Introduction to C++ Programs

C was created by Dennis Ritchie at the Bell Telephone Laboratories in 1972. The language wasn't created for the fun of it, but for a specific purpose: to design the UNIX operating system (which is used on many computers).
Because C is such a powerful and flexible language, its use quickly spread beyond Bell Labs. Programmers everywhere began using it to write all sorts of programs.
Now, what about the name? The C language is so named because its predecessor was called B. The B language was developed by Ken Thompson of Bell Labs. You should be able to guess why it was called B. it was not derived from language called A ,but from language called BCLP . so C language was derived from B language , and C++ language was derived from C language .
You might have heard about C++ and the programming technique called *object- oriented programming*. Perhaps you're wondering what the differences are between C and C++ .
C++ is a superset of C, which means that C++ contains everything C does, plus new additions for object-oriented programming. If you do go on to learn C++, almost everything you learn about C will still apply to the C++ superset.
Now the question is Why are there tow pluses in the name C++, the answer is: ++ is an operation in C and C++ languages, so ++ is used to recognize between the tow languages and to give a nice pin .

Programs

The word program is used in two ways: to describe individual instructions, or source code, created by the programmer, and to describe an entire piece of executable software.

New Term: A *program* can be defined as either a set of written instructions created by a programmer or an executable piece of software.

Working with systems

There are three systems of numbers :

- **The Decimal Number System**
- **The Binary System**
- **The Hexadecimal System**

As a computer programmer, you might sometimes be required to work with numbers expressed in binary and hexadecimal notation.

The Decimal Number System

The decimal system is the base -10 system that you use every day. A number in this system for example, 342 is expressed as powers of 10. The first digit (counting from the right) gives 10 to the 0 power , the second digit gives 10 to the 1 power, and so on. Any number to the 0 power equals 1, and any number to the 1 power equals itself. Thus, continuing with the example of 342, you have:

3	**$3 * 10^2 = 3 * 100 = 300$**
4	**$4 * 10^1 = 4 * 10 = 40$**
2	**$2 * 10^0 = 2 * 1 = 2$**
	Sum = 342

The base-10 system requires 10 different digits, 0 through 9. The following rules apply to base 10 and to any other base number system:

- **A number is represented as powers of the system's base.**
- **The system of base *n* requires *n* different digits.**

Now let's look at the other number system:

The Binary System

The binary number system is base 2 and therefore requires only two digits, 0 and 1. The binary system is useful for computer programmers, because it can be used to represent the digital on/off method in which computer chips and memory work. Here's an example of a binary number and its representation in the decimal notation you're more familiar with, writing 1011 vertically:

1	**$1 * 2^3 = 1 * 8 = 8$**
0	**$0 * 2^2 = 0 * 4 = 0$**
1	**$1 * 2^1 = 1 * 2 = 2$**
1	**$1 * 2^0 = 1 * 1 = 1$**
	Sum = 11 (decimal)

Binary has one shortcoming: It's cumbersome for representing large numbers

The Hexadecimal System

The hexadecimal system is base 16. Therefore, it requires 16 digits. The digits 0 through 9 are used, along with the letters A through F, which represent the decimal values 10 through 15. Here is an example of a hexadecimal number and its decimal equivalent:

2	**$2 * 16^2 = 2 * 256 = 512$**
D	**$13 * 16^1 = 13 * 16 = 208$**
A	**$10 * 16^0 = 10 * 1 = 10$**
	Sum = 730 (decimal)

The hexadecimal system (often called the *hex* system) is useful in computer work because it's based on powers of 2. Each digit in the hex system is equivalent to a four-digit binary number, and each two-digit hex number is equivalent to an eight-digit binary number.

The following Table show the Hexadecimal numbers and their decimal and binary equivalents:

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	**0**	**0000**
1	**1**	**0001**
2	**2**	**0010**
3	**3**	**0011**
4	**4**	**0100**
5	**5**	**0101**
6	**6**	**0110**
7	**7**	**0111**
8	**8**	**1000**
9	**9**	**1001**
A	**10**	**1010**
B	**11**	**1011**
C	**12**	**1100**
D	**13**	**1101**
E	**14**	**1110**
F	**15**	**1111**
10	**16**	**10000**
F0	**240**	**11110000**

FF	**255**	**11111111**

The Parts of a C++ Program

C++ programs consist of objects, functions, variables, and other component parts.

- **The parts of a C++ program.**
- **How the parts work together.**
- **What a function is and what it does.**

HELLO.CPP demonstrates the parts of a C++ program.

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

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.

include is a preprocessor instruction that says, "What follows is a filename. Find that file and read it in right here." 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 H files for your compiler. The file iostream.h (Input-Output-Stream) is used by cin, cout which assists with reading , writing to the screen.

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 void, which means that this function will not return any value at all.

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 is used to print a message to the screen and the object cin is used to read a value from a screen . The two objects, cout and cin are used in C++ to print and read strings and values to the screen . A string is just a set of characters.

cout is used: type the word cout, followed by the output redirection operator (<<).

Cin is used : type the word cin , followed by the input redirection operator (>>)

The final two characters, \n, tell cout to put a new line after the words Hello World!

the program declare main() to return an int. This value is "returned" to the operating system when your program completes. main() will always return 0.

Program Using cout.

```
1:     // program 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 << "C++ programmer!\n";
12:         return 0;
13: }

output:
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:      70000
Here is the sum of 8 and 5:     13
C++ programmer!
```

For output , the easiest way of specifying a field width and precision with stream input / output is by using manipulators (setw() and setprecision()) for which the header file <iomanip.h>
So,for example, we can write a program :

```
#include <iostream.h>
#include<iomanip.h>
Void main()
{
Int x,y;
Cin>>x>>y;
Cout<<x<<setw(12)<<y<<setw(12)<<x*y;
}
```

Comments

When you are writing a program, it is always clear and self-evident what you are trying to do.
Comments are simply text that is ignored by the compiler.

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 C-style comments. Every /* must be matched with a closing */.

HELP.CPP demonstrates comments.

```
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: }
```

```
Output:
Hello World!
```
That comment ended!

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.

- **How to declare and define variables and constants.**
- **How to assign values to variables and manipulate those values.**
- **How to write the value of a variable to the screen.**

What Is a Variable?

In C++ a variable is a place 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.

Setting Aside Memory

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.

Table Variable Types:

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)`	**4 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.
Int myAge;

Example 1:

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

Example 2:

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

Assigning Values to the Variables

You assign a value to a variable by using the assignment operator (=). For example, 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;
```

Example:

program using a variables.

```
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: }

Output: Width:5
Length: 10
Area: 50
```

typedef

It can become tedious, repetitious, and, most important, error 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.

`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
```

A demonstration of typedef.

```
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: }

Output: Width:5
Length: 10
Area: 50
```

Note: to clear the screen after the execution of the program we can use the function clrscr()
With the header file <conio.h> .

For example :

```
#include <iostream.h>
#include<conio.h>

Void  main()
{ int  x,y ;
Clrscr();
Cout<<x<<setw(12)<<setw(12)<<y;
```

}

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 Escape Characters.

Character	*What it means*
`\n`	**New line**
`\t`	**Horizontal Tab (8 column)**
`\b`	**Backspace (return one space to the back)**
`\"`	**double quote**
`\'`	**single quote**
`\?`	**Question mark**
`\\`	**Backslash**
`\r`	**Return to the beginning of the line**
`\v`	**Vertical tap (8 row)**

Using Character Variables

Like other variables, you must declare chars before using them, and you can initialize them at the time of declaration. Here are some examples:

```
char a, b, c;         /* Declare three uninitialized char variables */
char code = `x';      /* Declare the char variable named code */
                      /* and store the character x there */
code = '!';           /* Store ! in the variable named code */
```

You can create symbolic character constants by using either the #define directive or the const keyword:

```
#define EX `x'
char code = EX;      /* Sets code equal to `x' */
const char A = `Z';
```

Now that you know how to declare and initialize character variables, it's time for a demonstration. "Fundamentals of Input and Output." The cout can be used to print both characters and numbers.

The numeric nature of type char variables.

```
1:  /* Demonstrates the numeric nature of char variables */
2:
3:  #include <iostream.h>
4:
5:  /* Declare and initialize two char variables */
6:
7:  char c1 = `a';
```

```
8:  char c2 = 90;
9:
10: main()
11: {
12:     /* Print variable c1 as a character */
13:
14:     cout<<"\n As a character variable c1 is :"<< c1;
15:
16:
17:     /* Do the same for variable c2 */
18:
19:     cout<<"\n As a character, variable c2 is:"<<c2;
20:
21:
22:     return 0;
23: }
As a character, variable c1 is a
As a character, variable c2 is Z
```

Using Strings

Variables of type char can hold only a single character, so they have limited usefulness. You also need a way to store *strings*, which are sequences of characters. Although there is no special data type for strings, C or C++ handles this type of information with arrays of characters.

Arrays of Characters

To hold a string of six characters, for example, you need to declare an array of type char with seven elements. Arrays of type char are declared like arrays of other data types. For example, the statement

```
char string[10];
```

Initializing Character Arrays

character arrays can be initialized when they are declared . Character arrays can be assigned values element by element, as shown here:

```
char string[10] = { `A', `l', `a', `b', `a', `m', `a', `\0' };
```

It's more convenient, however, to use a *literal string*, which is a sequence of characters enclosed in double quotes:

```
char string[10] = "Alabama";
```

When you use a literal string in your program , the compiler automatically adds the terminating null character at the end of the string. If you don't specify the number of subscripts when you declare an array, the compiler calculates the size of the array for you. Thus, the following line creates and initializes an eight-element array:

```
char string[] = "Alabama";
```

Stream Functions

The C and C++ standard library has a variety of functions that deal with stream input and output. Most of these functions come in two varieties: one that always uses one of the standard streams, and one that requires the programmer to specify the stream. standard library's stream input/output functions.

Uses One of the Standard Streams	Description
printf()	Formatted output

puts()	**String output**
putchar()	**Character output**
scanf()	**Formatted input**
gets()	**String input**
getchar()	**Character input**

All these functions require that you include STDLIB.H.

Character Input function

The character input functions read input from a stream one character at a time. When called, each of these functions returns the next character in the stream.
Some character input functions are *buffered*. This means that the operating system holds all characters in a temporary storage space until you press Enter, and then the system sends the characters to the stdin stream. Others are *unbuffered,* meaning that each character is sent to stdin as soon as the key is pressed.

- **Some input functions automatically *echo* each character to stdout as it is received. Others don't echo; the character is sent to stdin and not stdout. Because stdout is assigned to the screen, that's where input is echoed.**

The uses of buffered, unbuffered, echoing, and nonechoing character input are explained in the following sections:
The getchar() Function
The function getchar() obtains the next character from the stream stdin. It provides buffered character input with echo, The use of getchar() is demonstrated in following program . simply displays a single character on-screen.

The getchar() function.

```
1: /* Demonstrates the getchar() function. */
2:
3: #include <stdio.h>   // header file used for getchar and putchar
4:
5: main()
6: {
7:     int ch;
8:
9:     ch = getchar()
10:         putchar(ch);
11:
12:    return 0;
13: }
```

ANALYSIS: On line 9, the getchar() function is called and waits to receive a character from stdin. Because getchar() is a buffered input function, no characters are received until you press Enter. However, each key you press is echoed immediately on the screen. putchar() function is used to display the input character after pressing the enter
Note: the header file for the above function is <stdio.h>

The getch() Function

The getch() function provides unbuffered character input without echo. the prototype for getch() is in the header file <conio.h>, as follows:
Because it is unbuffered, getch() returns each character as soon as the key is pressed, without waiting for the user to press Enter. Because getch() doesn't echo its input, the characters aren't displayed on-screen.
Using the getch() function.

```
1: /* Demonstrates the getch() function. */
2:
3:
4: #include <conio.h>   // used for getch and putch
5:
6: main()
7: {
8:     int ch;
9:
10:     ch = getch();
11:         putch(ch);
12:
13:     return 0;
14:}
```

ANALYSIS: When this program runs, getch() returns each character as soon as you press a key--it doesn't wait for you to press Enter. the only reason that each character is displayed on-screen by calling the putch(). To get a better understanding of how getch() works,you will find that nothing you type is echoed to the screen . The getch() function gets the characters without echoing them to the screen. the characters are being gotten because the original listing used putch() to display them.

The getche() Function

This is a short section, because getche() is exactly like getch(), except that it echoes each character to stdout. When the program runs, each key you press is displayed on-screen twice--once as echoed by getche(), and once as echoed by putch().

Note : the header file for getch,getche,putch is conio.h .

The scanf() Function

The scanf() function takes a variable number of arguments; it requires a minimum of two. The first argument is a format string that uses special characters to tell scanf() how to interpret the input. The second and additional arguments are the addresses of the variable(s) to which the input data is assigned. Here's an example:

```
scanf("%d", &x);
```

The first argument, "%d", is the format string. In this case, %d tells scanf() to look for one signed integer value. The second argument uses the address-of operator (&) to tell scanf() to assign the input value to the variable x. Now you can look at the format string details. The table show the specified characters used in scanf() conversion:

Type	Argument	Meaning of Type
D	int *	A decimal integer.
I	int *	An integer in decimal
O	int *	An integer in octal notation with or without the leading 0.
U	unsigned int	An unsigned decimal integer.

	*	
X	**int ***	**A hexadecimal integer with or without the leading 0X or 0x.**
C	**char ***	**One or more characters are read and assigned sequentially to the memory location indicated by the argument.**
S	**char ***	**A string of nonwhitespace characters is read into the specified memory location, and a terminating \0 is added.**
E,f,g	**float ***	**A floating-point number. Numbers can be input in decimal or scientific notation.**

Screen Output

Screen output functions are divided into three general categories along the same lines as the input functions: character output, line output, and formatted output.

Using the putchar() Function

The prototype for putchar(), which is located in STDIO.H, is as follows:

```
putchar(c);
```

Using puts() for String Output

If You want to display strings on-screen more often than a single character. The library function puts() displays strings up to but not including the terminating null character or spacing. The header file for puts function is STDIO.H

Using the puts() function to display strings.

```
1: /* Demonstrates puts(). */
2:
3: #include <stdio.h>
4: #include<iostream.h>
5:
6:
7:
8:
9: main()
10: {
11:   char  c[100];
12:
13:         cin>>c;
14:         puts(c);
15:
16:     puts("And this is the end!");
17:
18:     return 0;
19: }
```

Using printf() for Formatted Output

the above output functions have displayed characters and strings only. What about numbers? To display numbers, you must use the formatted output functions, printf(). This function can also display strings and characters.

Example of the printf() function:

```
  #include <stdio.h>
```

```
main()
 {
     float d1 = 10000.123;
     int n, f;
     puts("Outputting a number with different field widths.\n");
     printf("%5f\n", d1);
     printf("%10f\n", d1);
     printf("%15f\n", d1);
     printf("%20f\n", d1);
     printf("%25f\n", d1);

     return 0;
 }
Outputting a number with different field widths.
10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047
```

Mathematical functions

There are a number of functions used In the mathematical operations such as :

abs ,cos, exp, log, log10, pow, pow10, sin, sqrt, tan.

The header file used to this functions is <math.h> .

1- abs example:

```
#include <iostream.h>
#include <math.h>
void main()
{
 int  number = -1234;
cout<<number<<abs(number);
 }
```

2-cos example:

```
#include <iostream.h>
#include <math.h>

Void main()
{
  double result;
  double x = 0.5;

  result = cos(x);
  cout<<x<< result;
  }
```

3-sin example:

```
#include <iostream.h>
#include <math.h>
void main()
{
  double result, x = 0.5;

  result = sin(x);
  cout<<x<< result;
  }
```

4-tan example:

```
#include <iostream.h>
#include <math.h>

void main()
{
  double result, x;

  x = 0.5;
  result = tan(x);
  cout<<x<< result;
  }
```

5-Exp example:

```
#include <iostream.h>
#include <math.h>

void main()
{
  double result;
  double x = 4.0;

  result = exp(x);
  cout<<x<<result;
}
```

6-log example:

```
#include <math.h>
#include <iostream.h>
```

```
void main()
{
  double result;
  double x = 8.6872;

  result = log(x);
  cout<<x<< result;
}
```

7-log10 example:

```
#include <math.h>
#include <iostream.h>

Void main()
{
  double result;
  double x = 800.6872;

  result = log10(x);
  cout<<x<< result;
}
```

8-Pow Example:

```
#include <math.h>
#include <iostream.h>

Void main()
{
  double x = 2.0, y = 3.0;

 cout<<x<<y<< pow(x, y);
 }
```

9-Pow10 Example:

```
#include <math.h>
#include <iostream.h>

void main()
{
```

```
  double p = 3.0;

  cout<<p<< pow10(p);

}
```

10-Sqrt Example:

```
#include <math.h>
#include <iostream.h>

void main()
{
  double x = 4.0, result;
  result = sqrt(x);
  cout<<x<< result;
  }
```

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 either computer it is always the same.

