point |
| bool | Boolean |
| void | Valueless |

C++ allows certain of the basic types to have modifiers preceding them. A modifier alters the meaning of the base type so that it more precisely fits the needs of various situations. The data type modifiers are listed here:

signed

unsigned

long

short

The modifiers **signed**, **unsigned**, **long**, and **short** can be applied to int. The modifiers **signed** and **unsigned** can be applied to the **char** type. The type **double** can be modified by **long**. Table 2-1 shows all valid combinations of the basic types and the type modifiers. The table also shows t he guaranteed minimum range for each type as specified by the ANSI/ISO C++ standard.

|  |  |
| --- | --- |
| **Type** | **Minimal Range** |
| char | -127 to 127 |
| unsigned char | 0 to 255 |
| signed char | -127 to 127 |
| int | -32,767 to 32,767 |
| unsigned int | 0 to 65,535 |
| signed int | Same as **int** |
| short int | -32,767 to 32,767 |
| unsigned short int | 0 to 65,535 |
| signed short int | Same as **short int** |
| long int | -2,147,483,647 to 2,147,438,647 |
| signed long int | Same as **long int** |
| unsigned long int | 0 to 4,294,967,295 |
| float | 1E-37 to 1E+37, with six digits of precision |
| double | 1E-37 to 1E+37 with ten digits of precision |
| long double | 1E-37 to 1E+37 with ten digits of precision |

**Table 2-1** All Numeric Data Types Defined by C++ and Their Minimum Guaranteed Ranges as Specified by the ANSI/ISO C++ standard

It is important to understand that minimum ranges shown in Table 2-1 are just that: *minimum* ranges. A C++ is free to exceed one or more of these minimums, and most compilers do. Thus, the ranges of the C++ data types are implementation dependent. For example, on computers that use two’s complement arithmetic (which is nearly all), an integer will have a range of at least -32,768 to 32,767. In all cases, however, the range of a **short int** will have a subrange of **int**, which will be a subrange of a **long int**. The same applies to **float**, **double**, and **long double**. In this usage, the term *subrange* means a range narrower than or equal to. Thus, an **int** and **long int** can have the same range, but an **int** cannot be larger than a **long int**.

Since C++ specifies only the minimum range a data type must support, you should check your compiler’s documentation for the actual ranges provided.

### Integers

Variables of type **int** hold integer quantities that do not require fractional components. Variables of this type are often used for controlling loops and conditional statements, and for counting. Because they don’t have fractional components, operations on **int** quantities are much faster than they are on floating-point types.

Because integers are so important to programming, C++ defines several varieties. As shown in table 2-1, there are short, regular, and long integers. Furthermore, there are signed and unsigned versions of each. A signed integer can hold both positive and negative values. Thus, the use of **signed** on integers is redundant (but allowed) because the default declaration assumes a signed value. An unsigned integer can hold only positive values. To create an unsigned integer, use the **unsigned** modifier.

The difference between signed and unsigned integers is in the way the high-order bit of the integer is interpreted. If a signed integer is specified, then the C++ compiler will generate code that assumes that the high-order bit of an integer is to be used as a *sign flag*. If the sign flag is 0, then the number is positive; if it is 1, then the number is negative. Negative numbers are almost always represented using the *two’s complement* approach. In this method, all bits in the number (except the sign flag) are reversed, and then 1 is added to this number. Finally, the sign flag is set to 1.

For example, given 8-bit positive integer 34:

+ 34 = 0 0 1 0 0 0 1 0

We flip the sign flag (left-most bit) from 0 (positive) to negative (1) and flip the other bits:

1 1 0 1 1 1 0 1

Next, we add 1:

1 1 0 1 1 1 0 1

+ 1

-----------------------

-34 = 1 1 0 1 1 1 1 0

Signed integers are important for a great many algorithms, but they have only half the absolute magnitude of their unsigned relatives. For example, assuming a 16-bit integer, here is 32,767

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

For a signed value, if the high-order bit were set to 1, the number would then be interpreted as -1 (assuming the two’s complement format). However, if you declared this to be an **unsigned int**, then when the high-order bit was set to 1, the number would become 65,535.

To understand the difference between the way that signed and unsigned integers are interpreted by C++, try this short program:

1. /\* This program shows the difference between signed and unsigned integers. \*/

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     short int i; // a signed short integer.

9.     short unsigned int j; // an unsigned short integer

10.

11.     j = 60000;

12.     i = j;

13.     cout << i << " " << j;

14.

15.     return 0;

16. }

The output from this program is shown here:

-5536 60000

These values are displayed because the bit pattern that represents 60,000 as a short unsigned integer is interpreted as -5,536 as short signed integer (assuming 16-bit short integers).

C++ allows a shorthand notation for declaring **unsigned**, **short**, or **long** integers. You can simply use the word **unsigned**, **short**, or **long**, without the **int**. The **int** is implied. For example, the following two statements both declare unsigned integer variables:

unsigned x;

unsigned int y;

### Characters

Variables of type **char** hold 8-bit ASCII characters such as A, z, or G, or any other 8-bit quantity. To specify a character, you must enclose it between single quotes. Thus, this assigns X to the variable **ch**:

char = ch;

ch = ‘X‘;

You can output a **char** value using a **cout** statement. For example, this line outputs the value in **ch**:

cout << “This is ch: “ << ch;

This results in the following output:

This is ch: X

The **char** type can be modified with **signed** or **unsigned**. Technically, whether **char** is signed or unsigned by default is implementation-defined. However, for most compilers **char** is signed. In these environments, the use of **signed** on **char** is also redundant. For the rest of this book, it will be assumed that **char**s are signed entities.

The type **char** can hold values other than just the ASCII character set. It can also be used as a “small” integer with the range typically from -128 through 127 and can be substituted for an **int** when the situation does not require larger numbers. For example, the following program uses a **char** variable to control the loop that prints the alphabet on the screen:

1. // This program displays the alphabet using char letter.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main() {

7.

8.     char letter;

9.

10.     for(letter = 'A'; letter <= 'Z'; letter++){  // initializing at A, if letter less than or equal to Z we increment letter to next letter then print letter.

11.         cout << letter << " " ;

12.     }

13.

