Copperbelt University
School of Information and Communication Technology
Computer Science Department

Introduction to Programming in C++

Extracted from Sam's Teach yourself C++ in 21 Days

The Parts of a C++ Program

C++ programs consist of objects, functions, variables, and other component parts. To get a sense of how a program fits together you must see a complete working program.

A Simple Program

Even a simple program like the one below has many interesting parts. Let us review this program in more detail.

```
1: #include <iostream.h>
2:
3:  int main()
4: {
5:    cout << "Hello World!\n";
6:    return 0;
7: }
```

On line 1, the file *iostream.h* is included in the file. The first character is the # symbol, which is a signal to the preprocessor. Each time you start your compiler, the preprocessor is run. The preprocessor reads through your source code, looking for lines that begin with the hash symbol (#), and acts on those lines before the compiler runs. *include* is a preprocessor instruction that says, "What follows is a filename. Find that file and copy it to this position". The angle brackets around the filename tell the preprocessor to look in all the usual places for this file. If your compiler is set up correctly, the angle brackets will cause the preprocessor to look for the file *iostream.h* in the directory that holds all the header (H) files for your compiler. The file *iostream.h* (Input-Output-Stream) is used by *cout*, which assists with writing to the screen. The effect of line 1 is to include the file *iostream.h* into this program as if you had typed it in yourself. The preprocessor runs before your compiler each time the compiler is invoked. The preprocessor translates any line that begins with a hash symbol (#) into a special command, getting your code file ready for the compiler.

Line 3 begins the actual program with a function named main(). Every C++ program has a main() function. In general, a function is a block of code that performs one or more actions. Usually functions are invoked or called by other functions, but main() is special. When your program starts, main() is called automatically. main(), like all functions, must state what kind of value it will return. The return value type for main() in HELLO.CPP is int, which means that this function will return an integer value. All functions begin with an opening brace ({) and end with a closing brace (}). The braces for the main() function are on lines 4 and 7. Everything between the opening and closing braces is considered a part of the function. The object *cout* in line 5 is used to print a message to the screen. *cout* is used in C++ to print strings and values to the screen.

A string is just a set of characters. To use *cout*, type the word cout, followed by the output redirection operator (<<). Whatever follows the output redirection operator is written to the screen. If you want a string of characters written, be sure to enclose them in double quotes ("), as shown on line 5 of our example program. *Note: A text string is a series of, usually printable characters.* The final two characters of our example program, \n, tell cout to put a new line after the words Hello World! The main() function ends on line 7 with the closing brace.

A Brief Look at *cout*

Although we shall have a closer look at *cout* later on, for now we shall see how to use *cout* to print data to the screen without fully understanding how this *cout* works. To print a value to the screen, write the word *cout*, followed by the insertion operator (<<), which you create by typing the less-than character (<) twice. Even though this is two characters, C++ treats it as one. The following program gives the example.

```
1:    // Using cout
2:
3:    #include <iostream.h>
4:    int main()
5:    {
6:       cout << "Hello there.\n";
7:       cout << "Here is 5: " << 5 << "\n";
8:       cout << "The manipulator endl writes a new line to
               the screen." << endl;
9:       cout << "Here is a very big number:\t" << 70000 << endl;
10:      cout << "Here is the sum of 8 and 5:\t" << 8+5 <<  endl;
11:      cout << "Here's a fraction:\t\t" << (float) 5/8 << endl;
12:      cout << "And a very very big number:\t" << (double)
         7000*7000 << endl;
13:      cout << "Don't forget to replace Jesse Liberty with
               your name...\n";
14:      cout << "Gwembe Shangombo is a C++ programmer!\n";
15:      return 0;
16: }
```

On line 3, the statement #include <iostream.h> causes the iostream.h file to be added to your source code. This is required if you use *cout* and its related functions. On line 6 is the simplest use of *cout*, printing a string or series of characters. The symbol \n is a special formatting character. It tells *cout* to print a newline character to the screen. Three values are passed to *cout* on line 7, and each value is separated by the insertion operator. The first value is the string "Here is 5: ". Note the space after the colon. The space is part of the string. Next, the value 5 is passed to the insertion operator and the newline character (always in double quotes or single quotes). This causes the line

```
Here is 5: 5
```

to be printed to the screen. Because there is no newline character after the first string, the next value is printed immediately afterwards. This is called concatenating the two values. On line 8, an informative message is printed, and then the manipulator *endl* is used. The purpose of *endl* is to write a new line to the screen. On line 9, a new formatting character, *\t*, is introduced. This inserts a tab character and is used on lines 9-12 to line up the output. Line 9 shows that not only integers, but long integers as well can be printed. Line 10 demonstrates that *cout* will do simple addition. The value of 8+5 is passed to *cout*, but 13 is printed.