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.

New Term: A *character* is a single letter, number, or symbol that takes up one byte of memory.

Determining the size of variable types on your computer.

```
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: }
Output:
The size of an int is:          2 bytes.
The size of a short int is:     2 bytes.
The size of a long int is:      4 bytes.
The size of a char is:          1 bytes.
The size of a float is:         4 bytes.
The size of a double is:        8 bytes.
```

Note: On your computer, the number of bytes presented might be different.

Analysis: The one new feature 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(). Using sizeof().

Constants

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

C++ has two types of constants: literal and symbolic.

Literal Constants

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;

Enumerated Constants

Enumerated constants enable you to create new types and then to define variables of those types whose values are restricted to a set of possible values. For example, you can declare COLOR to be an enumeration, and you can define that there are five values for COLOR: RED, BLUE, GREEN, WHITE, and BLACK.
The syntax for enumerated constants is to write the keyword enum, followed by the type name, an open brace, each of the legal values separated by a comma, and finally a closing brace and a semicolon. Here's an example:

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

This statement performs two tasks:

1. **It makes COLOR the name of an enumeration, that is, a new type.**
2. **It makes RED a symbolic constant with the value 0, BLUE a symbolic constant with the value 1, GREEN a symbolic constant with the value 2, and so forth.**

Every enumerated constant has an integer value. If you don't specify otherwise, the first constant will have the value 0, and the rest will count up from there. Any one of the constants can be initialized with a particular value, however, and those that are not initialized will count upward from the ones before them. Thus, if you write

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

then RED will have the value 100; BLUE, the value 101; GREEN, the value 500; WHITE, the value 501; and BLACK, the value 700.
You can define variables of type COLOR, but they can be assigned only one of the enumerated values (in this case, RED, BLUE, GREEN, WHITE, or BLACK, or else 100, 101, 500, 501, or 700). You can assign any color value to your COLOR variable.

In fact, you can assign any integer value, even if it is not a legal color, although a good compiler will issue a warning if you do. It is important to realize that enumerator variables actually are of type unsigned int, and that the enumerated constants equate to integer variables. It is, however, very convenient to be able to name these values when working with colors, days of the week, or similar sets of values.

Example:

A demonstration of enumerated constants.

```
#include <iostream.h>
int main()
{
    enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Â_Saturday };
    Days  DayOff;
    int x;
    cout << "What day would you like off (0-6)? ";
    cin  >> x;
    DayOff = Days(x);
     If (Dayoff == Sunday )
     cout << “the off is at Sunday” ;
     else
     cout<<”the  off  is not at Sunday”;
     return 0;
}
```

Expressions and Statements:

In C++ a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement). All C++ statements end with a semicolon, even the null statement, which is just the semicolon and nothing else. One of the most common statements is the following assignment statement:

```
x = a + b;
```

New Term: A *null statement* is a statement that does nothing.

White space:

White space (tabs, spaces, and new lines) is generally ignored in statements. The assignment previously discussed could be written as:

```
x=a + b;
```

or as

```
x               =a

+       b       ;
```

New Term: *White space characters* (spaces, tabs, and new lines) cannot be seen. If these characters are printed, you see only the white of the paper.

Blocks and Compound Statements:

Any place you can put a single statement, you can put a compound statement, also called a block. A block begins with an opening brace ({) and ends with a closing brace (}). Although every statement in the block must end with a semicolon, the block itself does not end with a semicolon. For example:

```
{
   temp = a;
   a = b;
   b = temp;
}
```

This block of code acts as one statement and swaps the values in the variables a and b.

Expressions:

Anything that evaluates to a value is an expression in C++. An expression is said to return a value. Thus, 3+2; returns the value 5 and so is an expression. All expressions are statements.

Program Evaluating complex expressions.

```
1:   #include <iostream.h>
2:   int main()
3:   {
4:      int a=0, b=0, x=0, y=35;
5:      cout << "a: " << a << " b: " << b;
6:      cout << " x: " << x << " y: " << y << endl;
7:      a = 9;
8:      b = 7;
9:      y = x = a+b;
10:     cout << "a: " << a << " b: " << b;
11:     cout << " x: " << x << " y: " << y << endl;
12:       return 0;
13:  }
```

```
Output: a: 0 b: 0 x: 0 y: 35
          a: 9 b: 7 x: 16 y: 16
```

Mathematical Operators:

There are five mathematical operators: addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).

Addition and subtraction work as you would expect, although subtraction with unsigned integers can lead to surprising results.

A demonstration of subtraction and integer overflow.

```
1: // Listing demonstrates subtraction and
2: // integer overflow
3: #include <iostream.h>
4:
5: int main()
6: {
```

```
7:    unsigned int difference;
8:    unsigned int bigNumber = 100;
9:    unsigned int smallNumber = 50;
10:   difference = bigNumber - smallNumber;
11:   cout << "Difference is: " << difference;
12:   difference = smallNumber - bigNumber;
13:   cout << "\nNow difference is: " << difference <<endl;
14:       return 0;
15: }
```

Output: Difference is: 50
Now difference is: 4294967246

Analysis: The subtraction operator is invoked on line 10, and the result is printed on line 11, much as we might expect. The subtraction operator is called again on line 12, but this time a large unsigned number is subtracted from a small unsigned number. The result would be negative, but because it is evaluated (and printed) as an unsigned number, the result is an overflow.

Integer Division and Modulus

Integer division is somewhat different from everyday division. When you divide 21 by 4, the result is a real number (a number with a fraction). Integers don't have fractions, and so the "remainder" is lopped off. The answer is therefore 5. To get the remainder, you take 21 modulus 4 (21 % 4) and the result is 1. The modulus operator tells you the remainder after an integer division.

Mathematical Operators

It is not uncommon to want to add a value to a variable, and then to assign the result back into the variable. If you have a variable myAge and you want to increase the value by two, you can write

```
int  myAge = 5;
int  temp;
temp = myAge + 2;  // add 5 + 2 and put it in temp
myAge = temp;           // put it back in myAge
```

This method, however, is terribly convoluted and wasteful. In C++, you can put the same variable on both sides of the assignment operator, and thus the preceding becomes

```
myAge = myAge + 2;
```

which is much better. In algebra this expression would be meaningless, but in C++ it is read as "add two to the value in myAge and assign the result to myAge."
xEven simpler to write, but perhaps a bit harder to read is

```
myAge += 2;
```

The self-assigned addition operator (+=) adds the r value to the l value and then reassigns the result into the l value . This operator is pronounced "plus-equals." The statement would be read "myAge plus-equals two." If myAge had the value 4 to start, it would have 6 after this statement.
There are self-assigned subtraction (-=), division (/=), multiplication (*=), and modulus (%=) operators as well.

Increment and Decrement

The most common value to add (or subtract) and then reassign into a variable is 1. In C++, increasing a value by 1 is called incrementing, and decreasing by 1 is called decrementing. There are special operators to perform these actions.

The increment operator (++) increases the value of the variable by 1, and the decrement operator (--) decreases it by 1. Thus, if you have a variable, C, and you want to increment it, you would use this statement:

```
C++;    // Start with C and increment it.
```

This statement is equivalent to the more verbose statement

```
C = C + 1;
```

which you learned is also equivalent to the moderately verbose statement

```
C += 1;
```

Prefix and Postfix

Both the increment operator (++) and the decrement operator(--) come in two varieties: prefix and postfix. The prefix variety is written before the variable name (++myAge); the postfix variety is written after (myAge++).

In a simple statement, it doesn't much matter which you use, but in a complex statement, when you are incrementing (or decrementing) a variable and then assigning the result to another variable, it matters very much. The prefix operator is evaluated before the assignment, the postfix is evaluated after.

The semantics of prefix is this: Increment the value and then fetch it. The semantics of postfix is different: Fetch the value and then increment the original.

This can be confusing at first, but if x is an integer whose value is 5 and you write:

```
int a = ++x;
```

you have told the compiler to increment x (making it 6) and then fetch that value and assign it to a. Thus, a is now 6 and x is now 6.

If, after doing this, you write:

```
int b = x++;
```

you have now told the compiler to fetch the value in x (6) and assign it to b, and then go back and increment x. Thus, b is now 6, but x is now 7.

A demonstration of prefix and postfix operators.

```
1:  // demonstrates use of
2:  // prefix and postfix increment and
3:  // decrement operators
4:  #include <iostream.h>
5:  int main()
6:  {
7:     int myAge = 39;      // initialize two integers
8:     int yourAge = 39;
9:     cout << "I am: " << myAge << " years old.\n";
10:    cout << "You are: " << yourAge << " years old\n";
11:    myAge++;         // postfix increment
12:    ++yourAge;       // prefix increment
13:    cout << "One year passes...\n";
14:    cout << "I am: " << myAge << " years old.\n";
15:    cout << "You are: " << yourAge << " years old\n";
16:    cout << "Another year passes\n";
17:    cout << "I am: " << myAge++ << " years old.\n";
18:    cout << "You are: " << ++yourAge << " years old\n";
```

```
19:    cout << "Let's print it again.\n";
20:    cout << "I am: " << myAge << " years old.\n";
21:    cout << "You are: " << yourAge << " years old\n";
22:     return 0;
23: }
```

Output:

```
 I am      39 years old
You are   39 years old
One year passes
I am      40 years old
You are   40 years old
Another year passes
I am      40 years old
You are   41 years old
Let's print it again
I am      41 years old
You are   41 years old
```

Analysis: On lines 7 and 8, two integer variables are declared, and each is initialized with the value 39. Their values are printed on lines 9 and 10. On line 11, myAge is incremented using the postfix increment operator, and on line 12, yourAge is incremented using the prefix increment operator. The results are printed on lines 14 and 15, and they are identical (both 40). On line 17, myAge is incremented as part of the printing statement, using the postfix increment operator. Because it is postfix, the increment happens after the print, and so the value 40 is printed again. In contrast, on line 18, yourAge is incremented using the prefix increment operator. Thus, it is incremented before being printed, and the value displays as 41. Finally, on lines 20 and 21, the values are printed again. Because the increment statement has completed, the value in myAge is now 41, as is the value in yourAge.

Precedence:

In the complex statement:

```
x = 5 + 3 * 8;
```

which is performed first, the addition or the multiplication? If the addition is performed first, the answer is 8 * 8, or 64. If the multiplication is performed first, the answer is 5 + 24, or 29.

"Operator Precedence." Multiplication has higher precedence than addition, and thus the value of the expression is 29. When two mathematical operators have the same precedence, they are performed in left-to-right order. Thus x = 5 + 3 + 8 * 9 + 6 * 4;

is evaluated multiplication first, left to right. Thus, 8*9 = 72, and 6*4 = 24.

Now the expression is essentially :x = 5 + 3 + 72 + 24;

Now the addition, left to right, is 5 + 3 = 8; 8 + 72 = 80; 80 + 24 = 104.

Be careful with this. Some operators, such as assignment, are evaluated in right-to-left order! In any case, what if the precedence order doesn't meet your needs? Consider the expression:

```
TotalSeconds = NumMinutesToThink + NumMinutesToType * 60
```

In this expression, you do not want to multiply the NumMinutesToType variable by 60 and then add it to NumMinutesToThink. You want to add the two variables to get the total number of minutes, and then you want to multiply that number by 60 to get the total seconds.
In this case, you use parentheses to change the precedence order. Items in parentheses are evaluated at a higher precedence than any of the mathematical operators. Thus
TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60

Relational Operators:

There are six relational operators: equals (==), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), and not equals (!=).

The Relational Operators.

Name	*Operator*	*Sample*	*Evaluates*
Equals	==	**100 == 50;**	**False**
		50 == 50;	**true**
Not Equals	!=	**100 != 50;**	**true**
		50 != 50;	**false**
Greater Than	>	**100 > 50;**	**true**
		50 > 50;	**false**
Greater Than Or Equals	>=	**100 >= 50;**	**true**
		50 >= 50;	**true**
Less Than	<	**100 < 50;**	**false**
		50 < 50;	**false**
Less Than Or Equals	<=	**100 <= 50;**	**false**
		50 <= 50;	**true**

> **DO remember that relational operators return the value 1 (true) or 0 (false). DON'T confuse the assignment operator (=) with the equals relational operator (==). This is one of the most common C++ programming mistakes--be on guard for it.**

The if Statement:

Normally, your program flows along line by line in the order in which it appears in your source code. The if statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.
The simplest form of an if statement is this:

```
if (expression)
   statement;
```

The expression in the parentheses can be any expression at all, but it usually contains one of the relational expressions. If the expression has the value 0, it is considered false, and the

statement is skipped. If it has any nonzero value, it is considered true, and the statement is executed. Consider the following example:

```
if (bigNumber > smallNumber)
   bigNumber = smallNumber;
```

This code compares bigNumber and smallNumber. If bigNumber is larger, the second line sets its value to the value of smallNumber.

Because a block of statements surrounded by braces is exactly equivalent to a single statement, the following type of branch can be quite large and powerful:

```
if (expression)
{
   statement1;
   statement2;
   statement3;
}
```

Here's a simple example of this usage:

```
if (bigNumber > smallNumber)
{
   bigNumber = smallNumber;
   cout << "bigNumber: " << bigNumber << "\n";
   cout << "smallNumber: " << smallNumber << "\n";
}
```

This time, if bigNumber is larger than smallNumber, not only is it set to the value of smallNumber, but an informational message is printed.

A demonstration of branching based on relational operators.

```
1:  // demonstrates if statement
2:  // used with relational operators
3:  #include <iostream.h>
4:  int main()
5:  {
6:       int  RedSoxScore, YankeesScore;
7:       cout << "Enter the score for the Red Sox: ";
8:       cin >> RedSoxScore;
9:
10:      cout << "\nEnter the score for the Yankees: ";
11:      cin >> YankeesScore;
12:
13:      cout << "\n";
14:
15:      if (RedSoxScore > YankeesScore)
16:          cout << "Go Sox!\n";
17:
18:      if (RedSoxScore < YankeesScore)
19:      {
20:          cout << "Go Yankees!\n";
21:          cout << "Happy days in New York!\n";
22:      }
23:
24:      if (RedSoxScore == YankeesScore)
```

```
25:     {
26:         cout << "A tie? Naah, can't be.\n";
27:         cout << "Give me the real score for the Yanks: ";
28:         cin >> YankeesScore;
29:
30:         if (RedSoxScore > YankeesScore)
31:             cout << "Knew it! Go Sox!";
32:
33:         if (YankeesScore > RedSoxScore)
34:             cout << "Knew it! Go Yanks!";
35:
36:         if (YankeesScore == RedSoxScore)
37:             cout << "Wow, it really was a tie!";
38:     }
39:
40:     cout << "\nThanks for telling me.\n";
41:      return 0;
42: }
```

Output:

```
 Enter the score for the Red Sox: 10
Enter the score for the Yankees: 10
A tie? Naah, can't be
Give me the real score for the Yanks: 8
Knew it! Go Sox!
Thanks for telling me.
```

Analysis: **This program asks for user input of scores for two baseball teams, which are stored in integer variables. The variables are compared in the if statement on lines 15, 18, And24. If one score is higher than the other, an informational message is printed. If the scores are equal, the block of code that begins on line 24 and ends on line 38 is entered. The second score is requested again, and then the scores are compared again.**

Note that if the initial Yankees score was higher than the Red Sox score, the if statement on line 15 would evaluate as FALSE, and line 16 would not be invoked. The test on line 18 would evaluate as true, and the statements on lines 20 and 21 would be invoked. Then the if statement on line 24 would be tested, and this would be false (if line 18 was true). Thus, the program would skip the entire block, falling through to line 39.

Else:

Often your program will want to take one branch if your condition is true, another if it is false. The method shown so far, testing first one condition and then the other, The keyword else can make for far more readable code:

```
if (expression)
   statement;
else
   statement;
```

Example:

Demonstrating the else keyword.

```
1:  // demonstrates if statement
2:  // with else clause
3:  #include <iostream.h>
4:  int main()
5:  {
6:     int firstNumber, secondNumber;
7:     cout << "Please enter a big number: ";
8:     cin >> firstNumber;
9:     cout << "\nPlease enter a smaller number: ";
10:    cin >> secondNumber;
11:    if (firstNumber > secondNumber)
12:         cout << "\n Thanks!\n";
13:    else
14:         cout << "\n The second is bigger!";
15:
16:       return 0;
17: }
```

Output:

```
 Please enter a big number: 10
Please enter a smaller number: 12
The second is bigger!
```

Analysis: The **if** statement on line 11 is evaluated. If the condition is true , the statement on line 12 is run; if it is false, the statement on line 14 is run. If the **else** clause on line 13 were removed, the statement on line 14 would run whether or not the **if** statement was true. Remember, the **if** statement ends after line 12. If the **else** was not there, line 14 would just be the next line in the program. Remember that either or both of these statements could be replaced with a block of code in braces.