14.     return 0;

15. }

The **for** loop works because the character A is represented inside the computer by the value 65, and the values for the letters A to Z are in sequential, ascending order. Thus, **letter** is initially set to ‘A’. Each time through the loop, **letter** is incremented. Thus, after the first iteration, **letter** is equal to ‘B’.

The type **wchar\_t** holds characters that are part of large character sets. As you may know, many human languages, such as Chinese, define a large number of characters, more than will fit within the 8 bits provided by the **char** type. The **wchart\_t** type was added to C++ to accommodate this situation. While we won’t be making use of **wchar\_t** in this book, it is something that you will want to look into if you are tailoring programs for the international market.

### Floating-Point Types

Variables of the types **float** and **double** are employed either when a fractional component is required or when your application requires very large or small numbers. The difference between a **float** and a **double** variable is the magnitude of the largest (and smallest) number that each one can hold. Typically, a **double** can store a number approximately ten times larger than a **float**.

Of the two, **double** is the most commonly used. One reason this is that many of the math functions in the C++ function library use **double** values. For example, the **sqrt( )** function returns a **double** value that is the square root of its **double** argument. Here, **sqrt( )** is used to compute the length of the hypotenuse given the lengths of the two opposing sides.

1. /\*

2.     Use the Pythagorean theorem to find

3.     the length of the hypotenuse given

4.     the lengths of the two opposing sides.

5. \*/

6.

7. #include <iostream>

8. #include <cmath> // The <cmath> header is needed for the sqrt() function.

9. using namespace std;

10.

11. int main(){

12.

13.     double x,y,z;

14.

15.     x = 5.0;

16.     y = 4.0;

17.

18.     z = sqrt(x\*x + y\*y);

19.

20.     cout << "Hypotenuse is: " << z ;

21.

22.

23.     return 0;

24. }

25.

The output from the program is shown here:

Hypotenuse is 6.40312

One other point about the preceding example: Because **sqrt( )** is part of the C++ standard function library, it only requires the standard header <**cmath**>, which is included in the program.

The **long double** type lets you work with very large or small numbers. It is most useful in scientific programs. For example, the **long double** type might be useful when analyzing astronomical data.

### The bool type

The **bool** type is a relatively recent addition to C++. It stores Boolean (that is, true/false) values. C++ defines two Boolean constants, **true** and **false**, which are the only two values that a **bool** value can have.

Before continuing, it is important to understand how true and false are defined by C++. One of the fundamental concepts in C++ is that any nonzero value is interpreted as true and zero is false. This concept is fully compatible with the **bool** data type because when used in a Boolean expression, C++ automatically converts any nonzero value into **true**. It automatically converts zero into false. The reverse is also true; when used in a non-Boolean expression, **true** is converted into 1, and **false** is converted into zero. The convertibility of zero and nonzero values into their Boolean equivalents is especially important when using control statements, as you will see in Module 3.

Here is a program that demonstrates the **bool** type:

1. // Demonstrate bool values.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     bool b;

9.

10.     b = false;

11.     cout << "b is " << b << "\n";

12.

13.     b = true;

14.     cout << "b is " << b << "\n";

15.

16.     // a bool value can control the if statement

17.     if(b){

18.         cout << "This is executed.\n";

19.     }

20.

21.     b = false;

22.     if(b){

23.         cout << "This is not executed.\n";

24.     }

25.

26.     // outcome of a relational operator is a true/false value

27.     cout << "10 > 9 is " << (10 > 9) << "\n";

28.

29.     return 0;

30. }

31.

The output generated by this program is shown here:

b is 0

b is 1

This is executed.

10 > 9 is 1

There are three interesting things to notice about this program. First, as you can see, when a **bool** value is output using **cout**, 0 or 1 is displayed. As you will see later in this book, there is an output option that causes the words “false” and “true” to be displayed.

Second, the value of a **bool** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

if(b == true) ……

Third, the outcome of a relational operator, such as <, is a Boolean value. This is why the expression **10 > 9** displays the value 1. Further, the extra set of parentheses around  **10 > 9** is necessary because the << operator has a higher precedence than the >.

### void

The **void** type specifies a valueless expression. This probably seems strange now, but you will see how **void** is used later in this book.

### Project 2-1) Talking to Mars

At its closest point to Earth, Mars is approximately 34,000,000 miles away. Assuming there is someone on Mars that you want to talk with, what is the delay between the time a radio signal leaves Earth and the time it arrives on Mars? This project creates a program that answers this question. Recall that radio signals travel at the speed of light, approximately 186,000 miles per second. Thus, to compute the delay, you will need to divide the distance by the speed of light. Display the delay in terms of seconds and also in minutes.

1. /\*

2.     Project 2-1

3.

4.     Talking to Mars

5. \*/

6.

7. #include <iostream>

8. using namespace std;

9.

10. int main(){

11.

12.     // variables used in program

13.     double distance, lightspeed, delay, delay\_in\_min;

14.

15.     // distancce and lightspeed initial values

16.     distance = 34000000.0; // 34,000,000 miles

17.     lightspeed = 186000;   // 186,000 per second

18.

19.     delay = distance / lightspeed;

20.     cout << "Time delay when talking to Mars: " << delay << " seconds.\n";

21.

22.     delay\_in\_min = delay / 60;

23.     cout << "This is " << delay\_in\_min << " minutes.";

24.

25.     return 0;

26. }

The program’s output is shown below:

Time delay when talking to Mars: 182.796 seconds.

This is 3.04659 minutes.

## 2.2) Literals

*Literals* refer to fixed, human-readable values that **cannot** be altered by the program. For example, the value 101 is an integer literal. Literals are also commonly referred to as *constants*. For the most part, literals and their usage are so intuitive that they have been used in one form or another by all the preceding sample programs. Now the time has come to explain them formally.

C++ literals can be of any of the basic data types. The way each literal is represented depends upon its type. As explained earlier, *character* literals are enclosed between single quotes. For example, ‘a’ and ‘%’ are both character literals.

*Integer* literals are specified as numbers without fractional components. For example, 10 and -100 are integer constants. *Floating-point* literals require the use of the decimal point followed by the number’s fractional component. For example, 11.123 is a floating-point constant. C++ also allows you to use scientific notation for floating-point numbers.