On line 11, the value 5/8 is inserted into *cout*. The term *(float)* tells *cout* that you want this value evaluated as a decimal equivalent, and so a fraction is printed. On line 12 the value 7000 * 7000 is given to *cout*, and the term *(double)* is used to tell *cout* that you want this to be printed using scientific notation.

Comments

When you are writing a program, it is always clear and self-evident what you are trying to do. However, a month or so later, when you return to the program, it can be quite confusing and unclear. To fight the onset of confusion, and to help others understand your code, you'll want to use comments. Comments are simply text that is ignored by the compiler, but that may inform the reader of what you are doing at any particular point in your program.

Types of Comments

C++ comments come in two flavors: the double-slash (//) comment, and the slash-star (/*) comment. The double-slash comment, which will be referred to as a C++-style comment, tells the compiler to ignore everything that follows this comment, until the end of the line. The slash-star comment mark tells the compiler to ignore everything that follows until it finds a star-slash (*/) comment mark. These marks will be referred to as the C-style comments since C++ inherited them from the C language. Every /* must be matched with a closing */. Many C++ programmers use the C++-style comment most of the time, and reserve C-style comments for blocking out large blocks of a program. You can include C++-style comments within a block "commented out" by C-style comments; everything, including the C++-style comments, is ignored between the C-style comment marks.

Using Comments

As a general rule, the overall program should have comments at the beginning, telling you what the program does. Each function should also have comments explaining what the function does and what values it returns. The following example demonstrates the use of comments, showing that they do not affect the processing of the program or its output.

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:  /* this is a comment
6:  and it extends until the closing
7:  star-slash comment mark */
8:    cout << "Hello World!\n";
9:    // this comment ends at the end of the line
10:   cout << "That comment ended!\n";
11:
12:  // double slash comments can be alone on a line
13: /* as can slash-star comments */
14:     return 0;
15: }
```

Functions

While *main()* is a function, it is an unusual one. Typical functions are called, or invoked, during the course of your program execution. A program is executed line by line in the order it appears in your source code, until a function is reached. Then the program branches off to execute the function. When the function finishes execution, it returns control to the line of code immediately following the call to the function. A good analogy for this is sharpening your pencil. If you are drawing a picture, and your pencil breaks, you might stop drawing, go sharpen the pencil, and then return to what you were doing. When a program needs a service performed, it can call a function to perform the service and then pick up where it left off when the function has finished running. The following program demonstrates this idea.

```
1:    #include <iostream.h>
2:
3:    // function Demonstration Function
4:    // prints out a useful message
5:    void DemonstrationFunction()
6:    {
7:       cout << "In Demonstration Function\n";
8:    }
9:
10:    // function main - prints out a message, then
11:    // calls DemonstrationFunction, then prints out
12:    // a second message.
13:    int main()
14:    {
15:       cout << "In main\n" ;
16:       DemonstrationFunction();
17:       cout << "Back in main\n";
18:        return 0;
19: }
```

The function *DemonstrationFunction()* is defined on lines 5-7. When it is called, it prints a message to the screen and then returns. Line 13 is the beginning of the actual program. On line 15, *main()* prints out a message saying it is in *main()*. After printing the message, line 16 calls *DemonstrationFunction()*. This call causes the commands in *DemonstrationFunction()* to execute. In this case, the entire function consists of the code on line 7, which prints another message. When *DemonstrationFunction()* completes (line 8), it returns control to where it was called from. In this case the program returns to line 17, where *main()* prints its final line.

Using Functions

Functions either return a value or they return void, meaning they return nothing. A function that adds two integers might return the sum, and thus would be defined to return an integer value. A function that just prints a message has nothing to return and would be declared to return void. Functions consist of a header and a body. The header consists, in turn, of the return type, the function name, and the parameters to that function. The parameters to a function allow values to be passed into the function. Thus, if the function were to add two numbers, the numbers would be the parameters to the function. Here's a typical function header:

```
int Sum(int a, int b)
```

A parameter is a declaration of what type of value will be passed into the function. The actual value passed in by the calling function is called the argument. Many programmers use these two terms, parameters and arguments, as synonyms. Others are careful about the technical distinction. As scientists, please stick to the right terms. The body of a function consists of an opening brace, zero or more statements, and a closing brace. The statements constitute the work of the function. A function may return a value, using a return statement. This statement will also cause the function to exit. If you don't put a return statement into your function, it will automatically return void at the end of the function. The value returned must be of the type declared in the function header. Although functions will be dealt with in more details later, a good introduction will be given here. The following program demonstrates a function that takes two integer parameters and returns an integer value.

```
1:   #include <iostream.h>
2:   int Add (int x, int y)
3:   {
4:
5:     cout << "Add(), received " << x << " and " << y << "\n";
6:     return (x+y);
7:   }
8:
9:   int main()
10:    {
11:        cout << "I'm in main()!\n";
12:        int a, b, c;
13:        cout << "Enter two numbers: ";
14:        cin >> a;
15:        cin >> b;
16:        cout << "\nCalling Add()\n";
17:        c=Add(a,b);
18:        cout << "\nBack in main().\n";
19:        cout << "c was set to " << c;
20:        cout << "\nExiting...\n\n";
21:        return 0;
22: }
```

The function Add() is defined on line 2. It takes two integer parameters and returns an integer value. The program itself begins on line 9 and on line 11, where it prints a message. The program prompts the user for two numbers (lines 13 to 15). The user types each number, separated by a space, and then presses the Enter key. main() passes the two numbers typed in by the user as arguments to the Add() function on line 17. After this, processing branches to the Add() function, which starts on line 2. The parameters a and b are printed and then added together. The result is returned on line 6, and the function returns. In lines 14 and 15, the cin object is used to obtain values for the variables a and b, and cout is used to write the values to the screen.

Variables and Constants

Programs need a way to store the data they use. Variables and constants offer various ways to represent and manipulate that data.

What Is a Variable?

In C++ a variable is a memory location to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value. Your computer's memory can be viewed as a series of locations. Each location is one of many, many such and all are lined up. Each memory location is numbered sequentially. These numbers are known as memory addresses. A variable reserves one or more memory locations in which you may store a value. Your variable's name (for example, `myVariable`) is a label on one of these memory locations, so that you can find it easily without knowing its actual memory address.

Setting Memory Aside

When you define a variable in C++, you must tell the compiler what kind of variable it is: an integer, a character, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable. Each memory location is one byte large. If the type of variable you create is two bytes in size, it needs two bytes of memory, or two memory locations. The type of the variable (for example, integer) tells the compiler how much memory (how many memory locations) to set aside for the variable.

Size of Integers

On any one computer, each variable type takes up a single, unchanging amount of room. That is, an integer might be two bytes on one machine, and four on another, but on one particular computer, the amount of storage is always the same for a particular type, day in and day out. A *`char`* variable (used to hold characters) is most often one byte long. A *`short`* integer is two bytes on most computers, a *`long`* integer is usually four bytes, and an integer (without the keyword `short` or `long`) can be two or four bytes. A *character* is a single letter, number, or symbol that takes up one byte of memory. The following program helps you to determine the size of memory allocated to your variable type.