The syntax for the if statement is as follows: Form1

```
if (expression)
   statement;
next statement;
```

If the expression is evaluated as TRUE, the statement is executed and the program continues with the next statement. If the expression is not true, the statement is ignored and the program jumps to the next statement. Remember that the statement can be a single statement ending with a semicolon or a block enclosed in braces. Form 2

```
if (expression)
   statement1;
else
   statement2;
next statement;
```

If the expression evaluates TRUE, statement1 is executed; otherwise, statement2 is executed. Afterwards, the program continues with the next statement.

Example:

```
if (SomeValue < 10)
```

```
  cout << "SomeValue is less than 10");
else
  cout << "SomeValue is not less than 10!");
cout << "Done." << endl;
```

Advanced if Statements

It is worth noting that any statement can be used in an if or else clause, even another if or else statement. Thus, you might see complex if statements in the following form:

```
if (expression1)
{
   if (expression2)
      statement1;
   else
   {
      if (expression3)
         statement2;
      else
         statement3;
   }
}
else
   statement4;
```

This if statement says, "If expression1 is true and expression2 is true, execute statement1. If expression1 is true but expression2 is not true, then if expression3 is true execute statement2. If expression1 is true but expression2 and expression3 are false, execute statement3. Finally, if expression1 is not true, execute statement4.

A complex, nested if statement.

```
1:  // a complex nested
2:  // if statement
3:  #include <iostream.h>
4:  int main()
5:  {
6:     // Ask for two numbers
7:     // Assign the numbers to bigNumber and littleNumber
8:     // If bigNumber is bigger than littleNumber,
9:     // see if they are evenly divisible
10:    // If they are, see if they are the same number
11:
12:    int firstNumber, secondNumber;
13:    cout << "Enter two numbers.\nFirst: ";
14:    cin >> firstNumber;
15:    cout << "\nSecond: ";
16:    cin >> secondNumber;
17:    cout << "\n\n";
18:
19:    if (firstNumber >= secondNumber)
20:    {
21:      if ( (firstNumber % secondNumber) == 0) // evenly divisible?
22:      {
23:          if (firstNumber == secondNumber)
```

```
24:              cout << "They are the same!\n";
25:          else
26:              cout << "They are evenly divisible!\n";
27:      }
28:      else
29:          cout << "They are not evenly divisible!\n";
30:    }
31:    else
32:      cout << "Hey! The second one is larger!\n";
33:       return 0;
34: }
```

Output:

```
 Enter two numbers.
First: 10
Second: 2
They are evenly divisible!
```

Analysis: Two numbers are prompted for one at a time, and then compared. The first if statement, on line 19, checks to ensure that the first number is greater than or equal to the second. If not, the else clause on line 31 is executed. If the first if is true, the block of code beginning on line 20 is executed, and the second if statement is tested, on line 21. This checks to see whether the first number modulo the second number yields no remainder. If so, the numbers are either evenly divisible or equal. The if statement on line 23 checks for equality and displays the appropriate message either way.If the if statement on line 21 fails, the else statement on line 28 is executed.

Using Braces in Nested if Statements

Although it is legal to leave out the braces on if statements that are only a single statement, and it is legal to nest if statements, such as

```
if (x > y)            // if x is bigger than y
   if (x < z)         // and if x is smaller than z
      x = y;          // then set x to the value in z
```

A demonstration of why braces help clarify which else statement goes with which if statement.

```
1:  // demonstrates why braces
2:  // are important in nested if statements
3:  #include <iostream.h>
4:  int main()
5:  {
6:    int x;
7:    cout << "Enter a number less than 10 or greater than 100: ";
8:    cin >> x;
9:    cout << "\n";
10:
11:    if (x > 10)
12:       if (x > 100)
13:          cout << "More than 100, Thanks!\n";
```

```
14:    else                    // not the else intended!
15:       cout << "Less than 10, Thanks!\n";
16:
17:       return 0;
18: }
```

Output: Enter a number less than 10 or greater than 100: 20

Less than 10, Thanks!

Analysis: The programmer intended to ask for a number between 10 and 100, check for the correct value, and then print a thank-you note. If the if statement on line 11 evaluates TRUE, the following statement (line 12) is executed. In this case, line 12 executes when the number entered is greater than 10. Line 12 contains an if statement also. This if statement evaluates TRUE if the number entered is greater than 100. If the number is not greater than 100, the statement on line 13 is executed.If the number entered is less than or equal to 10, the if statement on line 10 evaluates to FALSE. Program control goes to the next line following the if statement, in this case line 16. If you enter a number less than 10, the output is as follows:

Enter a number less than 10 or greater than 100: 9

The else clause on line 14 was clearly intended to be attached to the if statement on line 11, and thus is indented accordingly. the else statement is really attached to the if statement on line 12, and thus this program has a subtle bug.

demonstration of the proper use of braces with an if statement

```
1:   // demonstrates proper use of braces
2:   // in nested if statements
3:   #include <iostream.h>
4:   int main()
5:   {
6:     int x;
7:     cout << "Enter a number less than 10 or greater than 100: ";
8:     cin >> x;
9:     cout << "\n";
10:
11:    if (x > 10)
12:    {
13:       if (x > 100)
14:           cout << "More than 100, Thanks!\n";
15:    }
16:    else                    // not the else intended!
17:       cout << "Less than 10, Thanks!\n";
18:       return 0;
19: }
```

Output: Enter a number less than 10 or greater than 100: 20

Analysis: The braces on lines 12 and 15 make everything between them into one statement, and now the else on line 16 applies to the if on line 11 as intended. The user typed 20, so the if statement on line 11 is true; however, the if statement on

line 13 is false, so nothing is printed. It would be better if the programmer put another else clause after line 14 so that errors would be caught and a message printed.

The Logical Operators:

Operator	*Symbol*	*Example*
AND	**&&**	**Expression1 && expression2**
OR	**||**	**Expression1 || expression2**
NOT	**!**	**!expression**

Logical AND:

A logical AND statement evaluates two expressions, and if both expressions are true, the logical AND statement is true as well. If it is true that you are hungry, AND it is true that you have money, THEN it is true that you can buy lunch. Thus,

```
if ( (x == 5) && (y == 5) )
```

would evaluate TRUE if both x and y are equal to 5, and it would evaluate FALSE if either one is not equal to 5. Note that both sides must be true for the entire expression to be true. Note that the logical AND is two && symbols.

Logical OR:

A logical OR statement evaluates two expressions. If either one is true, the expression is true. If you have money OR you have a credit card, you can pay the bill. You don't need both money and a credit card; you need only one, although having both would be fine as well. Thus,

```
if ( (x == 5) || (y == 5) )
```

evaluates TRUE if either x or y is equal to 5, or if both are.
Note that the logical OR is two || symbols.

Logical NOT:

A logical NOT statement evaluates true if the expression being tested is false. Again, if the expression being tested is false, the value of the test is TRUE! Thus

```
if ( !(x == 5) )
```

is true only if x is not equal to 5. This is exactly the same as writing

```
if (x != 5)
```

Relational Precedence

Relational operators and logical operators, being C++ expressions, each return a value: 1 (TRUE) or 0 (FALSE). Like all expressions, they have a precedence order that determines which relations are evaluated first. This fact is important when determining the value of the statement: if (x > 5 && y > 5 || z > 5)

It might be that the programmer wanted this expression to evaluate TRUE if both x and y are greater than 5 or if z is greater than 5. On the other hand, the programmer might have wanted this expression to evaluate TRUE only if x is greater than 5 and if it is also true that either y is greater than 5 or z is greater than 5. If x is 3, and y and z are both 10, the first interpretation will be true (z is greater than 5, so ignore x and y), but the second will be false . Although precedence will determine which relation is evaluated first, parentheses can both change the order and make the statement clearer:

```
if (  (x > 5)  && (y > 5 ||  z > 5) )
```

Using the values from earlier, this statement is false. Because it is not true that x is greater than 5, the left side of the AND statement fails, and thus the entire statement is false. Remember that an AND statement requires that both sides be true--something isn't both "good tasting" AND "good for you" if it isn't good tasting.

Conditional (Ternary) Operator

The conditional operator (?:) is C++'s only ternary operator; that is, it is the only operator to take three terms.

The conditional operator takes three expressions and returns a value:

```
(expression1) ? (expression2) : (expression3)
```

This line is read as "If expression1 is true, return the value of expression2; otherwise, return the value of expression3." Typically, this value would be assigned to a variable.

Example demonstrate of the conditional operator.

```
1:   // demonstrates the conditional operator
2:   //
3:   #include <iostream.h>
4:   int main()
5:   {
6:      int x, y, z;
7:      cout << "Enter two numbers.\n";
8:      cout << "First: ";
9:      cin >> x;
10:     cout << "\nSecond: ";
11:     cin >> y;
12:     cout << "\n";
13:
14:     if (x > y)
15:       z = x;
16:     else
17:       z = y;
18:
19:     cout << "z: " << z;
20:     cout << "\n";
21:
22:     z =  (x > y) ? x : y;
23:
24:     cout << "z: " << z;
25:     cout << "\n";
26:       return 0;
```

```
27: }
```

Output:

```
 Enter two numbers.
First: 5
Second: 8
z: 8
z: 8
```

Analysis: Three integer variables are created: x, y, and z. The first two are given values by the user. The if statement on line 14 tests to see which is larger and assigns the larger value to z. This value is printed on line 19. The conditional operator on line 22 makes the same test and assigns z the larger value. It is read like this: "If x is greater than y, return the value of x; otherwise, return the value of y." The value returned is assigned to z. That value is printed on line 24. As you can see, the conditional statement is a shorter equivalent to the if...else statement.

Looping

Many programming problems are solved by repeatedly acting on the same data. There are two ways to do this: recursion and iteration. Iteration means doing the same thing again and again. The principal method of iteration is the loop.

The Roots of Looping goto

In C++, a label is just a name followed by a colon (:). The label is placed to the left of a legal C++ statement, and a jump is accomplished by writing goto followed by the label name.

Looping with the keyword goto.

```
1:   // looping
2:   // with goto
3:
4:   #include <iostream.h>
5:
6:   int main()
7:   {
8:         int counter = 0;      // initialize counter
9:   loop:  counter ++;          // top of the loop
10:          cout << "counter: " << counter << "\n";
11:         if (counter < 5)          // test the value
12:            goto loop;              // jump to the top
13:
14:         cout << "Complete. Counter: " << counter << ".\n";
15:      return 0;
16: }
```

```
Output: counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
```

Complete. Counter: 5.

Analysis: On line 8, counter is initialized to 0. The label loop is on line 9, marking the top of the loop. Counter is incremented and its new value is printed. The value of counter is tested on line 11. If it is less than 5, the if statement is true and the goto statement is executed. This causes program execution to jump back to line 9. The program continues looping until counter is equal to 5, at which time it "falls through" the loop and the final output is printed.

The goto Statement

To use the goto statement, you write goto followed by a label name. This causes an unconditioned jump to the label.

Example:

```
if (value > 10)     goto end; if (value < 10)     goto end; cout << "value is Â10!";  end: cout << "done";
```

while Loops

A while loop causes your program to repeat a sequence of statements as long as the starting condition remains true.

while loops.

```
1:   // Looping
2:   // with while
3:
4:   #include <iostream.h>
5:
6:   int main()
7:   {
8:     int counter = 0;              // initialize the condition
9:
10:     while(counter < 5)     // test condition still true
11:     {
12:        counter++;             // body of the loop
13:        cout << "counter: " << counter << "\n";
14:     }
15:
16:     cout << "Complete. Counter: " << counter << ".\n";
17:      return 0;
18: }
Output: counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Complete. Counter: 5.
```

Analysis: This simple program demonstrates the fundamentals of the while loop. A condition is tested, and if it is true, the body of the while loop is executed. In this case, the condition tested on line 10 is whether counter is less than 5. If the condition is true, the body of the loop is executed; on line 12 the counter is incremented, and on line 13 the value is printed. When the conditional statement on line 10 fails (when

counter is no longer less than 5), the entire body of the while loop (lines 11-14) is skipped. Program execution falls through to line 15.

The while Statement

The syntax for the while statement is as follows:

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

condition is any C++ expression, and statement is any valid C++ statement or block of statements. When condition evaluates to TRUE (1), statement is executed, and then condition is tested again. This continues until condition tests FALSE, at which time the while loop terminates and execution continues on the first line below statement.

Example:

```
// count to 10
int x = 0;
while (x < 10)
cout << "X: " << x++;
```

More Complicated while Statements

The condition tested by a while loop can be as complex as any legal C++ expression. This can include expressions produced using the logical && (AND), || (OR), and ! (NOT) operators.

Complex while loops.

```
1:   // program of
2:   // Complex while statements
3:
4:   #include <iostream.h>
5:
6:   int main()
7:   {
8:     unsigned short small;
9:     unsigned long  large;
10:     const unsigned short MAXSMALL=65535;
11:
12:     cout << "Enter a small number: ";
13:     cin >> small;
14:     cout << "Enter a large number: ";
15:     cin >> large;
16:
17:      cout << "small: " << small << "...";
18:
19:     // for each iteration, test three conditions
20:     while (small < large && large > 0 && small < MAXSMALL)
21:
22:     {
23:       if (small % 5000 == 0)  // write a dot every 5k lines
24:         cout << ".";
25:
26:       small++;
27:
28:       large-=2;
29:     }
```

```
30:
31:     cout << "\nSmall: " << small << " Large: " << large << endl;
32:    return 0;
33: }
```

```
Output: Enter a small number: 2
Enter a large number: 100000
small: 2.....
Small: 33335 Large: 33334
```

Analysis: This program is a game. Enter two numbers, one small and one large. The smaller number will count up by ones, and the larger number will count down by twos. The goal of the game is to guess when they'll meet. On lines 12-15, the numbers are entered. Line 20 sets up a while loop, which will continue only as long as three conditions are met: small is not bigger than large.large isn't negative.small doesn't overrun the size of a small integer (MAXSMALL). On line 23, the value in small is calculated modulo 5,000. This does not change the value in small; however, it only returns the value 0 when small is an exact multiple of 5,000. Each time it is, a dot (.) is printed to the screen to show progress. On line 26, small is incremented, and on line 28, large is decremented by 2.When any of the three conditions in the while loop fails, the loop ends and execution of the program continues after the while loop's closing brace on line 29.

continue and break

At times you'll want to return to the top of a `while` loop before the entire set of statements in the `while` loop is executed. The `continue` statement jumps back to the top of the loop.
At other times, you may want to exit the loop before the exit conditions are met. The `break` statement immediately exits the `while` loop, and program execution resumes after the closing brace.

Program of break and continue.

```
1:   // program
2:   // Demonstrates break and continue
3:
4:   #include <iostream.h>
5:
6:   int main()
7:   {
8:     unsigned short small;
9:     unsigned long  large;
10:     unsigned long  skip;
11:     unsigned long target;
12:     const unsigned short MAXSMALL=65535;
13:
14:     cout << "Enter a small number: ";
15:     cin >> small;
16:     cout << "Enter a large number: ";
17:     cin >> large;
18:     cout << "Enter a skip number: ";
19:     cin >> skip;
20:     cout << "Enter a target number: ";
21:     cin >> target;
```

```
22:
23:   cout << "\n";
24:
25:    // set up 3 stop conditions for the loop
26:     while (small < large && large > 0 && small < 65535)
27:
28:     {
29:
30:       small++;
31:
32:       if (small % skip == 0)  // skip the decrement?
33:       {
34:         cout << "skipping on " << small << endl;
35:         continue;
36:       }
37:
38:       if (large == target)    // exact match for the target?
39:       {
40:         cout << "Target reached!";
41:         break;
42:       }
43:
44:       large-=2;
45:     }               // end of while loop
46:
47:     cout << "\nSmall: " << small << " Large: " << large << endl;
48:   return 0;
49: }
```

Output:

```
Enter a small number: 2
Enter a large number: 20
Enter a skip number: 4
Enter a target number: 6
skipping on 4
skipping on 8
Small: 10 Large: 8
```

Analysis: In this play, the user lost; small became larger than large before the target number of 6 was reached. On line 26, the while conditions are tested. If small continues to be smaller than large, large is larger than 0, and small hasn't overrun the maximum value for a small int, the body of the while loop is entered. On line 32, the small value is taken modulo the skip value. If small is a multiple of skip, the continue statement is reached and program execution jumps to the top of the loop at line 26. This effectively skips over the test for the target and the decrement of large. On line 38, target is tested against the value for large. If they are the same, the user has won. A message is printed and the break statement is reached. This causes an immediate break out of the while loop, and program execution resumes on line 46.

The continue Statement

continue; causes a while or for loop to begin again at the top of the loop.

Example:

```
if (value > 10)
    goto end;
if (value < 10)
    goto end;
cout << "value is 10!";
end:
cout << "done";
```

The break Statement

Break ; causes the immediate end of a while or for loop. Execution jumps to the closing brace.