All literal values have a data type, but this fact raises a question. As you know, there are several different types of integers, such as **int**, **short int**, and **unsigned long int**. There are also three different floating-point types: **float**, **double**, and **long double**. The question is: How does the compiler determine the type of a literal? For example, is 123.23 a **float** or a **double**? The answer to this question has two parts. First, the C++ compiler automatically makes certain assumptions about the type of a literal and, second, you can explicitly specify the type of a literal, if you like.

By default , the C++ compiler fits an integer literal into the smallest compatible data type that will hold it, beginning with **int**. Therefore, assuming 16-bit integers, 10 is **int** by default, but 103,000 is **long**. Even though the value 10 could be fit into a **char**, the compiler will not do this because it means crossing type boundaries.

By default, floating-point literals are assumed to be **double**. Thus, the value 123.23 is of type **double**.

For virtually all programs you will write as a beginner, the compiler defaults are perfectly adequate. In cases where the default assumption that C++ makes about a numeric literal is not what you want, C++ allows you specify the exact type of numeric literal by using a suffix. For floating-point types, if you follow the number with an *F*, the number is treated as a **float**. If you follow it with an *L*, the number becomes a **long double**. For integer types, the *U* suffix stands for **unsigned** and the *L* for **long**. (Both the *U* and the *L* must be used to specify an **unsigned long**.) Some examples are shown here:

|  |  |
| --- | --- |
| **Data Type** | **Examples of Constants** |
| int | 1 123 21000 -234 |
| long int | 35000L -34L |
| unsigned int | 10000U 987U 40000U |
| unsigned long | 12323UL 900000UL |
| float | 123.23F 4.34e-3F |
| double | 23.23 123123.33 -0.9876324 |
| long double | 1001.2L |

### Hexadecimal and Octal Literals

As you probably know, in programming it is sometimes easier to use a number system based on 8 or 16 instead of 10. The number system based on 8 is called *octal*, and it uses the digits 0 through 7. In octal, the number 10 is the same as 8 in decimal. The base-16 number system is called *hexadecimal* and uses digits 0 through 9 plus the letters *A* through *F*, which stand for 10, 11, 12, 13, 14, and 15. For example, the hexadecimal number 10 is 16 in decimal. Because of the frequency with which these two number systems are used, C++ allows you to specify integer literals in hexadecimal or octal instead of decimal. A hexadecimal literal must begin with 0x (a zero followed by an x.) An octal literal begins with a zero. Here are some examples:

hex = 0xFF; // 255 in decimal

oct = 011; // 9 in decimal

### String Literals

C++ supports one other type of literal in addition to those of the predefined data types: the string. A *string* is a set of characters enclosed by double quotes. For example, “this is a test” is a string. Keep in mind one important fact: although C++ allows you to define string constants, it does not have a built-in string data type. Instead, strings are supported in C++ as character arrays. (C++ *does*, however, provide a string type in its class library.)

### Character Escape Sequences

C++ provides the *character escape sequences*, sometimes referred to as *backslash character constants* (*table 2-3 on page 57*), so that you can enter them into a program. As you can see, the **\n** that you have been using is one of the escape sequences.

The following sample program illustrates a few of the escape sequences:

1. // Demonstrate some escape seqeunces.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     cout << "one\ttwo\tthree\n"; //  \t is for tabs and \n is for newline

9.     cout << "123\b\b45";         //  \b is for backsapce

10.

11.     return 0;

12. }

The output is shown here:

one two three

145

## 2.3) A Closer Look at Variables

As you learned, variables are declared using this form of statement:

*type var-name;*

where *type* is the data type of the variable and *var-name* is its name. When you create a variable, you are creating a instance of its type. Thus, the capabilities of a variable are determined by its type. Furthermore, the type of a variable cannot change during its lifetime.

### Initializing a Variable

You can assign a value to a variable at the same time it is declared. To do this, follow the variables’ name with an equal sign and the value being assigned. This is called a *variable initialization*. Its general form is shown here:

*type var = value;*

Here, *value* is the value that is given to *var* when *var* is created.

When declaring two or more variables of the same type using a comma separated list, you can give one or more of those variables an initial value. For example,

int a, b = 8, c = 19, d; // b and c have initializations

In this case, only **b** and **c** are initialized.

### Dynamic Initialization

Although the preceding examples have used only constants as initializers, C++ allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the volume of a cylinder given the radius of its base and its height:

1. // Demonstrate dynamic initialization.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     double radius = 4.0, height = 5.0;

9.

10.     //dynamically initialize radius

11.     double volume = 3.1416 \* radius \* radius \*height; // volume is dynamically initialized at runtime.

12.

13.     cout << "Volume is " << volume;

14.

15.     return 0;

16. }

The output is shown below:

Volume is 251.328

Here, three local variables—**radius**, **height**, and **volume**---are declared. The first two, **radius** and **height**, are initialized by constants. However, **volume** is initialized dynamically to the volume of the cylinder. The key point here is that the initialization expression can use any element valid at the time of the initialization, including calls to functions, other variables, or literals.

### Operators

C++ has four general classes of operators: *arithmetic*, *bitwise*, *relational*, and *logical*. C++ also has several additional operators that handle certain special situations. This chapter will examine every operator besides the bitwise and special operators which will be examined later.

## 2.4) Arithmetic Operators

The +, -, \*, and / all work the same way in C++ as they do in algebra. These can be applied to any built-in numeric data type. They can also be applied to values of type **char**.

The **%** (modulus) operator yields the remainder of an integer division. Recall that when / is applied to an integer, any remainder will be truncated; for example 10/3 will equal 3 in integer division. You can obtain the remainder of this division using the **%** operator. For example, 10 % 3 is 1. In C++, the **%** can be applied only to integer operands; it cannot be applied to floating-point types.

The following program demonstrates the modulus operator.

1. // Demonstrates the modulus operator.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     int x, y;

9.

10.     x = 10;

11.     y = 3;

12.     cout << x << " / " << y << " is " << x / y << " with a remainder of " << x % y << "\n";

13.

14.     x = 1;

15.     y = 2;

16.     cout << x << " / " << y << " is " << x / y << "\n" << x << " % " << y << " is " << x % y;

17.

18.     return 0;

19. }

The output is shown here:

10 / 3 is 3 with a remainder of 1

1 / 2 is 0

1 % 2 is 1

### Increment and Decrement

The ++ and the – are increment and decrement operators. Let’s begin by reviewing precisely what the increment and decrement operators do.