```
1:  #include <iostream.h>
2:
3:  int main()
4:  {
5:    cout << "The size of an int is:\t\t"    << sizeof(int)    <<     " bytes.\n";
6:    cout << "The size of a short int is:\t" << sizeof(short)  << " bytes.\n";
7:    cout << "The size of a long int is:\t"  << sizeof(long)   << " bytes.\n";
8:    cout << "The size of a char is:\t\t"    << sizeof(char)   <<  " bytes.\n";
9:    cout << "The size of a float is:\t\t"   << sizeof(float)  <<   " bytes.\n";
10:   cout << "The size of a double is:\t"    << sizeof(double) << " bytes.\n";
11:
12:       return 0;
13: }
```

You may discover that the number of bytes presented might be different from computer to computer. The one new feature in our previous program is the use of the *`sizeof()`* function in lines 5 through 10. *`sizeof()`* is provided by your compiler, and it tells you the size of the object you pass in as a parameter. For example, on line 5 the keyword *`int`* is passed into *`sizeof()`*.

Signed and Unsigned

In addition, all integer types come in two varieties: *signed* and *unsigned*. The idea here is that sometimes you need negative numbers, and sometimes you don't. Integers (*short* and *long*) without the word "unsigned" are assumed to be *signed*. Signed integers are either negative or positive. *Unsigned* integers are always positive. Because you have the same number of bytes for both *signed* and *unsigned* integers, the largest number you can store in an *unsigned* integer is twice as big as the largest positive number you can store in a *signed* integer. An *unsigned short* integer can handle numbers from 0 to 65,535. Half the numbers represented by a *signed short* are negative, thus a *signed short* can only represent numbers from -32,768 to 32,767.

Fundamental Variable Types

Several other variable types are built into C++. They can be conveniently divided into integer variables (the type discussed so far), floating-point variables, and character variables. Floating-point variables have values that can be expressed as fractions, that is, they are real numbers. Character variables hold a single byte and are used for holding the 256 characters and symbols of the ASCII and extended ASCII character sets. *The ASCII character set* is the set of characters standardized for use mostly on microcomputers. ASCII is an acronym for American Standard Code for Information Interchange. Nearly every computer operating system supports ASCII, though many support other international character sets as well, such as the Extended Binary Coded Decimals Interchange Code (EBCDIC) which is mostly used on mainframes. The types of variables used in C++ programs are described in the table below. This table shows the variable type, how much room we assume it takes in memory, and what kinds of values that can be stored in these variables. The values that can be stored are determined by the size of the variable types. This is computer dependent as shown in our earlier program.

Type	***Size***	***Values***
`unsigned short int`	2 bytes	0 to 65,535
`short int`	2 bytes	-32,768 to 32,767
`unsigned long int`	4 bytes	0 to 4,294,967,295
`long int`	4 bytes	-2,147,483,648 to 2,147,483,647
`int (16 bit)`	2 bytes	-32,768 to 32,767
`int (32 bit)`	4 bytes	-2,147,483,648 to 2,147,483,647
`unsigned int (16 bit)`	2 bytes	0 to 65,535
`unsigned int (32 bit)`	2 bytes	0 to 4,294,967,295
`Char`	1 byte	256 character values
`Float`	4 bytes	1.2e-38 to 3.4e38
`Double`	8 bytes	2.2e-308 to 1.8e308

Defining a Variable

You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but cannot contain spaces. Legal variable names include *x*, *J23qrsnf*, *and* *myAge*. Good variable names tell you what the

variables are for; using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called *myAge*: int myAge;

As a general programming practice, avoid such horrific names as *J23qrsnf*, and restrict single-letter variable names (such as `x` or `i`) to variables that are used only very briefly. Try to use expressive names such as *myAge* or *howMany*. Such names are easier to understand many days later when you are scratching your head trying to figure out what you meant when you wrote that line of code. Based on the first few lines of code in the following two programs, try to figure out what they do.

Example 1

```
main()
{
    unsigned short x;
    unsigned short y;
    ULONG z;
    z = x * y;
}
```

Example 2

```
main ()
{
    unsigned short Width;
    unsigned short Length;
    unsigned short Area;
    Area = Width * Length;
}
```

Clearly, the second program is easier to understand, and the inconvenience of having to type the longer variable names is more than made up for by how much easier it is to maintain the second program.

Case Sensitivity

C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named *age* is different from *Age*, which is different from *AGE*. However, some compilers may allow you to turn case sensitivity off. But please, don't be tempted to do this because your programs may not work with other compilers, and other C++ programmers may find your program hard to understand, let alone to modify. There are various conventions for how to name variables, and although it doesn't much matter which method you adopt, it is important to be consistent throughout your program. Many programmers prefer to use all lowercase letters for their variable names. If the name requires two words (for example, my car), there are two popular conventions: *my_car* or *myCar*. The latter form is called camel-notation, because the capitalization looks something like a camel's hump.

Keywords

Some words are reserved by C++, and you may not use them as variable names. These are called keywords used by the compiler to control your program. Keywords include *if*, *while*, *for*, and *main*. Your

compiler manual should provide a complete list, but generally, any reasonable name for a variable is almost certainly not a keyword.

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas. For example:

```
unsigned int myAge, myWeight;    // two unsigned int variables
long area, width, length;        // three longs
```

As you can see, *myAge* and *myWeight* are each declared as *unsigned* integer variables. The second line declares three individual *long* variables named *area*, *width*, and *length*. The type (*long*) is assigned to all the variables, so you cannot mix types in one definition statement.

Assigning Values to Your Variables

You assign a value to a variable by using the assignment operator (=). Thus, you would assign 5 to *Width* by writing

```
unsigned short Width;
Width = 5;
```

You can combine these steps and initialize *Width* when you define it by writing

```
unsigned short Width = 5;
```

Initialization looks very much like assignment, and with integer variables, the difference is minor. Later, when constants are covered, you will see that some values must be initialized because they cannot be assigned to. The essential difference is that initialization takes place at the moment you create the variable. Just as you can define more than one variable at a time, you can initialize more than one variable at creation. For example:

```
// create two long variables and initialize them
Ulong width = 5, length = 7;
```

This example initializes the long integer variable *width* to the value 5 and the *long* integer variable *length* to the value 7. You can even mix definitions and initializations:

```
int myAge = 39, yourAge, hisAge = 40;
```

This example creates three variables of *int* type, and it initializes the first and third. The following program is a complete program, ready to compile, that computes the area of a rectangle and writes the answer to the screen.

```
1:  // Demonstration of variables
2:  #include <iostream.h>
3:
4:  int main()
```

```
5:  {
6:    unsigned short int Width = 5, Length;
7:    Length = 10;
8:
9:    // create  an unsigned short and initialize with result
10:      // of multiplying Width by Length
11:    unsigned short int Area  = Width * Length;
12:
13:    cout << "Width:" << Width << "\n";
14:    cout << "Length: "  << Length << endl;
15:    cout << "Area: " << Area << endl;
16:    return 0;
17: }
```

Line 2 includes the required *include* statement for the *iostream's* library so that *cout* will work. Line 4 begins the program. On line 6, *Width* is defined as an *unsigned short* integer, and its value is initialized to 5. Another *unsigned short* integer, *Length*, is also defined, but it is not initialized. On line 7, the value 10 is assigned to *Length*. On line 11, an *unsigned short* integer, *Area*, is defined, and it is initialized with the value obtained by multiplying *Width* times *Length*. On lines 13-15, the values of the variables are printed to the screen. Note that the special word *endl* creates a new line.

typedef

It can become tedious, repetitious, and, most important, error-prone to keep writing *unsigned short int*. C++ enables you to create an alias for this phrase by using the keyword *typedef*, which stands for type definition. In effect, you are creating a synonym, and it is important to distinguish this from creating a new type (which we shall learn later). *typedef* is used by writing the keyword *typedef*, followed by the existing type and then the new name. For example:

```
typedef unsigned short int USHORT
```

creates the new name USHORT that you can use anywhere you might have written *unsigned short int*. Our previous program is repeated below but now using the type definition *USHORT* rather than *unsigned short int*.

```
1:  // *****************
2:  // Demonstrates typedef keyword
3:  #include <iostream.h>
4:
5:  typedef unsigned short int USHORT;       //typedef defined
6:
7:  void main()
8:  {
9:    USHORT  Width = 5;
10:   USHORT Length;
11:   Length = 10;
12:   USHORT Area  = Width * Length;
13:   cout << "Width:" << Width << "\n";
14:   cout << "Length: "  << Length << endl;
```