Example:

```
while (condition)
{
   if (condition2)
      break ;
   // statements;
}
```

while (1) Loops

The condition tested in a while loop can be any valid C++ expression. As long as that condition remains true, the while loop will continue. You can create a loop that will never end by using the number 1 for the condition to be tested. Since 1 is always true, the loop will never end, unless a break statement is reached.

while (1) loops.

```
1:   // program
2:   // Demonstrates a while true loop
3:
4:   #include <iostream.h>
5:
6:   int main()
7:   {
8:     int counter = 0;
9:
10:     while (1)
11:     {
12:        counter ++;
13:        if (counter > 10)
14:           break;
15:     }
16:     cout << "Counter: " << counter << "\n";
17:      return 0;
18:
```

Output: Counter: 11

Analysis: On line 10, a while loop is set up with a condition that can never be false. The loop increments the counter variable on line 12 and then on line 13 tests to see whether counter has gone past 10. If it hasn't, the while loop iterates. If counter is greater than 10, the break on line 14 ends the while loop,

and program execution falls through to line 16, where the results are printed.

do...while Loops

It is possible that the body of a while loop will never execute. The while statement checks its condition before executing any of its statements, and if the condition evaluates false, the entire body of the while loop is skipped.

the body of the while Loop.

```
1:    // program
2:     // Demonstrates skipping the body of
3:     // the while loop when the condition is false.
4:
5:     #include <iostream.h>
6:
7:     int main()
8:     {
9:        int counter;
10:       cout << "How many hellos?: ";
11:       cin >> counter;
12:       while (counter > 0)
13:       {
14:          cout << "Hello!\n";
15:          counter--;
16:       }
17:       cout << "Counter is OutPut: " << counter;
18:        return 0;
19: }
```

```
Output:
How many hellos?: 2
Hello!
Hello!
Counter is OutPut: 0

How many hellos?: 0
Counter is OutPut: 0
```

Analysis: The user is prompted for a starting value on line 10. This starting value is stored in the integer variable counter. The value of counter is tested on line 12, and decremented in the body of the while loop. The first time through counter was set to 2, and so the body of the while loop ran twice. The second time through, however, the user typed in 0. The value of counter was tested on line 12 and the condition was false; counter was not greater than 0. The entire body of the while loop was skipped, and Hello was never printed.

do...while

The do...while loop executes the body of the loop before its condition is tested and ensures that the body always executes at least one time.

Program Demonstrates do...while loop.

```
1:     // program
```

```
2:    // Demonstrates do while
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:      int counter;
9:      cout << "How many hellos? ";
10:     cin >> counter;
11:     do
12:     {
13:       cout << "Hello\n";
14:       counter--;
15:     }  while (counter >0 );
16:     cout << "Counter is: " << counter << endl;
17:     return 0;
18: }
Output: How many hellos? 2
Hello
Hello
Counter is: 0
```

Analysis: **The user is prompted for a starting value on line 9, which is stored in the integer variable counter. In the do...while loop, the body of the loop is entered before the condition is tested, and therefore the body of the loop is guaranteed to run at least once. On line 13 the message is printed, on line 14 the counter is decremented, and on line 15 the condition is tested. If the condition evaluates TRUE, execution jumps to the top of the loop on line 13; otherwise, it falls through to line 16. The continue and break statements work in the do...while loop exactly as they do in the while loop. The only difference between a while loop and a do...while loop is when the condition is tested.**

The do...while Statement

The syntax for the do...while statement is as follows:

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

statement is executed, and then condition is evaluated. If condition is TRUE, the loop is repeated; otherwise, the loop ends. The statements and conditions are otherwise identical to the while loop.

Example 1:

```
// count to 10
int x = 0;
do
cout << "X: " << x++;
while (x < 10)
```

Example 2:

```
// print lowercase alphabet.
```

```
char ch = `a';
do
{
cout << ch << ` `;
ch++;
} while ( ch <= `z' );
```

for Loops

When programming while loops, you'll often find yourself setting up a starting condition, testing to see if the condition is true, and incrementing or otherwise changing a variable each time through the loop.

```
1:   // program
2:   // Looping with while
3:
4:   #include <iostream.h>
5:
6:   int main()
7:   {
8:     int counter = 0;
9:
10:     while(counter < 5)
11:     {
12:         counter++;
13:         cout << "Looping!  ";
14:     }
15:
16:     cout << "\nCounter: " << counter << ".\n";
17:      return 0;
18: }
```

Output: Looping! Looping! Looping! Looping! Looping!
Counter: 5.

Analysis: The condition is set on line 8: counter is initialized to 0. On line 10, counter is tested to see whether it is less than 5. counter is incremented on line 12. On line 16, a simple message is printed, but you can imagine that more important work could be done for each increment of the counter. A for loop combines three steps into one statement. The three steps are initialization, test, and increment. A for statement consists of the keyword for followed by a pair of parentheses. Within the parentheses are three statements separated by semicolons.

Demonstrating the for loop.

```
1:     // program
2:     // Looping with for
3:
4:     #include <iostream.h>
5:
6:     int main()
7:     {
8:       int counter;
```

```
9:      for (counter = 0; counter < 5; counter++)
10:        cout << "Looping! ";
11:
12:      cout << "\nCounter: " << counter << ".\n";
13:       return 0;
14: }
```

Output: Looping! Looping! Looping! Looping! Looping!
Counter: 5.

Analysis: The for statement on line 8 combines the initialization of counter, the test that counter is less than 5, and the increment of counter all into one line. The body of the for statement is on line 9.

The for Statement

The syntax for the for statement is as follows:

```
for (initialization; test; action )
statement;
```

The initialization statement is used to initialize the state of a counter, or to otherwise prepare for the loop.

Example 1:

```
// print Hello ten times
for (int i = 0; i<10; i++)
cout << "Hello! ";
```

Example 2:

```
for (int i = 0; i < 10; i++)
{
   cout << "Hello!" << endl;
   cout << "the value of i is: " << i << endl;
}
```

Advanced for Loops

for statements are powerful and flexible. The three independent statements (initialization, test, and action) lend themselves to a number of variations.
A for loop works in the following sequence:

1. **Performs the operations in the initialization.**
2. **Evaluates the condition.**
3. **If the condition is TRUE, executes the action statement and the loop.**

After each time through, the loop repeats steps 2 and 3. Multiple Initialization and Increments It is not uncommon to initialize more than one variable, to test a compound logical expression, and to execute more than one statement. The initialization and the action may be replaced by multiple C++ statements, each separated by a comma.

Demonstrating multiple statements in for loops.

```
1:  //program
2:  // demonstrates multiple statements in
```

```
3:  // for loops
4:
5: #include <iostream.h>
6:
7:  int main()
8:  {
9:      for (int i=0, j=0; i<3; i++, j++)
10:         cout << "i: " << i << " j: " << j << endl;
11:     return 0;
12: }
```

Output:

```
 i: 0  j: 0
i: 1  j: 1
i: 2  j: 2
```

Analysis: **On line 9, two variables, i and j, are each initialized with the value 0. The test (i<3) is evaluated, and because it is true, the body of the for statement is executed, and the values are printed. Finally, the third clause in the for statement is executed, and i and j are incremented.Once line 10 completes, the condition is evaluated again, and if it remains true the actions are repeated (i and j are again incremented), and the body of loop is executed again. This continues until the test fails, in which case the action statement is not executed, and control falls out of the loop. Null Statements in for Loops Any or all of the statements in a for loop can be null. To accomplish this, use the semicolon to mark where the statement would have been. To create a for loop that acts exactly like a while loop, leave out the first and third statements.**

The Null statements in for loops.

```
1:   //  program
2:   // For loops with null statements
3:
4:   #include <iostream.h>
5:
6:   int main()
7:   {
8:      int counter = 0;
9:
10:      for( ; counter < 5; )
11:      {
12:        counter++;
13:        cout << "Looping!  ";
14:      }
15:
16:      cout << "\nCounter: " << counter << ".\n";
17:      return 0;
18: }
```

output: Looping! Looping! Looping! Looping! Looping!
Counter: 5.

Analysis: You may recognize this as exactly like the while loop illustrated On line 8, the counter variable is initialized. The for statement on line 10 does not initialize any values, but it does include a test for counter < 5. There is no increment statement, so this loop behaves exactly as if it had been written:
while (counter < 5)

Illustrating empty for loop statement.

```
1:    //program
2:    //empty for loop statement
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:       int counter=0;       // initialization
9:       int max;
10:       cout << "How many hellos?";
11:       cin >> max;
12:       for (;;)         // a for loop that doesn't end
13:       {
14:          if (counter < max)       // test
15:          {
16:            cout << "Hello!\n";
17:            counter++;          // increment
18:          }
19:          else
20:             break;
21:       }
22:      return 0;
23: }
```

Output:
How many hellos?3
Hello!
Hello!
Hello!

Analysis: The for loop has now been pushed to its absolute limit. Initialization, test, and action have all been taken out of the for statement. The initialization is done on line 8, before the for loop begins. The test is done in a separate if statement on line 14, and if the test succeeds, the action, an increment to counter, is performed on line 17. If the test fails, breaking out of the loop occurs on line 20. While this particular program is somewhat absurd, there are times when a for(;;) loop or a while (1) loop is just what you'll want. You'll see an example of a more reasonable use of such loops when switch statements are discussed later today.

Empty for Loops

So much can be done in the header of a for statement, there are times you won't need the body to do anything at all. In that case, be sure to put a null statement (;) as the body of the loop. The semicolon can be on the same line as the header, but this is easy to overlook.

Illustrates the null statement in a for loop.

```
1:    //program
2:    //Demonstrates null statement
3:    // as body of for loop
4:
5:    #include <iostream.h>
6:    int main()
7:    {
8:       for (int i = 0; i<5; cout << "i: " << i++ << endl)
9:       ;
10:      return 0;
11: }
Output:
 i: 0
i: 1
i: 2
i: 3
i: 4
```

Analysis: The for loop on line 8 includes three statements: the initialization statement establishes the counter i and initializes it to 0. The condition statement tests for i<5, and the action statement prints the value in i and increments it. There is nothing left to do in the body of the for loop, so the null statement (;) is used. Note that this is not a well-designed for loop: the action statement is doing far too much. This would be better rewritten as

```
8:       for (int i = 0; i<5; i++)
9:       cout << "i: " << i << endl;
```

Nested Loops

Loops may be nested, with one loop sitting in the body of another. The inner loop will be executed in full for every execution of the outer loop.

Program Illustrates nested for loops.

```
1:   //program
2:   //Illustrates nested for loops
3:
4:   #include <iostream.h>
5:
6:   int main()
7:   {
8:      int rows, columns;
9:      char theChar;
10:      cout << "How many rows? ";
11:      cin >> rows;
12:      cout << "How many columns? ";
13:      cin >> columns;
14:      cout << "What character? ";
15:      cin >> theChar;
16:      for (int i = 0; i<rows; i++)
17:      {
```

```
18:         for (int j = 0; j<columns; j++)
19:             cout << theChar;
20:         cout << "\n";
21:       }
22:      return 0;
23: }
```

```
Output: How many rows? 4
How many columns? 12
What character? x
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
```

Analysis: The user is prompted for the number of rows and columns and for a character to print. The first for loop, on line 16, initializes a counter (i) to 0, and then the body of the outer for loop is run. On line 18, the first line of the body of the outer for loop, another for loop is established. A second counter (j) is also initialized to 0, and the body of the inner for loop is executed. On line 19, the chosen character is printed, and control returns to the header of the inner for loop. Note that the inner for loop is only one statement (the printing of the character). The condition is tested (j < columns) and if it evaluates true, j is incremented and the next character is printed. This continues until j equals the number of columns. Once the inner for loop fails its test, in this case after 12 Xs are printed, execution falls through to line 20, and a new line is printed. The outer for loop now returns to its header, where its condition (i < rows) is tested. If this evaluates true, i is incremented and the body of the loop is executed. In the second iteration of the outer for loop, the inner for loop is started over. Thus, j is reinitialized to 0 and the entire inner loop is run again.

switch Statements

switch statements allow you to branch on any of a number of different values. The general form of the switch statement is:

```
switch (expression)
{
case valueOne: statement;
               break;
case valueTwo: statement;
               break;
....
case valueN:   statement;
               break;
default:       statement;
}
```

expression is any legal C++ expression, and the statements are any legal C++ statements or block of statements. switch evaluates expression and compares the result to each of the case values. If one of the case values matches the expression, execution jumps to those statements and continues to the end of the switch block, unless a break statement is encountered.

Demonstrating the switch statement.

```
1:  //program
2:  // Demonstrates switch statement
3:
4:  #include <iostream.h>
5:
6:  int main()
7:  {
8:    unsigned short int number;
9:    cout << "Enter a number between 1 and 5: ";
10:    cin >> number;
11:    switch (number)
12:    {
13:       case 0:   cout << "Too small, sorry!";
14:                 break;
15:       case 5:  cout << "Good job!\n";  // fall through
16:       case 4:  cout << "Nice Pick!\n"; // fall through
17:       case 3:  cout << "Excellent!\n"; // fall through
18:       case 2:  cout << "Masterful!\n"; // fall through
19:       case 1:  cout << "Incredible!\n";
20:                break;
21:       default: cout << "Too large!\n";
22:                break;
23:    }
24:    cout << "\n\n";
25:     return 0;
26: }
```

Output:
Enter a number between 1 and 5: 3
Excellent!
Masterful!
Incredible!

Output:
Enter a number between 1 and 5: 8
Too large!

Analysis: **The user is prompted for a number. That number is given to the switch statement. If the number is 0, the case statement on line 13 matches, the message Too small, sorry! is printed, and the break statement ends the switch. If the value is 5, execution switches to line 15 where a message is printed, and then falls through to line 16, another message is printed, and so forth until hitting the break on line 20. The net effect of these statements is that for a number between 1 and 5, that many messages are printed. If the value of number is not 0-5, it is assumed to be too large, and the default statement is invoked on line 21.**

The switch Statement

The syntax for the switch statement is as follows:

```
switch (expression)
```

```
{
case valueOne: statement;
case valueTwo: statement;
....
case valueN: statement
default: statement;
}
```

The switch statement allows for branching on multiple values of expression. The expression is evaluated, and if it matches any of the case values, execution jumps to that line. Execution continues until either the end of the switch statement or a break statement is encountered. If expression does not match any of the case statements, and if there is a default statement, execution switches to the default statement, otherwise the switch statement ends.

Example 1:

```
switch (choice)
{
case 0:
      cout << "Zero!" << endl;
      break
case 1:
      cout << "One!" << endl;
     break;
case 2:
     cout << "Two!" << endl;
default:
     cout << "Default!" << endl;
```

Example 2:

```
switch (choice)
{
choice 0:
choice 1:
choice 2:
     cout << "Less than 3!";
     break;
choice 3:
     cout << "Equals 3!";
     break;
default:
     cout << "greater than 3!";
}
```

Using Numeric Arrays

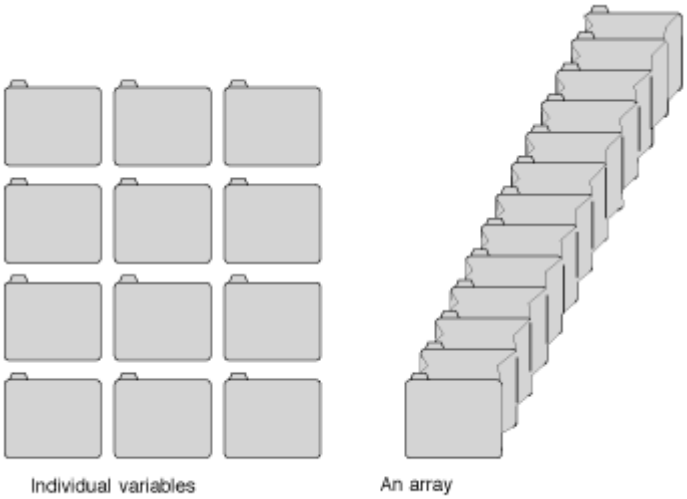
Arrays are a type of data storage that you often use in C++ programs. You had a brief introduction to arrays :

- **What an array is**
- **The definition of single- and multidimensional numeric arrays**
- **How to declare and initialize arrays**

What Is an Array?

An *array* is a collection of data storage locations, each having the same data type and the same name. Each storage location in an array is called an *array element*. Why do you need arrays in your programs? This question can be answered with an example. If you're keeping track of your business expenses for 2006 and filing your receipts by month, you could have a separate folder for each month's receipts, but it would be more convenient to have a single folder with 12 compartments.

Extend this example to computer programming. Imagine that you're designing a program to keep track of your business expenses. The program could declare 12 separate variables, one for each month's expense total. This approach is analogous to having 12 separate folders for your receipts. Good programming practice, however, would utilize an array with 12 elements, storing each month's total in the corresponding array element. This approach is comparable to filing your receipts in a single folder with 12 compartments. The figure below illustrates the difference between using individual variables and an array.