The increment operator adds 1 to its operand, and the decrement operator subtracts 1. Therefore,

x = x + 1;

is the same as

x++;

and

x = x – 1;

is the same as

--x;

Both the increment and decrement operators can either precede (prefix) or follow (postfix) the operand. For example,

x = x + 1;

can be written as

++x; // prefix form

or as

x++; // postfix form

In this example, there is no difference whether the increment is applied as prefix or a postfix. However, when an increment or decrement is used as part of a larger expression, there is an important difference. When an increment or decrement precedes its operand, C++ will perform the operation prior to obtaining the operand’s value for use by the rest of the expression. If the operator follows its operand, then C++ will obtain the operand’s value before incrementing or decrementing it. Consider the following:

x = 10;

y = ++x;

In this case, **y** will be set to 11. However, if the code is written as

x = 10;

y = x++;

then **y** will be set to 10. In both cases, **x** will still set to 11; the difference is when it happens. There are significant advantages in being able to control when the increment or decrement operation takes place.

The precedence of the arithmetic operators is shown here:

|  |  |
| --- | --- |
| **Highest** | ++ -- |
|  | - (unary minus |
|  | \* / % |
| **Lowest** | + - |

Operators on the same precedence level are evaluated by the compiler from left to right. Parentheses may be used to alter the order of evaluation; parentheses force an operation, or a set of operations, to have a higher precedence level.

## 2.5) Relation and Logical Operators

In the terms *relational operator* and *logical operator*, *relational* refers to the relationships that values can have with one another, and *logical* refers to the ways in which true and false values can be connected together. Since the relational operators produce true or false results, they often work with the logical operators.

The relational and logical operators are shown in Table 2-4 (page 65). In C++, the outcome of a relational or logical expression produces a **bool** result. That is, the outcome of a relational or logical expression is either **true** or **false**.

The operands for a relational operator can be nearly any type as long as they can be meaningfully compared. Any expression other than one that has a **void** result can be used.

Here is a program that demonstrates several of the relational and logical operators.

1. // Demonstrate the relational and logical operators.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.     int i, j;

8.     bool b1, b2;

9.

10.     i = 10;

11.     j = 11;

12.     if(i < j){

13.         cout << "i < j\n";

14.     }

15.     if(i <= j){

16.         cout << "i <= j\n";

17.     }

18.     if(i != j){

19.        cout << "i != j\n";

20.     }

21.     if(i == j){

22.        cout << "i == j\n";

23.     }

24.     if(i >= j){

25.        cout << "i >= j\n";

26.     }

27.     if(i > j){

28.        cout << "i > j\n";

29.     }

30.

31.     b1 = true;

32.     b2 = false;

33.     if(b1 && b2){

34.         cout << "this won't execute!";

35.     }

36.     if(!(b1 && b2)){

37.         cout << "!(b1 && b2) is true\n";

38.     }

39.     if(b1 || b2){

40.         cout << "b1 || b2 is true\n";

41.     }

42.

43.     return 0;

44. }

The output from the program is shown here:

i < j

i <= j

i != j

!(b1 && b2) is true

b1 || b2 is true

Both the relational and logical operators are lower in precedence than the arithmetic operators. This means that an expression like 10 > 1 + 12 is evaluated as if it were written 10 > (1 + 12). The result is, of course, false.

You can link any number of relational operators together using logical operators. For example, this expression joins three relational operations:

var > 14 || !(10 < count) && 3 <= item

The following table shows the relative precedence of the relational and logical operators:

|  |  |
| --- | --- |
| **Highest** | ! |
|  | > >= < <= |
|  | == != |
|  | && |
| **Lowest** | || |

### Project 2-2) Construct an XOR Logical Operation

C++ does not define a logical operator that performs an exclusive-OR operation, usually referred to as XOR. The XOR is a binary operation that yields true when one and only one operand is true.

In this project, you will construct an XOR operation using the **&&**, **||**, and **!** operators:

1. /\*

2.

3. Project 2-2

4.

5. Create an XOR using the C++ logical operators.

6.

7. \*/

8.

9. #include <iostream>

10. using namespace std;

11.

12. int main(){

13.

14.     bool p, q;

15.

16.     p = true;

17.     q = true;

18.

19.     cout << p << " XOR " << q << " is " << ( (p || q) && !(p && q) ) << "\n";

20.

21.     p = false;

22.     q = true;

23.

24.     cout << p << " XOR " << q << " is " << ( (p || q) && !(p && q) ) << "\n";

25.

26.     p = true;

27.     q = false;

28.

29.     cout << p << " XOR " << q << " is " << ( (p || q) && !(p && q) ) << "\n";

30.

31.

32.     p = false;

33.     q = false;

34.

35.     cout << p << " XOR " << q << " is " << ( (p || q) && !(p && q) ) << "\n";

36.

37.

38.     return 0;

39. }

The following output is produced:

1 XOR 1 is 0

0 XOR 1 is 1

1 XOR 0 is 1

0 XOR 0 is 0

## 2.6) The Assignment Operator

The *assignment operator* is the single equal sign, =. The assignment operator works in C++ much as it does in any other computer language. It has the general form:

*var*  = *expression*;

Here, the value of the expression is given to *var*.

The assignment operator does have one interesting attribute: it allows you to create a chain of assignments. For example, consider the fragment:

int x, y, z;

x = y = z = 100; // set x, y, and z to 100

## 2.7) Compound Assignments

C++ provides special *compound* *assignment* operators that simplify the coding of certain assignment statements. Let’s begin with an example. The assignment statement shown here:

x = x + 10;

can be written using a compound assignment as

x += 10;

The operator pair += tells the compiler to assign to **x** the value of **x** plus 10.

Here is another example. The statement

x = x – 100;

is the same as

x -= 100;

Both statements assign to **x** the value of **x** minus 100.

There are compound assignment operators for most of the binary operators (that is, those that require two operands). Thus, statements of the form

var = var op expression;

can be converted into this compound form:

var op = expression;

Compound assignment operators are also sometimes called the *shorthand assignment* operators.

The compound assignment operators provide two benefits. First, they are more compact than their original “longhand” equivalents. Second, they can result in more efficient executable code (because their operand is evaluated only once). For these reasons, you will often see the compound assignment operator used in professionally written C++ programs.