```
15:    cout << "Area: " << Area <<endl;
16: }
```

As you can see, on line 5, *USHORT* is type defined as a synonym for *unsigned short int*. The program is very much like the previous one and the output is the same.

When to Use short and When to Use long

One source of confusion for new C++ programmers (Newbies) is when to declare a variable to be type *long* and when to declare it to be type *short*. The rule, when understood, is fairly straightforward: If there is any chance that the value you'll want to put into your variable will be too big for its type, use a larger type. As seen earlier, *unsigned short* integers, assuming that they are two bytes, can hold a value only up to 65,535. *Signed short* integers can hold only half of that. *Unsigned long* integers can hold an extremely large number (4,294,967,295) that is still quite finite. If you need a larger number, you'll have to go to *float* or *double*, and then you lose some precision. Floats and doubles can hold extremely large numbers, but only the first 7 or 19 digits are significant on most computers. That means that the number is rounded off after that many digits.

Wrapping Around an unsigned Integer

The fact that *unsigned long* integers have a limit to the values they can hold is only rarely a problem, but what happens if you do run out of room? When an *unsigned* integer reaches its maximum value, it wraps around and starts over, much as a car odometer might. The program below shows what happens if you try to put too large a value into a `short` integer.

```
1: #include <iostream.h>
2:  int main()
3:  {
4:     unsigned short int smallNumber;
5:     smallNumber = 65535;
6:     cout << "small number:" << smallNumber << endl;
7:     smallNumber++;
8:     cout << "small number:" << smallNumber << endl;
9:     smallNumber++;
10:    cout << "small number:" << smallNumber << endl;
11:    return 0;
12: }
```

On line 4, *smallNumber* is declared to be an `unsigned short int`, which can be two-byte variable, able to hold a value between 0 and 65,535. On line 5, the maximum value is assigned to *smallNumber*, and it is printed on line 6. On line 7, *smallNumber* is incremented; that is, 1 is added to it. The symbol for incrementing is ++ (as in the name C++, an incremental increase from C). Thus, the value in *smallNumber* would be `65,536`. However, `unsigned short` integers can't hold a number larger than 65,535, so the value is wrapped around to `0`, which is printed on line 8. On line 9 *smallNumber* is incremented again, and then its new value, `1`, is printed.

Wrapping Around a signed Integer

A `signed` integer is different from an `unsigned` integer, in that half of the values you can represent are negative. Instead of picturing a traditional car odometer, you might picture one that rotates up for positive numbers and down for negative numbers. One kilometre from 0 is either 1 or -1. When you run out of positive numbers, you run right into the smallest negative numbers and then count back down to 0. The following program shows what happens when you add 1 to the maximum positive number in an `unsigned short` integer. The program gives a demonstration of adding too large a number to a signed integer.

```
1:  #include <iostream.h>
2:  int main()
3:  {
4:     short int smallNumber;
5:     smallNumber = 32767;
6:     cout << "small number:" << smallNumber << endl;
7:     smallNumber++;
8:     cout << "small number:" << smallNumber << endl;
9:     smallNumber++;
10:    cout << "small number:" << smallNumber << endl;
11:        return 0;
12: }
```

On line 4, *`smallNumber`* is declared this time to be a `signed short` integer (if you don't explicitly say that it is `unsigned`, it is assumed to be `signed`). The program proceeds much as the preceding one, but the output is quite different. To fully understand this output, you must be comfortable with how `signed` numbers are represented as bits in a two-byte integer. The bottom line, however, is that just like an `unsigned` integer, the `signed` integer wraps around from its highest positive value to its lowest negative value.

Characters

Character variables (`type char`) are typically 1 byte, enough to hold 256 values. A `char` can be interpreted as a small number (0-255) or as a member of the ASCII set. ASCII stands for the American Standard Code for Information Interchange. The ASCII character set is used to encode all the letters, numerals, and punctuation marks in computers. Computers do not know about letters, punctuation marks, or sentences. All they understand are numbers. In fact, all they really know about is whether or not a sufficient amount of electricity is at a particular junction of wires. If so, it is represented internally as a `1`; if not, it is represented as a `0`. By grouping ones and zeros, the computer is able to generate patterns that can be interpreted as numbers, and these in turn can be assigned to letters and punctuation marks. In the ASCII code, the lowercase letter "a" is assigned the value `97`. All the lower- and uppercase letters, all the numerals, and all the punctuation marks are assigned values between 1 and 128. Another 128 marks and symbols are reserved for use by the computer maker.

Characters and Numbers

When you put a character, for example, `` `a' ``, into a `char` variable, what is really there is just a number between 0 and 255. The compiler knows, however, how to translate back and forth between characters (represented by a single quotation mark and then a letter, numeral, or punctuation mark, followed by a closing single quotation mark) and one of the ASCII values. The value/letter relationship is arbitrary; there

is no particular reason that the lowercase "a" is assigned the value `97`. As long as everyone (your keyboard, compiler, and screen) agrees, there is no problem. It is important to realize, however, that there is a big difference between the value `5` and the character `` `5' ``. The latter is actually valued at `53`, much as the letter `` `a' `` is valued at `97`. The following program prints the character values for the integers 32 through 127.