Variables are like individual folders, where as an array is like a single folder with many compartments.

Declaring Arrays

Arrays can have any legal variable name, but they cannot have the same name as another variable or array within their scope. Therefore ,you cannot have an array named myCats[5] and a variable named myCats at the same time.

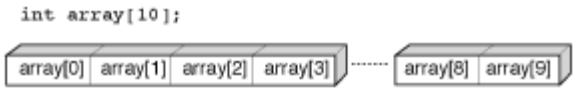
Single-Dimensional Arrays

A *single-dimensional array* has only a single subscript. A *subscript* is a number in brackets that follows an array's name. This number can identify the number of individual elements in the array. An example should make this clear. For the business expenses program, you could use the following line to declare an array of type float:

```
float expenses[12];
```

The array is named expenses, and it contains 12 elements. Each of the 12 elements is the exact equivalent of a single float variable. All of C++ data types can be used for arrays. C++ array elements are always numbered starting at 0, so the 12 elements of expenses are numbered 0 through 11.

When you declare an array, the compiler sets aside a block of memory large enough to hold the entire array. Individual array elements are stored in sequential memory locations, as shown in Figure below .



Array elements are stored in sequential memory locations.

The location of array declarations in your source code is important. As with non array variables, the declaration's location affects how your program can use the array.

An array element can be used in your program anywhere a non array variable of the same type can be used. Individual elements of the array are accessed by using the array name followed by the element subscript enclosed in square brackets. For example, the following statement stores the value 89.95 in the second array element (remember, the first array element is expenses[0], not expenses[1]):

```
expenses[1] = 89.95;
```

Likewise, the statement

```
expenses[10] = expenses[11];
```

assigns a copy of the value that is stored in array element expenses[11] into array element expenses[10]. When you refer to an array element, the array subscript can be a literal constant, as in these examples. However, your programs might often use a subscript that is a C++ integer variable or expression, or even another array element. Here are some examples:

```
float expenses[100];
int a[10];
/* additional statements go here */
expenses[i] = 100;        /* i is an integer variable */
expenses[2 + 3] = 100;    /* equivalent to expenses[5] */
expenses[a[2]] = 100;     /* a[] is an integer array */
```

That last example might need an explanation. If, for instance, you have an integer array named a[] and the value 8 is stored in element a[2], writing `expenses[a[2]]`
has the same effect as writing `expenses[8];`

When you use arrays, keep the element numbering scheme in mind: In an array of *n* elements, the allowable subscripts range from 0 to *n*-1. If you use the subscript value *n*, you might get program errors. The C++ compiler doesn't recognize whether your program uses an array subscript that is out of bounds. Your program compiles and links, but out-of-range subscripts generally produce erroneous results.

WARNING: Remember that array elements start with 0, not 1. Also remember that the last element is one less than the number of elements in the array. For example, an array with 10 elements contains elements 0 through 9.

Sometimes you might want to treat an array of n elements as if its elements were numbered 1 through *n*. For instance, in the previous example, a more natural method might be to store January's expense total in expenses[1], February's in expenses[2], and so on. The simplest way to do this is to declare the array with one more element than needed, and ignore element 0. In this case, you would declare the array as follows. You could also store some related data in element 0 (the yearly expense total, perhaps).

```
float expenses[13];
```

The following program EXPENSES.Cpp demonstrates the use of an array. This is a simple program with no real practical use; it's for demonstration purposes only.

Program demonstrates the use of an array.

```
1:  /* EXPENSES.Cpp - Demonstrates use of an array */
2:
3:  #include <conio.h>
4:
5:  /* Declare an array to hold expenses, and a counter variable */
6:
7:  float expenses[13];
8:  int count;
9:
10: main()
11: {
12:    /* Input data from keyboard into array */
13:
14:    for (count = 1; count < 13; count++)
15:    {
16:       cout<<"Enter expenses for month : "<<count);
17:       cin>>expenses[count];
18:    }
19:
20:    /* Print array contents */
21:
22:    for (count = 1; count < 13; count++)
23:    {
24:       cout<<"Month = \n"<< count<<"\t"<<expenses[count];
25:    }
26:    return 0;
27: }
```

Output:

```
Enter expenses for month 1: 100
Enter expenses for month 2: 200.12
Enter expenses for month 3: 150.50
Enter expenses for month 4: 300
Enter expenses for month 5: 100.50
Enter expenses for month 6: 34.25
Enter expenses for month 7: 45.75
Enter expenses for month 8: 195.00
Enter expenses for month 9: 123.45
Enter expenses for month 10: 111.11
Enter expenses for month 11: 222.20
Enter expenses for month 12: 120.00
Month 1 = $100.00
Month 2 = $200.12
Month 3 = $150.50
Month 4 = $300.00
Month 5 = $100.50
Month 6 = $34.25
Month 7 = $45.75
```

```
Month 8 = $195.00
Month 9 = $123.45
Month 10 = $111.11
Month 11 = $222.20
Month 12 = $120.00
```

ANAALYSIS: When you run EXPENSES.Cpp , the program prompts you to enter expenses for months 1 through 12. The values you enter are stored in an array. You must enter a value for each month. After the 12th value is entered, the array contents are displayed on-screen.

The flow of the program is similar to listings you've seen before. Line 1 starts with a comment that describes what the program does. Notice that the name of the program, EXPENSES.Cpp, is included. When the name of the program is included in a comment, you know which program you're viewing. This is helpful when you're reviewing printouts of a listing.

Line 5 contains an additional comment explaining the variables that are being declared. In line 7, an array of 13 elements is declared. In this program, only 12 elements are needed, one for each month, but 13 have been declared. The for loop in lines 14 through 18 ignores element 0. This lets the program use elements 1 through 12, which relate directly to the 12 months. Going back to line 8, a variable, count, is declared and is used throughout the program as a counter and an array index.

The program's main() function begins on line 10. As stated earlier, this program uses a for loop to print a message and accept a value for each of the 12 months. Notice that in line 17, the cin function uses an array element. In line 7, the expenses array was declared as float.

Lines 22 through 25 contain a second for loop that prints the values just entered. An additional formatting command has been added to the cout function so that the expenses values print in a more orderly fashion.

DON'T forget that array subscripts start at element 0.

DO use arrays instead of creating several variables that store the same thing. For example, if you want to store total sales for each month of the year, create an array with 12 elements to hold sales rather than creating a sales variable for each month.

Initializing Arrays

You can initialize a simple array of built-in types, such as integers and characters, when you first declare the array. After the array name, you put an equal sign (=) and a list of comma-separated values enclosed in braces. For example,

```
int IntegerArray[5] = { 10, 20, 30, 40, 50 };
```

declares `IntegerArray` to be an array of five integers. It assigns `IntegerArray[0]` the value `10`, `IntegerArray[1]` the value `20`, and so forth.

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write `int IntegerArray[] = { 10, 20, 30, 40, 50 };`

you will create exactly the same array as you did in the previous example.
If you need to know the size of the array, you can ask the compiler to compute it for you. For example,

```
const USHORT IntegerArrayLength;
IntegerArrayLength = sizeof(IntegerArray)/sizeof(IntegerArray[0]);
```

sets the constant USHORT variable IntegerArrayLength to the result obtained from dividing the size of the entire array by the size of each individual entry in the array. That quotient is the number of members in the array.
You cannot initialize more elements than you've declared for the array. Therefore,

```
Int IntegerArray[5] = { 10, 20, 30, 40, 50, 60};
```

generates a compiler error because you've declared a five-member array and initialized six values. It is legal, however, to write:

```
int IntegerArray[5] = { 10, 20};
```

Although uninitialized array members have no guaranteed values, actually, aggregates will be initialized to 0. If you don't initialize an array member, its value will be set to 0.

DO let the compiler set the size of initialized arrays. DON'T write past the end of the array. DO give arrays meaningful names, as you would with any variable.DO remember that the first member of the array is at offset 0.
You can dimension the array size with a const or with an enumeration.

Program Using consts and enums in arrays.

```
1:    // program
2:    // Dimensioning arrays with consts and enumerations
3:
4:    #include <iostream.h>
5:    int main()
6:    {
7:       enum WeekDays { Sun, Mon, Tue,
8:            Wed, Thu, Fri, Sat, DaysInWeek };
9:       int ArrayWeek[DaysInWeek] = { 10, 20, 30, 40, 50, 60, 70 };
10:
11:       cout << "The value at Tuesday is: " << ArrayWeek[Tue];
12:     return 0;
13: }
Output: The value at Tuesday is: 30
```

Analysis: Line 7 creates an enumeration called WeekDays. It has eight members. Sunday is equal to 0, and DaysInWeek is equal to 7. Line 11 uses the enumerated constant Tue as an offset into the array. Because Tue evaluates to 2, the third element of the array, DaysInWeek[2], is returned and printed in line 11.

Array Elements

You access each of the array elements by referring to an offset from the array name. Array elements are counted from zero. Therefore, the first array element is arrayName[0]. In the LongArray example, LongArray[0] is the first array element, LongArray[1] the second, and so forth.
This can be somewhat confusing. The array SomeArray[3] has three elements. They are SomeArray[0], SomeArray[1], and SomeArray[2]. More generally, SomeArray[n] has n elements that are numbered SomeArray[0] through SomeArray[n-1].

Therefore, LongArray[25] is numbered from LongArray[0] through LongArray[24].

Program Using an integer array.

```
1:    //program using integer Arrays
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       int myArray[5];
7:       int i;
8:       for ( i=0; i<5; i++)  // 0-4
9:       {
10:          cout << "Value for myArray[" << i << "]: ";
11:         cin >> myArray[i];
12:      }
13:      for (i = 0; i<5; i++)
14:         cout << i << ": " << myArray[i] << "\n";
15:    return 0;
16: }
```

Output:

```
Value for myArray[0]:  3
Value for myArray[1]:  6
Value for myArray[2]:  9
Value for myArray[3]:  12
Value for myArray[4]:  15
0: 3
1: 6
2: 9
3: 12
4: 15
```

Analysis: Line 6 declares an array called myArray, which holds five integer variables. Line 8 establishes a loop that counts from 0 through 4, which is the proper set of offsets for a five-element array. The user is prompted for a value, and that value is saved at the correct offset into The array. The first value is saved at myArray[0], the second at myArray[1], and so forth. The second for loop prints each value to the screen.

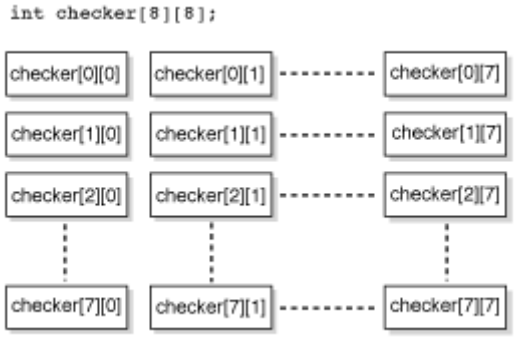
NOTE: Arrays count from 0, not from 1. This is the cause of many bugs in programs written by C++ novices. Whenever you use an array, remember that an array with 10 elements counts from ArrayName[0] to ArrayName[9]. There is no ArrayName[10].

Multidimensional Arrays

A multidimensional array has more than one subscript. A two-dimensional array has two subscripts, a three-dimensional array has three subscripts, and so on. There is no limit to the number of dimensions a C++ array can have. (There *is* a limit on total array size).
For example, you might write a program that plays checkers. The checkerboard contains 64 squares arranged in eight rows and eight columns. Your program could represent the board as a two-dimensional array, as follows:

```
int checker[8][8];
```

The resulting array has 64 elements: checker[0][0], checker[0][1], checker[0][2]...checker[7][6], checker[7][7]. The structure of this two-dimensional array is illustrated in the figure below.



two-dimensional array has a row-and-column structure.

Similarly, a three-dimensional array could be thought of as a cube. Four-dimensional arrays (and higher) are probably best left to your imagination. All arrays, no matter how many dimensions they have, are stored sequentially in memory.

Initializing Multidimensional Arrays

You can initialize multidimensional arrays. You assign the list of values to array elements in order, with the last array subscript changing while each of the former holds steady. Therefore, if you have an array:

```
int theArray[5][3]
```

the first three elements go into `theArray[0]`; the next three into `theArray[1]`; and so forth.

You initialize this array by writing

```
int theArray[5][3] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 }
```

For the sake of clarity, you could group the initializations with braces. For example,

```
int theArray[5][3] = {  {1,2,3},
{4,5,6},
{7,8,9},
{10,11,12},
{13,14,15} };
```

The compiler ignores the inner braces, which make it easier to understand how the numbers are distributed.

Each value must be separated by a comma, without regard to the braces. The entire initialization set must be within braces, and it must end with a semicolon.

The following program Listing a two-dimensional array. The first dimension is the set of numbers from 0 to 5. The second dimension consists of the double of each value in the first dimension.

Program Creating a multidimensional array.

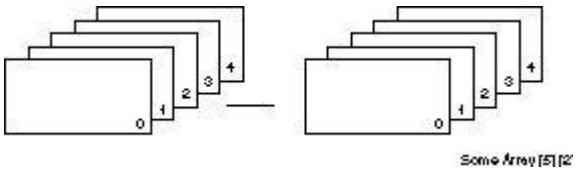
```
1:    #include <iostream.h>
2:    int main()
3:    {
4:      int SomeArray[5][2] = { {0,0}, {1,2}, {2,4}, {3,6}, {4,8}};
5:      for (int i = 0; i<5; i++)
6:        for (int j=0; j<2; j++)
7:        {
```

```
8:            cout << "SomeArray[" << i << "][" << j << "]: ";
9:            cout << SomeArray[i][j]<< endl;
10:        }
11:
12:    return 0;
13: }
```

Output:

```
SomeArray[0][0]: 0
SomeArray[0][1]: 0
SomeArray[1][0]: 1
SomeArray[1][1]: 2
SomeArray[2][0]: 2
SomeArray[2][1]: 4
SomeArray[3][0]: 3
SomeArray[3][1]: 6
SomeArray[4][0]: 4
SomeArray[4][1]: 8
```

Analysis: Line 4 declares SomeArray to be a two-dimensional array. The first dimension consists of five integers; the second dimension consists of two integers. This creates a 5x2 grid, as Figure shown below.



The values are initialized in pairs, although they could be computed as well. Lines 5 and 6 create a nested for loop. The outer for loop ticks through each member of the first dimension. For every member in that dimension, the inner for loop ticks through each member of the second dimension. This is consistent with the printout. SomeArray[0][0] is followed by SomeArray[0][1]. The first dimension is incremented only after the second dimension is incremented by 1. Then the second dimension starts over.

Now look at an example that demonstrates the advantages of arrays. creates a 1,00-element, two-dimensional array and fills it with random numbers. The program then displays the array elements on-screen. Imagine how many lines of source code you would need to perform the same task with non array variables.
You see a new library function, getch(), in this program. The getch() function reads a single character from the keyboard. getch() pauses the program until the user presses a key.

Program RANDOM..CPP creates a multidimensional array.

```
1:  /* RANDOM.Cpp - Demonstrates using a multidimensional array */
2:
3:  #include <stdio.h>
```

```
4:  #include <stdlib.h>
5:  /* Declare a two-dimensional array with 1000 elements */
6:
7:  int  random_array[10][10];
8:  int a, b;
9:
10: main()
11: {
12:     /* Fill the array with random numbers. The C++ library */
13:     /* function rand() returns a random number. Use one */
14:     /* for loop for each array subscript. */
15:
16:     for (a = 0; a < 10; a++)
17:     {
18:        for (b = 0; b < 10; b++)
19:        {
20:
21:
22:              random_array[a][b] = rand();
23:
24:        }
25:     }
26:
27:     /* Now display the array elements 10 at a time */
28:
29:     for (a = 0; a < 10; a++)
30:     {
31:        for (b = 0; b < 10; b++)
32:        {
33:
34:
35:
36:              cout<<random_array[a][b];
37:           }
38:           cout<<"\nPress Enter to continue";
39:
40:           getchar();
41:        }
42:
43:     return 0;
44: }       /* end of main() */
```

```
random_array[0][0] = 346
random_array[0][1] = 130
random_array[0][2] = 10982
random_array[0][3] = 1090
random_array[0][4] = 11656
random_array[0][5] = 7117
random_array[0][6] = 17595
random_array[0][7] = 6415
random_array[0][8] = 22948
random_array[0][9] = 31126
Press Enter to continue
random_array[1][0] = 9004
```

```
random_array[1][1] = 14558
random_array[1][2] = 3571
random_array[1][3] = 22879
random_array[1][4] = 18492
random_array[1][5] = 1360
random_array[1][6] = 5412
random_array[1][7] = 26721
random_array[1][8] = 22463
random_array[1][9] = 25047
Press Enter to continue
...     ...
random_array[9][0] = 6287
random_array[9][1] = 26957
random_array[9][2] = 1530
random_array[9][3] = 14171
random_array[9][4] = 6951
random_array[9][5] = 213
random_array[9][6] = 14003
random_array[9][7] = 29736
random_array[9][8] = 15028
random_array[9][9] = 18968
```

ANALYSIS: this program has two nested for loops. Before you look at the for statements in detail, note that lines 7 and 8 declare four variables. The first is an array named random_array, used to hold random numbers. random_array is a two dimensional type int array that is 10-by-10, giving a total of 1,00 type int elements (10 * 10). Imagine coming up with 1,00 unique variable names if you couldn't use arrays! Line 8 then declares three variables, a and b, used to control the for loops.