## 2.8 Type Conversion in Assignments

When variables of one type are mixed with variables of another type, a *type conversion* will occur. In an assignment statement, the type conversion rule is easy: The value of the right side (expression side) of the assignment is converted to the type of the left side (target variable), as illustrated here:

int x;

char ch;

float f;

ch = x; /\* line 1 \*/

x = f; /\* line 2 \*/

f = ch; /\* line 3 \*/

f = x; /\* line 4 \*/

In line 1, the high-order bits of the integer variable **x** are lopped off, leaving **ch** with the lower 8 bits. If **x** were between -128 and 127, **ch** and **x** would have identical values. Otherwise, the value of **ch** would reflect only the lower-order bits of **x**. In line 2, **x** will receive the nonfractional part of **f**. In line 3, **f** will convert the 8-bit integer value stored in **ch** to the same value in the floating-point format. This also happens in line 4, except that **f** will convert an integer value into floating-point format.

When converting from integers to characters and long integers to integers, the appropriate number of high-order bits will be removed. In many 32-bit environments, this means that 24 bits will be lost when going from integer to a character, and 16 bits will be lost when going from an integer to a short integer. When converting from a floating-point type to an integer, the fractional part will be lost. If the target type is not large enough to store the result, then a garbage value will result.

## Expressions

Operators, variables and literals are constituents of *expressions*.

## 2.9) Type Conversions in Expressions

When constants and variables of different types are mixed in an expression, they are converted to the same type. First, all **char** and **short int** values are automatically elevated to **int**. This process is called *integral promotion*. Next, all operands are converted “up” to the type of the largest operand, which is called *type promotion*. The promotion is done on an operation-by-operation basis. For example, if one operand is an **int** and the other is a **long int**, then the **int** is promoted to **long int**. Or, if either operand is a **double**, the other operand is promoted to **double**. This means that conversions such as that from a **char** to a **double** are perfectly valid. Once a conversion has been applied, each pair of operands will be of the same type, and the result of each operation will be the same as the type of both operands.

### Converting to and from bool

As mentioned earlier, values of type **bool** are automatically converted into the integers 0 and 1 when sued in an integer expression. When an integer result is converted to type **bool**, 0 becomes **false** and nonzero becomes **true**. Although **bool** is a fairly recent addition to C++, the automatic conversions to and from integers mean that it has virtually no impact on older code. Furthermore, the automatic conversions allow C++ to maintain its original definition of true and false as zero and nonzero.

## 2.10) Casts

It is possible to force an expression to be of a specific type by using a construct called a *cast*. A cast is an explicit type conversion. C++ defines five types of casts. Four allow detailed and sophisticated control over casting and are described later in this book after objects have been explained. However, there is one type of cast that you can use now. It is C++’s most general cast because it can transform any type into any other type. The general form of this cast is:

(*type*) *expression*

where *type* is the target type into which you want to convert the expression. For example, if you wish to make sure the expression **x/2** is evaluated to type **float**, you can write

(float) x / 2

Casts are considered operators. As an operator, a cast is unary and has the same precedence as any other unary operator (an operator that operates only on one operand).

There are times when a cast can be very useful. For example, you may wish to use an integer for loop control, but can also perform computation on it that requires a fractional part, as in the program shown here:

1. // Demonstrate a cast

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     int i;

9.

10.     for(i = 1; i <= 10; i++){

11.         cout << i << " / 2 is: " << (float) i / 2 << "\n";

12.     }

13.

14.     return 0;

15. }

Here is the output from this program:

1 / 2 is: 0.5

2 / 2 is: 1

3 / 2 is: 1.5

4 / 2 is: 2

5 / 2 is: 2.5

6 / 2 is: 3

7 / 2 is: 3.5

8 / 2 is: 4

9 / 2 is: 4.5

10 / 2 is: 5

Without the cast (**float**) in this example, only an integer division would be performed. The cast ensures that the fractional part of the answer will be displayed.

## 2.11) Spacing and Parentheses

Summing up the book, use spaces and parentheses to make your code more readable. Use of redundant or additional parentheses will not cause errors or slow down the execution of an expression. An example of spacing and parentheses making code more readable is shown below:

x = y/3-34\*temp+127; // Hard to read

x = (y/3) – (34 \* temp) + 127; // Easier to read!

## Project 2-3) Compute the Regular Payments on a Loan

In this project, you will create a program that computes the regular payments on a loan, such as a car loan. Given the principal, the length of time, number of payments per year, and the interest rate, the program will compute the payment. Since this is a financial calculation, you will need to use floating-point data types for the computation. Since **double** is the most commonly used floating-point type, we will use it in this project. This project also demonstrates another C++ library function **pow( )**.

To compute the payments , you will use the following formula:

Payment =

where IntRate specifies the interest rate, Principal contains the starting balance, PayPerYear specifies the number of payments per year, and NumYears specifies the length of the loan in years.

Notice that in the denominator of the formula, you will raise one value to the power of another. To do this, you will use **pow( )**. Here is how you will call it:

*result* = pow(*base*, *exp*);

**pow** returns the value of *base* raised to the *exp* power. The arguments to **pow( )** are **double** values, and **pow( )** returns a value of the type **double**.

1. /\*

2.     Compute the regular payments for a loan.

3.

4.     Call this file RegPay.cpp

5. \*/

6.

7. #include <iostream>

8. #include <cmath>

9. using namespace std;

10.

11. int main(){

12.

13.     double Principal;    // original principal

14.     double IntRate;      // interest rate, such as 0.075

15.     double PayPerYear;   // number of payments per year

16.     double NumYears;     // number of years

17.     double Payment;      // the regular payment

18.     double numer, denom; // temporary work variables

19.     double b, e;         // base and exponent for call to pow()

20.

21.

22.     cout << "Enter the principal: ";

23.     cin >> Principal;

24.

25.     cout << "Enter the interest rate (i.e., 0.075): ";

26.     cin >> IntRate;

27.

28.     cout << "Enter the number of payments per year: ";

29.     cin >> PayPerYear;

30.

31.     cout << "Enter number of years: ";

32.     cin >> NumYears;

33.

34.     numer = IntRate \* Principal / PayPerYear;

35.

36.     e = -(PayPerYear \* NumYears);

37.     b = (IntRate / PayPerYear) + 1;

38.

39.     denom = 1 - pow(b, e);

40.

41.     Payment = numer / denom;

42.

43.     cout << "Payment is: " << Payment << "\n";

44.