```
1:  #include <iostream.h>
2:  int main()
3:  {
4:  for (int i = 32; i<128; i++)
5:        cout << (char) i;
6:  return 0;
7: }
```

Output: !"#$%G'()*+,./0123456789:;<>?@ABCDEFGHIJKLMNOP
_QRSTUVWXYZ[\]^'abcdefghijklmnopqrstuvwxyz<|>~s

Special Printing Characters

The C++ compiler recognizes some special characters for formatting. The table below shows the most common ones used. You put these into your code by typing the backslash (called the escape character), followed by the character. Thus, to put a tab character into your code, you would enter a single quotation mark, the slash, the letter t, and then a closing single quotation mark:

```
char tabCharacter = `\t';
```

This example statement declares a `char` variable (*`tabCharacter`*) and initializes it with the character value `\t`, which is recognized as a tab. The special printing characters are used when printing either to the screen or to a file or other output device. An *escape character* changes the meaning of the character that follows it. For example, normally the character `n` means the letter n, but when it is preceded by the escape character (`\`) it means new line.

Character	***What it means***
`\n`	new line
`\t`	Tab
`\b`	Backspace
`\"`	double quote
`\'`	single quote
`\?`	question mark
`\\`	Backslash

Constants

Like variables, constants are data storage locations. Unlike variables, and as the name implies, constants don't change their values throughout the running of a program. You must initialize a constant when you create it, and you cannot assign a new value to it later.

Literal Constants

C++ has two types of constants: literal and symbolic. A literal constant is a value typed directly into your program wherever it is needed. For example: int myAge = 39;

myAge is a variable of type `int`; `39` is a literal constant. You can't assign a value to `39`, and its value can't be changed.

Symbolic Constants

A symbolic constant is a constant that is represented by a name, just as a variable is represented. Unlike a variable, however, after a constant is initialized, its value can't be changed. If your program has one integer variable named *students* and another named *classes*, you could compute how many students you have, given a known number of classes, if you knew there were 15 students per class:

```
students = classes * 15;       //* indicates multiplication.
```

In this example, `15` is a literal constant. Your code would be easier to read, and easier to maintain, if you substituted a symbolic constant for this value:

```
students = classes * studentsPerClass
```

If you later decided to change the number of students in each class, you could do so where you define the constant *studentsPerClass* without having to make a change every place you used that value. There are two ways to declare a symbolic constant in C++. The old, traditional, and now obsolete way is with a preprocessor directive, *#define*. To define a constant the traditional way, you would enter this:

```
#define studentsPerClass 15
```

Note that *studentsPerClass* is of no particular type (`int`, `char`, and so on). `#define` does a simple text substitution. Every time the preprocessor sees the word *studentsPerClass*, it substitutes it for 15. Because the preprocessor runs before the compiler, your compiler never sees your constant; it sees the number `15`. The other way to define constants is with the use of the reserved word *const*. Although *#define* works, there is a new, much better way to define constants in C++:

```
const unsigned short int studentsPerClass = 15;
```

This example also declares a symbolic constant named *studentsPerClass*, but this time *studentsPerClass* is typed as an `unsigned short int`. This method has several advantages in making your code easier to maintain and in preventing bugs. The biggest difference is that this constant has a type, and the compiler can enforce that it is used according to its type. Constants cannot be changed while the program is running. If you need to change `studentsPerClass`, for example, you need to change the code and recompile the program.