This program also includes the header file STDLIB.H (for standard library) on line 4. It is included to provide the prototype for the rand() function used on line 22.

The bulk of the program is contained in two nests of for statements. The first is in lines 16 through 25, and the second is in lines 29 through 42. Both for nests have the same structure. They work just like the loops , but they go one level deeper. In the first set of for statements, line 22 is executed repeatedly. Line 22 assigns the return value of a function, rand(), to an element of the random_array array, where rand() is a library function that returns a random number.

Going backward through the listing, Line 18 loops through b, the middle subscript of the random array. Each time b changes. Line 16 increments variable a, which loops through the farthest left subscript. Each time this subscript changes, it loops through all 10 values of subscript b. This loop initializes every value in the random array to a random number.

Lines 29 through 42 contain the second nest of for statements. These work like the previous for statements, but this loop prints each of the values assigned previously. After 10 are displayed, line 38 prints a message and waits for Enter to be pressed. Line 40 takes care of the keypress using getchar(). If Enter hasn't been pressed, getchar() waits until it is. Run this program and watch the displayed values.

Char Arrays

A string is a series of characters. The only strings you've seen until now have been unnamed string constants used in cout statements, such as: cout << "hello world.\n";

In C++ a string is an array of chars ending with a null character. You can declare and initialize a string just as you would any other array. For example,

```
Char  Greeting[ ] ={ `H', `e', `l', `l', `o', ` `, `W','o','r','l','d', `\0' };
```

The last character, `\0', is the null character, which many C++ functions recognize as the terminator for a string. Although this character-by-character approach works, it is difficult to type and admits too many opportunities for error. C++ enables you to use a shorthand form of the previous line of code. It is char Greeting[ ] = "Hello World";

You should note two things about this syntax:

- **Instead of single quoted characters separated by commas and surrounded by braces, you have a double-quoted string, no commas, and no braces.**
- **You don't need to add the null character because the compiler adds it for you.**

The string Hello World is 12 bytes. Hello is 5 bytes, the space 1, World 5, and the null character 1.

You can also create un initialized character arrays. As with all arrays, it is important to ensure that you don't put more into the buffer than there is room for.

A program demonstrates the use of an un initialized buffer.

Listing Filling an array.

```
1:    //program Listing char array buffers
2:
3:    #include <iostream.h>
4:
5:    int main()
6:    {
7:       char buffer[80];
8:       cout << "Enter the string: ";
9:       cin >> buffer;
10:       cout << "Here's the buffer:  " << buffer << endl;
11:     return 0;
12: }
```

Output:

```
Enter the string: Hello World
Here's the buffer:  Hello
```

Analysis: On line 7, a buffer is declared to hold 80 characters. This is large enough to hold a 79-character string and a terminating null character. On line 8, the user is prompted to enter a string, which is entered into buffer on line 9. It is the syntax of cin to write a terminating null to buffer after it writes the string.

There are two problems with the program . First, if the user enters more than 79 characters, cin writes past the end of the buffer. Second, if the user enters a space, cin thinks that it is the end of the string, and it stops writing to the buffer.

To solve these problems, you must call a special method on cin: get(). cin.get() takes three parameters:The buffer to fill The maximum number of characters to get The delimiter that terminates input. The default delimiter is newline.

program Filling an array.

```
1:    // program Listing using cin.get()
2:
3:    #include <iostream.h>
4:
5:    int main()
6:    {
7:       char buffer[80];
8:       cout << "Enter the string: ";
9:       cin.get(buffer, 79);       // get up to 79 or newline
10:       cout << "Here's the buffer:  " << buffer << endl;
11:     return 0;
12: }
```

Output:

```
Enter the string: Hello World
Here's the buffer:  Hello World
```

Analysis: Line 9 calls the method get() of cin. The buffer declared in line 7 is passed in as the first argument. The second argument is the maximum number of characters to get. In this case, it must be 79 to allow for the terminating null. There is no need to provide a terminating character because the default value of newline is sufficient.

Copying Strings

strcpy() and strncpy()

C++ inherits from C a library of functions for dealing with strings. Among the many functions provided are two for copying one string into another: strcpy() and strncpy(). strcpy() copies the entire contents of one string into a designated buffer.

program Using strcpy().

```
1:    #include <iostream.h>
2:    #include <string.h>
3:    int main()
4:    {
5:       char String1[ ] = "No man is an island";
6:       char String2[80];
7:
8:       strcpy(String2,String1);
9:
10:       cout << "String1: " << String1 << endl;
11:       cout << "String2: " << String2 << endl;
12:     return 0;
13: }
```

Output:
String1: No man is an island
String2: No man is an island

Analysis: The header file string.h is included in line 2. This file contains the prototype of the strcpy() function. strcpy() takes two character arrays--a destination followed by a source. If the source were larger than the destination, strcpy() would overwrite past the end of the buffer.
To protect against this, the standard library also includes strncpy(). This variation takes a maximum number of characters to copy. strncpy() copies up to the first null character or the maximum number of characters specified into the destination buffer.

program Using strncpy().

```
1:    #include <iostream.h>
2:    #include <string.h>
3:    int main()
4:    {
5:       const int  MaxLength = 80;
6:       char String1[ ] = "No man is an island";
7:       char String2[MaxLength+1];
8:
9:
10:       strncpy(String2,String1,MaxLength);
11:
12:       cout << "String1: " << String1 << endl;
13:       cout << "String2: " << String2 << endl;
14:     return 0;
15: }
```

Output:
String1: No man is an island
String2: No man is an island

Analysis: In line 10, the call to strcpy() has been changed to a call to strncpy(), which takes a third parameter: the maximum number of characters to copy. The buffer String2 is declared to take MaxLength+1 characters. The extra character is for the null, which both strcpy() and strncpy() automatically add to the end of the string.

Concatenating Strings

If you're not familiar with the term *concatenation*, you might be asking, "What is it?" and "Is it legal?" Well, it means to join two strings--to tack one string onto the end of another--and, in most states, it is legal. The C++ standard library contains two string concatenation functions--strcat() and strncat()--both of which require the header file STRING.H.

The strcat() Function

The prototype of strcat() is

```
char *strcat (char *str1, char *str2 );
```

The function appends a copy of str2 onto the end of str1, moving the terminating null character to the end of the new string. You must allocate enough space for str1 to hold the resulting string. The return value of strcat() is a pointer to str1.

Listing Using strcat() to concatenate strings.

```
1:  /* The strcat() function. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char str1[27] = "a";
7:  char str2[2];
8:
9:  main()
10: {
11:     int n;
12:
13:     /* Put a null character at the end of str2[]. */
14:
15:     str2[1] = `\0';
16:
17:     for (n = 98; n< 123; n++)
18:     {
19:         str2[0] = n;
20:         strcat(str1, str2);
21:         puts(str1);
22:     }
23:      return(0);
24: }
```

Output:

```
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
```

```
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

ANALYSIS: The ASCII codes for the letters b through z are 98 to 122. This program uses these ASCII codes in its demonstration of strcat(). The for loop on lines 17 through 22 assigns these values in turn to str2[0]. Because str2[1] is already the null character (line 15), the effect is to assign the strings "b", "c", and so on to str2. Each of these strings is concatenated with str1 (line 20), and then str1 is displayed on-screen (line 21).

The strncat() Function

The library function strncat() also performs string concatenation, but it lets you specify how many characters of the source string are appended to the end of the destination string. The prototype is

```
char *strncat(char *str1, char *str2, size_t n);
```

If str2 contains more than n characters, the first n characters are appended to the end of str1. If str2 contains fewer than n characters, all of str2 is appended to the end of str1. In either case, a terminating null character is added at the end of the resulting string. You must allocate enough space for str1 to hold the resulting string. The function returns a pointer to str1.

Listing Using the strncat() function to concatenate strings.

```
1:  /* The strncat() function. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
```

```
5:
6:  char str2[ ] = "abcdefghijklmnopqrstuvwxyz";
7:
8:  main()
9:  {
10:     char str1[27];
11:     int n;
12:
13:     for (n=1; n< 27; n++)
14:     {
15:        strcpy(str1, "");
16:        strncat(str1, str2, n);
17:        puts(str1);
18:     }
19: }
```

Output:

```
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

ANALYSIS: You might wonder about the purpose of line 15, strcpy(str1, "");. This line copies to str1 an empty string consisting of only a single null character. The result is that the first character in str1--str1[0]--is set equal to 0 (the null character). The same thing could have been accomplished with the statements str1[0] = 0; or str1[0] = `\0';.

Comparing Strings

Strings are compared to determine whether they are equal or unequal. If they are unequal, one string is "greater than" or "less than" the other. Determinations of "greater" and

"less" are made with the ASCII codes of the characters. In the case of letters, this is equivalent to alphabetical order, with the one seemingly strange exception that all uppercase letters are "less than" the lowercase letters. This is true because the uppercase letters have ASCII codes 65 through 90 for A through Z, while lowercase a through z are represented by 97 through 122. Thus, "ZEBRA" would be considered to be less than "apple" by these C functions.

The ANSI C library contains functions for two types of string comparisons: comparing two entire strings, and comparing a certain number of characters in two strings.

Comparing Two Entire Strings

The function strcmp() compares two strings character by character. Its prototype is
int strcmp(char *str1, char *str2);
The arguments str1 and str2 are pointers to the strings being compared. The function's return values are given in the Table that demonstrates strcmp().

Table of The values returned by strcmp().

Return Value	Meaning
< 0	str1 is less than str2.
0	str1 is equal to str2.
> 0	str1 is greater than str2.

Listing Using strcmp() to compare strings.

```
1:  /* The strcmp() function. */
2:   #include<iostream.h>
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char str1[80], str2[80];
9:      int x;
10:
11:     while (1)
12:     {
13:
14:         /* Input two strings. */
15:
16:         cout<<"\n\n Input the first string, a blank to exit: ";
17:         gets(str1);
18:
19:         if ( strlen(str1) == 0 )
20:             break;
21:
22:         cout<<"\n Input the second string: ";
23:         gets(str2);
```

```
24:
25:       /* Compare them and display the result. */
26:
27:       x = strcmp(str1, str2);
28:
29:       cout<<"\n strcmp"<<str1<<"    "<<str2<<" returns:"<<x;
30:    }
31:    return(0);
32: }
```

```
Input the first string, a blank to exit: First string
Input the second string: Second string
Strcmp First string  Second string   returns  -1
Input the first string, a blank to exit: test string
Input the second string: test string
Strcmp test string    test string   returns 0
Input the first string, a blank to exit: zebra
Input the second string: aardvark
Strcmp  zebra  aardvark  returns 1
Input the first string, a blank to exit:
```

ANALYSIS: This program demonstrates strcmp(), prompting the user for two strings (lines 16, 17, 22, and 23) and displaying the result returned by strcmp() on line 29. Experiment with this program to get a feel for how strcmp() compares strings. Try entering two strings that are identical except for case, such as Smith and SMITH. You'll see that strcmp() is case-sensitive, meaning that the program considers uppercase and lowercase letters to be different.

Comparing Partial Strings

The library function strncmp() compares a specified number of characters of one string to another string. Its prototype is

```
int strncmp(char *str1, char *str2, size_t n);
```

The function strncmp() compares n characters of str2 to str1. The comparison proceeds until n characters have been compared or the end of str1 has been reached. The method of comparison and return values are the same as for strcmp(). The comparison is case-sensitive.

Listing Comparing parts of strings with strncmp().

```
1:  /* The strncmp() function. */
2:  #include<iostream.h>
3:  #include <stdio.h>
4:  #include<string.h>
5:
6:  char str1[] = "The first string.";
7:  char str2[] = "The second string.";
8:
9:  main()
10: {
11:    int   n, x;
12:
13:    puts(str1);
14:    puts(str2);
15:
16:    while (1)
17:    {
18:       puts("\n\nEnter number of characters to compare, 0 to exit.");
19:       cin>>n;
20:
21:       if (n <= 0)
22:          break;
23:
24:       x = strncmp(str1, str2, n);
25:
26:       cout<<"\nComparing “<<n<<” characters, strncmp() returns “<<x;
27:    }
28:    return(0);
29: }
```

Output:

```
The first string.
The second string.
Enter number of characters to compare, 0 to exit.
3
Comparing 3 characters, strncmp() returns 0
Enter number of characters to compare, 0 to exit.
6
Comparing 6 characters, strncmp() returns -1.
Enter number of characters to compare, 0 to exit.
0
```

ANALYSIS: This program compares two strings defined on lines 6 and 7. Lines 13 and 14 print the strings to the screen so that the user can see what they are. The program executes a while loop on lines 16 through 27 so that multiple comparisons can be done. If the user asks to compare zero characters on lines 18 and 19, the program breaks on line 22; otherwise, a strncmp() executes on line 24, and the result is printed on line 26.

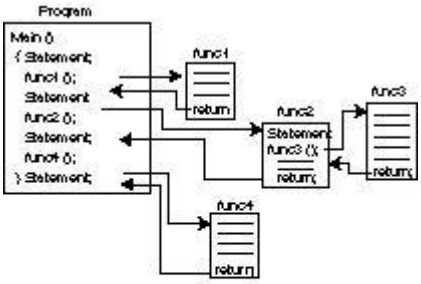
Functions

- **What a function is and what its parts are.**
- **How to declare and define functions.**
- **How to pass parameters into functions.**
- **How to return a value from a function.**

What Is a Function?

A function is, in effect, a subprogram that can act on data and return a value. Every C++ program has at least one function, `main()`. When your program starts, `main()` is called automatically. `main()` might call other functions, some of which might call still others.

Each function has its own name, and when that name is encountered, the execution of the program branches to the body of that function. When the function returns, execution resumes on the next line of the calling function. This flow is illustrated in the Figure below .



05ZCP01

Figure ***Illustration of flow***

When a program calls a function, execution switches to the function and then resumes at the line after the function call. Well-designed functions perform a specific and easily understood task. Complicated tasks should be broken down into multiple functions, and then each can be called in turn.

Functions come in two varieties: user-defined and built-in. Built-in functions are part of your compiler package--they are supplied by the manufacturer for your use.

Declaring and Defining Functions

Using functions in your program requires that you first declare the function and that you then define the function. The declaration tells the compiler the name, return type, and parameters of the function. The definition tells the compiler how the function works. No function can be called from any other function that hasn't first been declared. The declaration of a function is called its prototype.

Declaring the Function

There are three ways to declare a function:

- **Write your prototype into a file, and then use the `#include` directive to include it in your program.**
- **Write the prototype into the file in which your function is used.**
- **Define the function before it is called by any other function. When you do this, the definition acts as its own declaration.**

Although you can define the function before using it, and thus avoid the necessity of creating a function prototype, this is not good programming practice for three reasons.

First, it is a bad idea to require that functions appear in a file in a particular order. Doing so makes it hard to maintain the program as requirements change.

Second, it is possible that function `A()` needs to be able to call function `B()`, but function `B()` also needs to be able to call function `A()` under some circumstances. It is not possible to define function `A()` before you define function `B()` and also to define function `B()` before you define function `A()`, so at least one of them must be declared in any case.

Third, function prototypes are a good and powerful debugging technique. If your prototype declares that your function takes a particular set of parameters, or that it returns a particular type of value, and then your function does not match the prototype, the compiler can flag your error instead of waiting for it to show itself when you run the program.

Function Prototypes

Many of the built-in functions you use will have their function prototypes already written in the files you include in your program by using `#include`. For functions you write yourself, you must include the prototype.

The function prototype is a statement, which means it ends with a semicolon. It consists of the function's return type, name, and parameter list.

The parameter list is a list of all the parameters and their types, separated by commas.

The function prototype and the function definition must agree exactly about the return type, the name, and the parameter list. If they do not agree, you will get a compile-time error. Note, however, that the function prototype does not need to contain the names of the parameters, just their types. A prototype that looks like this is perfectly legal:

```
long Area(int, int);
```

This prototype declares a function named `Area()` that returns a `long` and that has two parameters, both integers. Although this is legal, it is not a good idea. Adding parameter names makes your prototype clearer. The same function with named parameters might be

```
long Area(int length, int width);
```

It is now obvious what this function does and what the parameters are.