45.

46.     // Just assuming that this math is correct since im not familiar with

47.     cout << "Total interest paid over the life of the loan is: " << Payment \* NumYears;

48.     return 0;

49. }

Here is a sample run:

Enter the principal: 10000

Enter the interest rate (i.e., 0.075): 0.075

Enter the number of payments per year: 12

Enter number of years: 5

Payment is: 200.379

Total interest paid over the life of the loan is: 1001.9

# Module 3 - Program Control Statements

This module discusses the statements that control a program’s flow of execution. There are three categories of program control statements: *selection* statements, which include the **if** and the **switch**; *iteration* statements, which include the **for**, **while**, and the **do-while** loops; and *jump* statements, which include **break**, **continue**, **return**, and **goto**. Except for **return**, which is discussed later in this book, the remaining control statements, including the **if** and **for** statements to which you have already had a brief introduction, are examined here.

## 3.1) The if Statement

The complete form of the **if** statement is:

if(*expression*) *statement*;

else *statement*;

where the targets of the **if** and **else** are single statements. The **else** clause is optional. The targets of both the **if** and **else** can be blocks of statements. The general form of the **if** using blocks of statements is:

if(*expression*){

*statement sequence;*

}

else{

*statement sequence*;

}

If the conditional expression is true, the target of the **if** will be executed; otherwise, the target of the **else**, if it exists, will be executed. At no time will both be executed. The conditional expression controlling the **if** may be any type of valid C++ expression that produces a true or false result.

The following program demonstrates the **if** by playing a simple version of the “guess the magic number” game. This program also introduces another C++ library function, called **rand( )**, which returns a randomly selected integer value. It requires the <**cstdlib**> header.

1. // Magic Number program.

2.

3. #include <iostream>

4. #include <cstdlib>

5. using namespace std;

6.

7. int main(){

8.

9.     int magic;  // magic number

10.     int guess;  // user's guess

11.

12.     magic = rand();  // gets a random number

13.

14.     cout << "Enter your guess: ";

15.     cin >> guess;

16.

17.     if(guess == magic){

18.         cout << "\*\* Right \*\*";

19.     }

20.     else{

21.         cout << "...Sorry, you're wrong.";

22.     }

23.

24.     return 0;

25. }

Below is a sample output:

Enter your guess: 3

...Sorry, you're wrong.

### The Conditional Expression

Sometimes newcomers to C++ are confused by the fact that any valid C++ expression can be used to control the **if**. That is, the conditional expression need not be restricted to only those involving the relational and logical operators, or to operands of type **bool**. All that is required is that the controlling expression evaluate to either a true or false result. As you should recall from the previous module, a value of 0 is automatically converted into **false**, and all non-zero values are converted to **true**. Thus, any expression that results in a 0 or non-zero value can be used to control the **if**. For example, this program reads two integers from the keyboard and displays the quotient. To avoid a divide-by-zero error, an **if** statement, controlled by the second number, is used.

1. // Use an int value to control the if.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     int a, b;

9.

10.     cout << "Enter numerator: ";

11.     cin >> a;

12.     cout << "Enter a denominator: ";

13.     cin >> b;

14.

15.     if(b){

16.         cout << "Result is: " << a /b << "\n";

17.     }

18.     else{

19.         cout << "Cannot divide by zero\n";

20.     }

21.

22.     return 0;

23. }

Here are two sample runs:

Enter numerator: 4

Enter a denominator: 3

Result is: 1

Enter numerator: 12

Enter a denominator: 0

Cannot divide by zero

Notice that **b** (the divisor) is tested for zero using **if(b)**. This approach works because when **b** is zero, the condition controlling the **if** is false and the **else** executes. Otherwise, the condition is true (non-zero) and the division takes place. It is not necessary (and would be considered bad style by many C++ programmers) to write this **if** as shown here:

if( b == 0){

cout << a / b << “\n”;

}

This form of the statement is redundant and potentially inefficient.

### Nested ifs

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. The main thing about nested **if**s in C++ is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and not already associated with an **else**. Here is an example:

1. if(i){

2. if(j){

3. result = 1;

4. }

5. if(k){

6. result = 2;

7. }

8. else {

9. result = 4; // this is associated with if(k)

10. }

11. }

12. else{

13. result = 4; // this is associated with if(i)

14. }

As the comments indicate, the final **else** is not associated with **if(j)** (even though it is the closest **if** without an **else**), because it is not in the same block. Rather, the final **else** is associated with **if(i)**. The inner **else** is associated with **if(k)** because that is the nearest **if**.

You can use a nested **if** to add a further improvement to the Magic Number program.

This addition provides the player with feedback about a wrong guess.

1. // Magic Number program.

2.

3. #include <iostream>

4. #include <cstdlib>

5. using namespace std;

6.

7. int main(){

8.

9.     int magic;  // magic number

10.     int guess;  // user's guess

11.

12.     magic = rand();  // gets a random number

13.

14.     cout << "Enter your guess: ";

15.     cin >> guess;

16.

17.     if(guess == magic){

18.         cout << "\*\* Right \*\*";

19.     }

20.     else{

21.         cout << "...Sorry, you're wrong.\n";

22.         if(guess < magic){

23.             cout << "Your guess is too high.\n";

24.         }

25.         else{

26.             cout << "Your guess is too low.\n";

27.         }

28.     }

29.

30.     return 0;

31. }

### The if-else-if Ladder

A common programming construct that is based upon nested **if**s is the **if-else-if** *ladder*, also referred to as the **if-else-if** *staircase*. It looks like this:

if(*condition*)

*statement*;

else if(*condition*)

*statement*;

else if(*condition*)

*statement*;

.

.

.

else

*statement*;

The conditional expressions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** often acts as a default condition; that is, if all the other conditions fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are false, then no action will take place.

The following program demonstrates the **if-else-if** ladder:

1. // Demonstrates an if-else-if ladder.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     int x;

9.

10.     for(x=0; x<6; x++){

11.         if(x==1){

12.             cout << "x is one\n";

13.         }

14.         else if(x==2){

15.             cout << "x is two\n";

16.         }

17.         else if(x==3){

18.             cout << "x is three\n";

19.         }

20.         else if(x==4){

21.             cout << "x is four\n";

22.         }

23.         else{

24.             cout << "x is not between 1 and 4\n";

25.         }

26.     }

27.