Note that all functions have a return type. If none is explicitly stated, the return type defaults to `int`. Your programs will be easier to understand, however, if you explicitly declare the return type of every function, including `main()`. Listing demonstrates a program that includes a function prototype for the `Area()` function.

Listing A function declaration and the definition and use of that function.

```
1:   // Listing - demonstrates the use of function prototypes
2:
3:   typedef unsigned short USHORT;
4:   #include <iostream.h>
5:   USHORT FindArea(USHORT length, USHORT width); //function prototype
6:
7:   int main()
8:   {
9:     USHORT lengthOfYard;
10:    USHORT widthOfYard;
11:    USHORT areaOfYard;
12:
13:    cout << "\nHow wide is your yard? ";
14:    cin >> widthOfYard;
15:    cout << "\nHow long is your yard? ";
16:    cin >> lengthOfYard;
17:
18:    areaOfYard= FindArea(lengthOfYard,widthOfYard);
19:
20:    cout << "\nYour yard is ";
21:    cout << areaOfYard;
22:    cout << " square feet\n\n";
23:             return 0;
24:  }
25:
26:  USHORT FindArea(USHORT l, USHORT w)
27:  {
28:       return l * w;
29: }
Output: How wide is your yard? 100
```

```
How long is your yard? 200

Your yard is 20000 square feet
```

Analysis: The prototype for the `FindArea()` function is on line 5. Compare the prototype with the definition of the function on line 26. Note that the name, the return type, and the parameter types are the same. If they were different, a compiler error would have been generated. In fact, the only required difference is that the function prototype ends with a semicolon and has no body.
Also note that the parameter names in the prototype are `length` and `width`, but the parameter names in the definition are `l` and `w`. As discussed, the names in the prototype are not used; they are there as information to the programmer. When they are included, they should match the implementation when possible. This is a matter of good programming style and reduces confusion, but it is not required, as you see here.

The arguments are passed in to the function in the order in which they are declared and defined, but there is no matching of the names. Had you passed in `widthOfYard`, followed by `lengthOfYard`, the `FindArea()` function would have used the value in `widthOfYard` for `length` and `lengthOfYard` for `width`. The body of the function is always enclosed in braces, even when it consists of only one statement, as in this case.

Defining the Function

The definition of a function consists of the function header and its body. The header is exactly like the function prototype, except that the parameters must be named, and there is no terminating semicolon.

The body of the function is a set of statements enclosed in braces.

Functions

Function Prototype Syntax

```
return_type function_name ( [type [parameterName]]...);
```

Function Definition Syntax

```
return_type function_name ( [type parameterName]...)
{
    statements;
}
```

A function prototype tells the compiler the return type, name, and parameter list. Functions are not required to have parameters, and if they do, the prototype is not required to list their names, only their types. A prototype always ends with a semicolon (;). A function definition must agree in return type and parameter list with its prototype. It must provide names for all the parameters, and the body of the function definition must be surrounded

by braces. All statements within the body of the function must be terminated with semicolons, but the function itself is not ended with a semicolon; it ends with a closing brace. If the function returns a value, it should end with a `return` statement, although `return` statements can legally appear anywhere in the body of the function. Every function has a return type. If one is not explicitly designated, the return type will be `int`. Be sure to give every function an explicit return type. If a function does not return a value, its return type will be `void`.

Function Prototype Examples

```
long FindArea(long length, long width); // returns long, has two parameters
void PrintMessage(int messageNumber); // returns void, has one parameter
int GetChoice();                      // returns int, has no parameters

BadFunction();                        // returns int, has no parameters
```

Function Definition Examples

```
long Area(long l, long w)
{
     return l * w;
}

void PrintMessage(int whichMsg)
{
     if (whichMsg == 0)
          cout << "Hello.\n";
     if (whichMsg == 1)
          cout << "Goodbye.\n";
     if (whichMsg > 1)
          cout << "I'm confused.\n";
}
```

Execution of Functions

When you call a function, execution begins with the first statement after the opening brace (`{`). Branching can be accomplished by using the `if` statement. Functions can also call other functions and can even call themselves.

Local Variables

Not only can you pass in variables to the function, but you also can declare variables within the body of the function. This is done using local variables, so named because they exist only locally within the function itself. When the function returns, the local variables are no longer available.

Local variables are defined like any other variables. The parameters passed in to the function are also considered local variables and can be used exactly as if they had been defined within the body of the function. Listing is an example of using parameters and locally defined variables within a function.

Listing The use of local variables and parameters.

```
1:     #include <iostream.h>
2:
3:     float Convert(float);
4:     int main()
5:     {
6:        float TempFer;
7:        float TempCel;
8:
9:        cout << "Please enter the temperature in Fahrenheit: ";
10:       cin >> TempFer;
11:       TempCel = Convert(TempFer);
12:       cout << "\nHere's the temperature in Celsius: ";
13:       cout << TempCel << endl;
14:               return 0;
15:    }
16:
17:    float Convert(float TempFer)
18:    {
19:       float TempCel;
20:       TempCel = ((TempFer - 32) * 5) / 9;
21:       return TempCel;
22: }

Output: Please enter the temperature in Fahrenheit: 212

Here's the temperature in Celsius: 100

Please enter the temperature in Fahrenheit: 32

Here's the temperature in Celsius: 0

Please enter the temperature in Fahrenheit: 85

Here's the temperature in Celsius: 29.4444
```

Analysis: On lines 6 and 7, two `float` variables are declared, one to hold the temperature in Fahrenheit and one to hold the temperature in degrees Celsius. The user is prompted to enter a Fahrenheit temperature on line 9, and that value is passed to the function `Convert()`.

Execution jumps to the first line of the function `Convert()` on line 19, where a local variable, also named `TempCel`, is declared. Note that this local variable is not the same as the variable `TempCel` on line 7. This variable exists only within the function `Convert()`. The value passed as a parameter, `TempFer`, is also just a local copy of the variable passed in by `main()`.

This function could have named the parameter `FerTemp` and the local variable `CelTemp`, and the program would work equally well. You can enter these names again and recompile the program to see this work.

The local function variable `TempCel` is assigned the value that results from subtracting 32 from the parameter `TempFer`, multiplying by 5, and then dividing by 9. This value is then returned as the return value of the function, and on line 11 it is assigned to the variable `TempCel` in the `main()` function. The value is printed on line 13.
The program is run three times. The first time, the value `212` is passed in to ensure that the boiling point of water in degrees Fahrenheit (212) generates the correct answer in degrees Celsius (100). The second test is the freezing point of water. The third test is a random number chosen to generate a fractional result.

As an exercise, try entering the program again with other variable names as illustrated here:

```
1:     #include <iostream.h>
2:
3:     float Convert(float);
4:     int main()
5:     {
6:        float TempFer;
7:        float TempCel;
8:
9:        cout << "Please enter the temperature in Fahrenheit: ";
10:       cin >> TempFer;
11:       TempCel = Convert(TempFer);
12:       cout << "\nHere's the temperature in Celsius: ";
13:       cout << TempCel << endl;
14:    }
15:
16:    float Convert(float Fer)
17:    {
18:       float Cel;
19:       Cel = ((Fer - 32) * 5) / 9;
20:       return Cel;
21:    }
```

You should get the same results.

New Term: A variable has scope, which determines how long it is available to your program and where it can be accessed. Variables declared within a block are scoped to that block; they can be accessed only within that block and "go out of existence" when that block ends. Global variables have global scope and are available anywhere within your program.

Normally scope is obvious, but there are some tricky exceptions. Currently, variables declared within the header of a `for` loop `(for int i = 0; i<SomeValue; i++)` are scoped

to the block in which the `for` loop is created, but there is talk of changing this in the official C++ standard.

None of this matters very much if you are careful not to reuse your variable names within any given function.

Global Variables

Variables defined outside of any function have global scope and thus are available from any function in the program, including `main()`.

Local variables with the same name as global variables do not change the global variables. A local variable with the same name as a global variable hides the global variable, however. If a function has a variable with the same name as a global variable, the name refers to the local variable--not the global--when used within the function. Listing illustrates these points.

Listing Demonstrating global and local variables.

```
1:   #include <iostream.h>
2:   void myFunction();           // prototype
3:
4:   int x = 5, y = 7;            // global variables
5:   int main()
6:   {
7:
8:        cout << "x from main: " << x << "\n";
9:        cout << "y from main: " << y << "\n\n";
10:       myFunction();
11:       cout << "Back from myFunction!\n\n";
12:       cout << "x from main: " << x << "\n";
13:       cout << "y from main: " << y << "\n";
14:       return 0;
15:  }
16:
17:  void myFunction()
18:  {
19:       int y = 10;
20:
21:       cout << "x from myFunction: " << x << "\n";
22:       cout << "y from myFunction: " << y << "\n\n";
23: }

Output: x from main: 5
y from main: 7

x from myFunction: 5
y from myFunction: 10

Back from myFunction!

x from main: 5
```

```
y from main: 7
```

Analysis: This simple program illustrates a few key, and potentially confusing, points about local and global variables. On line 1, two global variables, `x` and `y`, are declared. The global variable `x` is initialized with the value `5`, and the global variable `y` is initialized with the value `7`.
On lines 8 and 9 in the function `main()`, these values are printed to the screen. Note that the function `main()` defines neither variable; because they are global, they are already available to `main()`.

When `myFunction()` is called on line 10, program execution passes to line 18, and a local variable, `y`, is defined and initialized with the value `10`. On line 21, `myFunction()` prints the value of the variable `x`, and the global variable `x` is used, just as it was in `main()`. On line 22, however, when the variable name `y` is used, the local variable `y` is used, hiding the global variable with the same name.

The function call ends, and control returns to `main()`, which again prints the values in the global variables. Note that the global variable `y` was totally unaffected by the value assigned to `myFunction()`'s local `y` variable.

More on Local Variables

Variables declared within the function are said to have "local scope." That means, as discussed, that they are visible and usable only within the function in which they are defined. In fact, in C++ you can define variables anywhere within the function, not just at its top. The scope of the variable is the block in which it is defined. Thus, if you define a variable inside a set of braces within the function, that variable is available only within that block. Listing illustrates this idea.

Listing Variables scoped within a block.

```
1:    // Listing - demonstrates variables
2:    // scoped within a block
3:
4:    #include <iostream.h>
5:
6:    void myFunc();
7:
8:    int main()
9:    {
10:      int x = 5;
11:      cout << "\nIn main x is: " << x;
12:
13:      myFunc();
14:
15:      cout << "\nBack in main, x is: " << x;
16:        return 0;
```

```
17:    }
18:
19:    void myFunc()
20:    {
21:
22:       int x = 8;
23:       cout << "\nIn myFunc, local x: " << x << endl;
24:
25:       {
26:          cout << "\nIn block in myFunc, x is: " << x;
27:
28:          int x = 9;
29:
30:          cout << "\nVery local x: " << x;
31:       }
32:
33:       cout << "\nOut of block, in myFunc, x: " << x << endl;
34: }

Output: In main x is: 5
In myFunc, local x: 8

In block in myFunc, x is: 8
Very local x: 9
Out of block, in myFunc, x: 8

Back in main, x is: 5
```

Analysis: This program begins with the initialization of a local variable, `x`, on line 10, in `main()`. The printout on line 11 verifies that `x` was initialized with the value `5`. `MyFunc()` is called, and a local variable, also named `x`, is initialized with the value `8` on line 22. Its value is printed on line 23.

A block is started on line 25, and the variable `x` from the function is printed again on line 26. A new variable also named `x`, but local to the block, is created on line 28 and initialized with the value `9`.

The value of the newest variable `x` is printed on line 30. The local block ends on line 31, and the variable created on line 28 goes "out of scope" and is no longer visible.

When `x` is printed on line 33, it is the `x` that was declared on line 22. This `x` was unaffected by the `x` that was defined on line 28; its value is still `8`.

On line 34, `MyFunc()` goes out of scope, and its local variable `x` becomes unavailable. Execution returns to line 15, and the value of the local variable `x`, which was created on line 10, is printed. It was unaffected by either of the variables defined in `MyFunc()`.

Needless to say, this program would be far less confusing if these three variables were given unique names!

Function Statements

There is virtually no limit to the number or types of statements that can be in a function body. Although you can't define another function from within a function, you can call a function, and of course `main()` does just that in nearly every C++ program. Functions can even call themselves, which is discussed soon, in the section on recursion.

Although there is no limit to the size of a function in C++, well-designed functions tend to be small. Many programmers advise keeping your functions short enough to fit on a single screen so that you can see the entire function at one time. This is a rule of thumb, often broken by very good programmers, but a smaller function is easier to understand and maintain.

Each function should carry out a single, easily understood task. If your functions start getting large, look for places where you can divide them into component tasks.

Function Arguments

Function arguments do not have to all be of the same type. It is perfectly reasonable to write a function that takes an integer, two `long`s, and a character as its arguments.

Any valid C++ expression can be a function argument, including constants, mathematical and logical expressions, and other functions that return a value.

Using Functions as Parameters to Functions

Although it is legal for one function to take as a parameter a second function that returns a value, it can make for code that is hard to read and hard to debug.

As an example, say you have the functions `double()`, `triple()`, `square()`, and `cube()`, each of which returns a value. You could write

```
Answer = (double(triple(square(cube(myValue)))));
```

This statement takes a variable, `myValue`, and passes it as an argument to the function `cube()`, whose return value is passed as an argument to the function `square()`, whose return value is in turn passed to `triple()`, and that return value is passed to `double()`. The return value of this doubled, tripled, squared, and cubed number is now passed to `Answer`.

It is difficult to be certain what this code does (was the value tripled before or after it was squared?), and if the answer is wrong it will be hard to figure out which function failed.

An alternative is to assign each step to its own intermediate variable:

```
unsigned long myValue = 2;
unsigned long cubed  =   cube(myValue);         // cubed = 8
unsigned long squared = square(cubed);          // squared = 64
unsigned long tripled = triple(squared);        // tripled = 196
unsigned long Answer =  double(tripled);        // Answer = 392
```

Now each intermediate result can be examined, and the order of execution is explicit.

Parameters Are Local Variables

The arguments passed in to the function are local to the function. Changes made to the arguments do not affect the values in the calling function. This is known as passing by value, which means a local copy of each argument is made in the function. These local copies are treated just like any other local variables. Listing illustrates this point.

Listing A demonstration of passing by value.

```
1:     // Listing - demonstrates passing by value
2:
3:      #include <iostream.h>
4:
5:      void swap(int x, int y);
6:
7:      int main()
8:      {
9:        int x = 5, y = 10;
10:
11:        cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
12:        swap(x,y);
13:        cout << "Main. After swap, x: " << x << " y: " << y << "\n";
14:           return 0;
15:      }
16:
17:      void swap (int x, int y)
18:      {
19:        int temp;
20:
21:        cout << "Swap. Before swap, x: " << x << " y: " << y << "\n";
22:
23:        temp = x;
24:        x = y;
25:        y = temp;
26:
27:        cout << "Swap. After swap, x: " << x << " y: " << y << "\n";
28:
29: }

Output: Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

Analysis: This program initializes two variables in `main()` and then passes them to the `swap()` function, which appears to swap them. When they are examined again in `main()`, however, they are unchanged!
The variables are initialized on line 9, and their values are displayed on line 11. `swap()` is called, and the variables are passed in.

Execution of the program switches to the `swap()` function, where on line 21 the values are printed again. They are in the same order as they were in `main()`, as expected. On lines 23 to 25 the values are swapped, and this action is confirmed by the printout on line 27. Indeed, while in the `swap()` function, the values are swapped.

Execution then returns to line 13, back in `main()`, where the values are no longer swapped.

As you've figured out, the values passed in to the `swap()` function are passed by value, meaning that copies of the values are made that are local to `swap()`. These local variables are swapped in lines 23 to 25, but the variables back in `main()` are unaffected.

On Days 8 and 10 you'll see alternatives to passing by value that will allow the values in `main()` to be changed.

Return Values

Functions return a value or return `void`. `Void` is a signal to the compiler that no value will be returned.

To return a value from a function, write the keyword `return` followed by the value you want to return. The value might itself be an expression that returns a value. For example:

```
return 5;
return (x > 5);
return (MyFunction());
```

These are all legal `return` statements, assuming that the function `MyFunction()` itself returns a value. The value in the second statement, `return (x > 5)`, will be zero if `x` is not greater than 5, or it will be `1`. What is returned is the value of the expression, `0` (`false`) or `1` (`true`), not the value of `x`.

When the `return` keyword is encountered, the expression following `return` is returned as the value of the function. Program execution returns immediately to the calling function, and any statements following the return are not executed.

It is legal to have more than one `return` statement in a single function. Listing illustrates this idea.

Listing A demonstration of multiple return statements.