28.     return 0;

29. }

The program produces the following output:

x is not between 1 and 4

x is one

x is two

x is three

x is four

x is not between 1 and 4

As you an see, the default **else** is executed only if none of the preceding **if** statements succeeds.

## 3.2) The switch Statement

The second of C++’s selection statements is the **switch**. The **switch** provides for a multiway branch. Thus, it enables a program to select among several alternatives. Although a series of nested **if** statements can perform multiway tests, for many situations the **switch** is a more efficient approach. It works like this: the value of an expression is successively tested against a list of constants. When a match is found, the statement sequence associated with that match is executed. The general form of the **switch** statement is:

switch(*expression*){

case *constant1*:

*statement sequence*

break;

case *constant2*:

*statement sequence*

break;

case *constant2*:

*statement sequence*

break;

.

.

.

default:

*statement sequence*

}

The **switch** expression must evaluate to either a character or an integer value. (Floating-point expressions, for example, are not allowed.) Frequently, the expression controlling the **switch** is simply a variable. The **case** constants must be integer or character literals.

The **default** statement sequence is performed if no matches are found. The **default** is optional; if it is not present, no action takes place if all matches fail. When a match is found, the statements associated with that **case** are executed until the **break** is encountered or, in a concluding **case** or **default** statement, until the end of the **switch** is reached.

There are four important things to know about the **switch** statement:

* + The **switch** differs from the **if** in that the **switch** can only test for equality (that is, for matches between the **switch** expression and the **case** constants), whereas the **if** conditional expression can be of any type.
  + No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement enclosed by an outer switch may have **case** constants that are the same.
  + A **switch** statement is usually more efficient than nested **if**s.
  + The statement sequences associated with each **case** are *not* blocks. However, the entire **switch** statement *does* define a block. The importance of this will become apparent as you learn more about C++>

The following program demonstrates the **switch**. It asks for a number between 1 and 3, inclusive. It then displays a proverb linked to that number. Any other number causes an error message to be displayed.

1. /\*

2.     A simple proverb generator that

3.     demonstrates the switch.

4. \*/

5.

6. #include <iostream>

7. using namespace std;

8.

9. int main(){

10.

11.     int num;

12.

13.     cout << "Enter a number from 1 to 3: ";

14.     cin >> num;

15.

16.     switch(num){

17.         case 1:

18.             cout << "A rolling stone gathers no moss.\n";

19.             break;

20.         case 2:

21.             cout << "A bird in hand is worth two in the bush.\n";

22.             break;

23.         case 3:

24.             cout << "A fool and his money are soon parted.\n";

25.             break;

26.         default:

27.             cout << "You must enter either 1, 2, or 3.\n";

28.     }

29.

30.     return 0;

31. }

Here are two sample runs:

Enter a number from 1 to 3: 3

A fool and his money are soon parted.

Enter a number from 1 to 3: 4

You must enter either 1, 2, or 3.

Technically, the **break** statement is optional, although most applications of the **switch** will use it. When encountered within the statement sequence of a **case**, the **break** statement causes program flow to exit from the entire **switch** statement and resume at the next statement outside the **switch**. However, if a **break** statement does not end the statement sequence associated with a **case**, then all the statements *at* and *below* the matching **case** will be executed until a **break** (or the end of a **switch**) is encountered. For example, study the following program carefully. Can you figure out what it will display on the screen?

1. // A switch without break statements.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     int i;

9.

10.     for(i=0; i<5; i++){

11.         switch(i){

12.             // No break statements for any of the cases!

13.             // This will cause it so for each iteration of i it will look at a case it matches and then

14.             // execute all the other ones below as well.

15.             // e.g., when i = 0 we match case 0 so we print what's in the case but we also print what's in

16.             // all the other cases. When i = 1, we match case 1 so we print what's in the case but we also

17.             // print waht's in all the other cases.

18.             case 0:

19.                 cout << "less than 1\n";

20.             case 1:

21.                 cout << "less than 2\n";

22.             case 2:

23.                 cout << "less than 3\n";

24.             case 3:

25.                 cout << "less than 4\n";

26.             case 4:

27.                 cout << "less than 5\n";

28.         }

29.         cout << "\n";

30.     }

31.

32.     return 0;

33. }

34.

This program displays the following output:

less than 1

less than 2

less than 3

less than 4

less than 5

less than 2

less than 3

less than 4

less than 5

less than 3

less than 4

less than 5

less than 4

less than 5

less than 5

As this program illustrates, execution will continue into the next **case** if no **break** statement is present.

You can have empty **cases**, as shown in this example:

switch(i) {

case 1:

case 2:

case 3:

cout << “i is less than 4”;

break;

case 4:

cout << “i is 4”;

break;

}

In this fragment, if **I** has the value 1, 2, or 3, then the message

i is less than 4

is displayed. If it is 4, then

i is 4

is displayed. The “stacking” of **cases**, as shown in this example, is very common when several **cases** share common code.

### Nested switch Statements

It is possible to have a **switch** as part of the statement sequence of an outer **switch**. Even if the **case** constants of the inner and outer **switch** contain common values, no conflicts will arise. For example, the following code fragment is perfectly acceptable:

#include <iostream>

using namespace std;

int main(){

    switch(ch1){

        case 'A':

            cout << This A is part of outer switch";

            switch(ch2){

                case 'A':

                    cout << "This A is part of inner switch";

                    break;

                case 'B':

                    ...

            }

            break;

        case 'B':

            ...

    }

    return 0;

}

*Note: Use an* ***if-else-if*** *ladder rather than a* ***switch*** *when testing floating-point values or other objects that are not of types valid for use in a* ***switch*** *expression.*

### Project 3-1) Start Building a C++ Help System

This project builds a simple help system that displays the syntax for the C++ control statements. The program displays a menu containing the control statements and then waits for you to choose one. After one is chosen, the syntax of the statement is displayed. In this first version of the program, help is available for only the **if** and **switch** statements. The other control statements are added by subsequent projects.

1. /\*

2.     Project 3-1

3.

4.     A simple help system.

5.

6. \*/

7.

8. #include <iostream>

9. using namespace std;

10.

11. int main(){

12.     char choice;

13.

14.     cout << "Help on: \n";

15.     cout << "  1. if\n";

16.     cout << "  2. switch\n";

17.     cout << "Choose one: ";

18.     cin >> choice;

19.     cout << "\n";

20.

21.     switch(choice){

22.         case '1':

23.             cout << "The if:\n\n";

24.             cout << "if(condition) statement;\n";

25.             cout << "else statement\n";

26.             break;

27.         case '2':

28.             cout << "The switch:\n\n";

29.             cout << "switch(expression) {\n";

30.             cout << "  case constant:\n";

31.             cout << "    statement sequence\n";

32.             cout << "    break;\n";

33.             cout << "  // ...\n";

34.             cout << "}\n";

35.             break;

36.     }

37.

38.     return 0;

39. }

Here are two sample runs:

Help on:

1. if

2. switch

Choose one: 1

The if:

if(condition) statement;

else statement

Help on:

1. if

2. switch

Choose one: 2

The switch:

switch(expression) {

case constant:

statement sequence

break;

// ...

}

## 3.3) The for loop

Let’s begin by reviewing the basics, starting with the most traditional forms of the **for**.

The general form of the **for** loop for repeating a single statement is:

for(*initialization*; *expression*; *increment*) *statement*;

For repeating a block, the general form is:

for(*initialization*; *expression*; *increment*){

*statement sequence*

}

The *initialization* is usually an assignment statement that sets the initial value of the *loop control variable*, which acts as the counter that controls the loop. The *expression* is a conditional expression that determines whether the loop will repeat. The *increment* defines the amount by which the loop control variable will change each time the loop is repeated. Notice that these three major sections of the loop must be separated by semicolons. The **for** loop will continue to execute as long as the conditional expression tests true. Once the condition becomes false, the loop will exit, and program execution will resume on the statement following the **for** block.

The following program uses a **for** loop to print the square roots of the numbers between 1 and 99. Notice that in this example, the loop control variable is called **num**.

1. // Show square roots of 1 to 99.

2.

3. #include <iostream>

4. #include <cmath>

5. using namespace std;

6.

7. int main(){

8.

9.     int num;

10.     double sq\_root;

11.

12.     for(num = 1; num < 100; num++){

13.         sq\_root = sqrt((double) num); // Inside sqrt we use a cast to turn num into a double since sqrt requires input to be a double

14.         cout << num << " " << sq\_root << "\n";

15.     }

16.

17.     return 0;

18. }

This program uses the standard function **sqrt( )**. The **sqrt( )** function returns the square root of its argument. The argument must be of type **double**, and the function returns a value of type **double.** The header <**cmath**> is required.

The **for** loop can proceed in a positive or negative fashion, and it can increment the loop control variable by any amount. For example, the following program prints the numbers 50 to -50, in decrements of 10:

1. // A negatively running for loop.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     int i;

9.

10.     for(i = 50; i >= -50; i = i - 10){

11.         cout << i << ' ';

12.     }

13.

14.     return 0;

15. }

Here is the output of the program:

50 40 30 20 10 0 -10 -20 -30 -40 -50

An important point about **for** loops is that the conditional expression is always tested at the top of the for loop. This means that the code inside the loop may not be executed at all if the condition is false to begin with. Here is an example:

1. for(count=10; count < 5; count++){

2. cout << count; // this statement will never execute

3. }

This loop will never execute, because its control variable, **count**, is greater than 5 when the loop is first entered. This makes the conditional expression, **count < 5**, false from the outset; thus, not even one iteration of the loop will occur.

### Some Variations on the for Loop

The **for** loop is one of the most versatile statements in the C++ language because it allows a wide range of variations. For example, multiple loop control variables can be used. Consider the following fragment of code:

1. #include <iostream>

2. using namespace std;

3.

4. int main(){

5.

6.     int x, y; // initialization of loop control variables

7.

8.     for(x=0,  y=10; x <=y; ++x, --y){

9.         cout << x << ' ' << y << "\n";

10.     }

11.

12.     return 0;

13. }

Here is the output:

0 10

1 9

2 8

3 7

4 6

5 5

Here, commas separate the two initialization statements and the two increment expression. This is necessary in order for the compiler to understand that there are two initialization and two increment statements. In C++, the comma is an operator that essentially means “do this and this”. It’s most common in the **for** loop. You can have any number of initialization and increment statements, but in practice, more than two or three make the **for** loop unwieldy.

The condition controlling the loop may be any valid C++ expression. It does not need to involve the loop control variable. In the next example, the loop continues to execute until the **rand( )** function produces a value greater than 20,000.

1. /\*

2.     Loop until a random number that is greater than 20,000.

3. \*/

4.

5. #include <iostream>

6. #include <cstdlib>

7. using namespace std;

8.

9. int main(){

10.

11.     int i, r;

12.

13.     r = rand();

14.

15.     for(i =0; r <= 20000; i++){

16.         r = rand();

17.     }

18.

19.     cout << "Number is " << r << ". It was generated on try " << i << ".";

20.

21.

22.     return 0;

23. }

Here is a sample run:

Number is 26500. It was generated on try 3.

### Missing Pieces

Another aspect of the **for** loop that is different in C++ than in many other computer languages is that pieces of the loop definition need not be there. For example, if you want to write a loop that runs until the number 123 is typed in at the keyboard, it could look like this.

1. // A for loop with no increment.

2.

3. #include <iostream>

4. using namespace std;

5.

6. int main(){

7.

8.     int x;

9.

10.     for(x=0; x !=123; ){    // no increment expression in this for loop!

11.         cout << "Enter a number: ";

12.         cin >> x;

13.     }

14.

15.     return 0;

16. }

Here, the increment portion of the **for** definition is blank. This means that each time the loop repeats, **x** is tested to see whether it equals 123, but no further action takes place. If, however, you type 123 at the keyboard, the loop condition becomes false and the loop exits. The **for** loop will not modify the loop control variable if no increment portion of the loop is present.

Another variation on the **for** is to move the initialization section outside of the loop, as shown in this fragment.

1. x = 0;

2.

3. for( ; x<10; ){

4. cout << x << ‘ ‘;

5. ++x;

6. }

Here, the initialization section has been left blank, and **x** is initialized before the loop is entered. Placing the initialization outside of the loop is generally done only when the initial value is derived through a complex process that does not lend itself to containment inside the **for** statement. Notice that in this example, the increment portion of the **for** is located inside the body of the loop.

### The Infinite for Loop