```
1:     // Listing - demonstrates multiple return
2:     // statements
3:
4:     #include <iostream.h>
5:
6:     int Doubler(int AmountToDouble);
7:
8:     int main()
9:     {
10:
11:         int result = 0;
12:         int input;
13:
14:         cout << "Enter a number between 0 and 10,000 to double: ";
15:         cin >> input;
16:
17:         cout << "\nBefore doubler is called... ";
18:         cout << "\ninput: " << input << " doubled: " << result << "\n";
19:
20:         result = Doubler(input);
21:
22:         cout << "\nBack from Doubler...\n";
23:         cout << "\ninput: " << input << "   doubled: " << result << "\n";
24:
25:
26:         return 0;
27:    }
28:
29:    int Doubler(int original)
30:    {
31:         if (original <= 10000)
32:              return original * 2;
33:         else
34:              return -1;
35:         cout << "You can't get here!\n";
36: }
Output: Enter a number between 0 and 10,000 to double: 9000

Before doubler is called...
input: 9000 doubled: 0

Back from doubler...

input: 9000   doubled: 18000

Enter a number between 0 and 10,000 to double: 11000

Before doubler is called...
input: 11000 doubled: 0

Back from doubler...
input: 11000  doubled: -1
```

Analysis: A number is requested on lines 14 and 15, and printed on line 18, along with the local variable result. The function `Doubler()` is called on line 20, and the input value is passed as a parameter. The result will be assigned to the local variable `result`, and the values will be reprinted on lines 22 and 23.
On line 31, in the function `Doubler()`, the parameter is tested to see whether it is greater than 10,000. If it is not, the function returns twice the original number. If it is greater than 10,000, the function returns `-1` as an error value.

The statement on line 35 is never reached, because whether or not the value is greater than 10,000, the function returns before it gets to line 35, on either line 32 or line 34. A good compiler will warn that this statement cannot be executed, and a good programmer will take it out!

Default Parameters

For every parameter you declare in a function prototype and definition, the calling function must pass in a value. The value passed in must be of the declared type. Thus, if you have a function declared as

```
long myFunction(int);
```

the function must in fact take an integer variable. If the function definition differs, or if you fail to pass in an integer, you will get a compiler error.

The one exception to this rule is if the function prototype declares a default value for the parameter. A default value is a value to use if none is supplied. The preceding declaration could be rewritten as

```
long myFunction (int x = 50);
```

This prototype says, "`myFunction()` returns a `long` and takes an integer parameter. If an argument is not supplied, use the default value of `50`." Because parameter names are not required in function prototypes, this declaration could have been written as

```
long myFunction (int = 50);
```

The function definition is not changed by declaring a default parameter. The function definition header for this function would be

```
long myFunction (int x)
```

If the calling function did not include a parameter, the compiler would fill `x` with the default value of `50`. The name of the default parameter in the prototype need not be the same as the name in the function header; the default value is assigned by position, not name.

Any or all of the function's parameters can be assigned default values. The one restriction is this: If any of the parameters does not have a default value, no previous parameter may have a default value.

If the function prototype looks like

```
long myFunction (int Param1, int Param2, int Param3);
```

you can assign a default value to `Param2` only if you have assigned a default value to `Param3`. You can assign a default value to `Param1` only if you've assigned default values to both `Param2` and `Param3`. Listing demonstrates the use of default values.

Listing A demonstration of default parameter values.

```
1:   // Listing demonstrates use
2:   // of default parameter values
3:
4:   #include <iostream.h>
5:
6:   int AreaCube(int length, int width = 25, int height = 1);
7:
8:   int main()
9:   {
10:       int length = 100;
11:       int width = 50;
12:       int height = 2;
13:       int area;
14:
15:       area = AreaCube(length, width, height);
16:       cout << "First area equals: " << area << "\n";
17:
18:       area = AreaCube(length, width);
19:       cout << "Second time area equals: " << area << "\n";
20:
21:       area = AreaCube(length);
22:       cout << "Third time area equals: " << area << "\n";
23:        return 0;
24:   }
25:
26:   AreaCube(int length, int width, int height)
27:   {
28:
29:       return (length * width * height);
30: }

Output: First area equals: 10000
Second time area equals: 5000
Third time area equals: 2500
```

Analysis: On line 6, the `AreaCube()` prototype specifies that the `AreaCube()` function takes three integer parameters. The last two have default values.
This function computes the area of the cube whose dimensions are passed in. If no `width` is

passed in, a `width` of 25 is used and a `height` of 1 is used. If the `width` but not the `height` is passed in, a `height` of 1 is used. It is not possible to pass in the `height` without passing in a `width`.

On lines 10-12, the dimensions `length`, `height`, and `width` are initialized, and they are passed to the `AreaCube()` function on line 15. The values are computed, and the result is printed on line 16.

Execution returns to line 18, where `AreaCube()` is called again, but with no value for `height`. The default value is used, and again the dimensions are computed and printed.

Execution returns to line 21, and this time neither the `width` nor the `height` is passed in. Execution branches for a third time to line 27. The default values are used. The area is computed and then printed.

DO remember that function parameters act as local variables within the function. DON'T try to create a default value for a first parameter if there is no default value for the second. DON'T forget that arguments passed by value can not affect the variables in the calling function. DON'T forget that changes to a global variable in one function change that variable for all functions.

Overloading Functions

C++ enables you to create more than one function with the same name. This is called function overloading. The functions must differ in their parameter list, with a different type of parameter, a different number of parameters, or both. Here's an example:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

`myFunction()` is overloaded with three different parameter lists. The first and second versions differ in the types of the parameters, and the third differs in the number of parameters.

The return types can be the same or different on overloaded functions. You should note that two functions with the same name and parameter list, but different return types, generate a compiler error.

New Term: Function *overloading* i s also called function *polymorphism*. Poly means many, and morph means form: a polymorphic function is many-formed.

Function polymorphism refers to the ability to "overload" a function with more than one meaning. By changing the number or type of the parameters, you can give two or more functions the same function name, and the right one will be called by matching the parameters used. This allows you to create a function that can average integers, doubles, and other values without having to create individual names for each function, such as `AverageInts()`, `AverageDoubles()`, and so on.

Suppose you write a function that doubles whatever input you give it. You would like to be able to pass in an `int`, a `long`, a `float`, or a `double`. Without function overloading, you would have to create four function names:

```
int DoubleInt(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

With function overloading, you make this declaration:

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

This is easier to read and easier to use. You don't have to worry about which one to call; you just pass in a variable, and the right function is called automatically. Listing illustrates the use of function overloading.

Listing A demonstration of function polymorphism.

```
1:    // Listing - demonstrates
2:    // function polymorphism
3:
4:    #include <iostream.h>
5:
6:    int Double(int);
7:    long Double(long);
8:    float Double(float);
9:    double Double(double);
10:
11:   int main()
12:   {
13:       int      myInt = 6500;
14:       long     myLong = 65000;
15:       float    myFloat = 6.5F;
16:       double   myDouble = 6.5e20;
17:
18:       int      doubledInt;
19:       long     doubledLong;
20:       float    doubledFloat;
21:       double   doubledDouble;
22:
23:       cout << "myInt: " << myInt << "\n";
```

```
24:       cout << "myLong: " << myLong << "\n";
25:       cout << "myFloat: " << myFloat << "\n";
26:       cout << "myDouble: " << myDouble << "\n";
27:
28:       doubledInt = Double(myInt);
29:       doubledLong = Double(myLong);
30:       doubledFloat = Double(myFloat);
31:       doubledDouble = Double(myDouble);
32:
33:       cout << "doubledInt: " << doubledInt << "\n";
34:       cout << "doubledLong: " << doubledLong << "\n";
35:       cout << "doubledFloat: " << doubledFloat << "\n";
36:       cout << "doubledDouble: " << doubledDouble << "\n";
37:
38:        return 0;
39:   }
40:
41:  int Double(int original)
42:  {
43:    cout << "In Double(int)\n";
44:    return 2 * original;
45:  }
46:
47:  long Double(long original)
48:  {
49:    cout << "In Double(long)\n";
50:    return 2 * original;
51:  }
52:
53:  float Double(float original)
54:  {
55:    cout << "In Double(float)\n";
56:    return 2 * original;
57:  }
58:
59:  double Double(double original)
60:  {
61:    cout << "In Double(double)\n";
62:    return 2 * original;
63: }
Output: myInt: 6500
myLong: 65000
myFloat: 6.5
myDouble: 6.5e+20
In Double(int)
In Double(long)
In Double(float)
In Double(double)
DoubledInt: 13000
DoubledLong: 130000
DoubledFloat: 13
DoubledDouble: 1.3e+21
```

Analysis: The `Double()` function is overloaded with `int`, `long`, `float`, and `double`. The prototypes are on lines 6-9, and the definitions are on lines 41-63.
In the body of the main program, eight local variables are declared. On lines 13-16, four of

the values are initialized, and on lines 28-31, the other four are assigned the results of passing the first four to the `Double()` function. Note that when `Double()` is called, the calling function does not distinguish which one to call; it just passes in an argument, and the correct one is invoked.

The compiler examines the arguments and chooses which of the four `Double()` functions to call. The output reveals that each of the four was called in turn, as you would expect.

Inline Functions

When you define a function, normally the compiler creates just one set of instructions in memory. When you call the function, execution of the program jumps to those instructions, and when the function returns, execution jumps back to the next line in the calling function. If you call the function 10 times, your program jumps to the same set of instructions each time. This means there is only one copy of the function, not 10.

There is some performance overhead in jumping in and out of functions. It turns out that some functions are very small, just a line or two of code, and some efficiency can be gained if the program can avoid making these jumps just to execute one or two instructions. When programmers speak of efficiency, they usually mean speed: the program runs faster if the function call can be avoided.

If a function is declared with the keyword `inline`, the compiler does not create a real function: it copies the code from the inline function directly into the calling function. No jump is made; it is just as if you had written the statements of the function right into the calling function.

Note that inline functions can bring a heavy cost. If the function is called 10 times, the inline code is copied into the calling functions each of those 10 times. The tiny improvement in speed you might achieve is more than swamped by the increase in size of the executable program. Even the speed increase might be illusory. First, today's optimizing compilers do a terrific job on their own, and there is almost never a big gain from declaring a function `inline`. More important, the increased size brings its own performance cost.

What's the rule of thumb? If you have a small function, one or two statements, it is a candidate for `inline`. When in doubt, though, leave it out. Listing demonstrates an `inline` function.

Listing Demonstrates an inline function.

```
1:   // Listing - demonstrates inline functions
2:
3:   #include <iostream.h>
4:
5:   inline int Double(int);
6:
7:   int main()
```

```
8:   {
9:     int target;
10:
11:    cout << "Enter a number to work with: ";
12:    cin >> target;
13:    cout << "\n";
14:
15:    target = Double(target);
16:    cout << "Target: " << target << endl;
17:
18:    target = Double(target);
19:    cout << "Target: " << target << endl;
20:
21:
22:    target = Double(target);
23:    cout << "Target: " << target << endl;
24:       return 0;
25:  }
26:
27:  int Double(int target)
28:  {
29:    return 2*target;
30: }
Output: Enter a number to work with: 20

Target: 40
Target: 80
Target: 160
```

Analysis: On line 5, `Double()` is declared to be an inline function taking an `int` parameter and returning an `int`. The declaration is just like any other prototype except that the keyword `inline` is prepended just before the return value.

This compiles into code that is the same as if you had written the following:

```
target = 2 * target;
```

everywhere you entered

```
target = Double(target);
```

By the time your program executes, the instructions are already in place.

NOTE: Inline is a hint to the compiler that you would like the function to be inlined. The compiler is free to ignore the hint and make a real function call.

Recursion

A function can call itself. This is called recursion, and recursion can be direct or indirect. It is direct when a function calls itself; it is indirect recursion when a function calls another function that then calls the first function.

Some problems are most easily solved by recursion, usually those in which you act on data and then act in the same way on the result. Both types of recursion, direct and indirect, come in two varieties: those that eventually end and produce an answer, and those that never end and produce a runtime failure. Programmers think that the latter is quite funny (when it happens to someone else).

It is important to note that when a function calls itself, a new copy of that function is run. The local variables in the second version are independent of the local variables in the first, and they cannot affect one another directly, any more than the local variables in `main()` can affect the local variables in any function it calls, as was illustrated in Listing .

To illustrate solving a problem using recursion, consider the Fibonacci series:

```
1,1,2,3,5,8,13,21,34...
```

Each number, after the second, is the sum of the two numbers before it. A Fibonacci problem might be to determine what the 12th number in the series is.

One way to solve this problem is to examine the series carefully. The first two numbers are 1. Each subsequent number is the sum of the previous two numbers. Thus, the seventh number is the sum of the sixth and fifth numbers. More generally, the nth number is the sum of n - 2 and n - 1, as long as n > 2.

Recursive functions need a stop condition. Something must happen to cause the program to stop recursing, or it will never end. In the Fibonacci series, n < 3 is a stop condition.

The algorithm to use is this:

1. **Ask the user for a position in the series.**

2. **Call the `fib()` function with that position, passing in the value the user entered.**

3. **The `fib()` function examines the argument (`n`). If `n < 3` it returns 1; otherwise, `fib()` calls itself (recursively) passing in `n-2`, calls itself again passing in `n-1`, and returns the sum.**

If you call `fib(1)`, it returns `1`. If you call `fib(2)`, it returns `1`. If you call `fib(3)`, it returns the sum of calling `fib(2)` and `fib(1)`. Because `fib(2)` returns `1` and `fib(1)` returns `1`, `fib(3)` will return `2`.

If you call `fib(4)`, it returns the sum of calling `fib(3)` and `fib(2)`. We've established that `fib(3)` returns `2` (by calling `fib(2)` and `fib(1)`) and that `fib(2)` returns `1`, so `fib(4)` will sum these numbers and return `3`, which is the fourth number in the series.

Taking this one more step, if you call `fib(5)`, it will return the sum of `fib(4)` and `fib(3)`. We've established that `fib(4)` returns `3` and `fib(3)` returns `2`, so the sum returned will be `5`.

This method is not the most efficient way to solve this problem (in `fib(20)` the `fib()` function is called 13,529 times!), but it does work. Be careful: if you feed in too large a number, you'll run out of memory. Every time `fib()` is called, memory is set aside. When it returns, memory is freed. With recursion, memory continues to be set aside before it is freed, and this system can eat memory very quickly. Listing 5.10 implements the `fib()` function.

WARNING: When you run Listing , use a small number (less than 15). Because this uses recursion, it can consume a lot of memory.

Listing Demonstrates recursion using the Fibonacci series.

```
1:     // Listing - demonstrates recursion
2:     // Fibonacci find.
3:     // Finds the nth Fibonacci number
4:     // Uses this algorithm: Fib(n) = fib(n-1) + fib(n-2)
5:     // Stop conditions: n = 2 || n = 1
6:
7:     #include <iostream.h>
8:
9:     int fib(int n);
10:
11:    int main()
12:    {
13:
14:      int n, answer;
15:      cout << "Enter number to find: ";
16:      cin >> n;
17:
18:      cout << "\n\n";
19:
20:      answer = fib(n);
21:
22:      cout << answer << " is the " << n << "th Fibonacci number\n";
23:         return 0;
24:    }
25:
26:    int fib (int n)
27:    {
28:      cout << "Processing fib(" << n << ")... ";
29:
```

```
30:       if (n < 3 )
31:       {
32:          cout << "Return 1!\n";
33:          return (1);
34:       }
35:       else
36:       {
37:         cout << "Call fib(" << n-2 << ") and fib(" << n-1 << ").\n";
38:         return( fib(n-2) + fib(n-1));
39:       }
40: }

Output: Enter number to find:  5

Processing fib(5)... Call fib(3) and fib(4).
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
5 is the 5th Fibonacci number
```

Analysis: The program asks for a number to find on line 15 and assigns that number to `target`. It then calls `fib()` with the `target`. Execution branches to the `fib()` function, where, on line 28, it prints its argument.
The argument `n` is tested to see whether it equals `1` or `2` on line 30; if so, `fib()` returns. Otherwise, it returns the sums of the values returned by calling `fib()` on `n-2` and `n-1`.

In the example, `n` is 5 so `fib(5)` is called from `main()`. Execution jumps to the `fib()` function, and `n` is tested for a value less than 3 on line 30. The test fails, so `fib(5)` returns the sum of the values returned by `fib(3)` and `fib(4)`. That is, `fib()` is called on `n-2` (5 - 2 = 3) and `n-1` (5 - 1 = 4). `fib(4)` will return `3` and `fib(3)` will return `2`, so the final answer will be 5.

Because `fib(4)` passes in an argument that is not less than 3, `fib()` will be called again, this time with 3 and 2. `fib(3)` will in turn call `fib(2)` and `fib(1)`. Finally, the calls to `fib(2)` and `fib(1)` will both return `1`, because these are the stop conditions.

The output traces these calls and the return values. Compile, link, and run this program, entering first 1, then 2, then 3, building up to 6, and watch the output carefully. Then, just for fun, try the number 20. If you don't run out of memory, it makes quite a show!

Recursion is not used often in C++ programming, but it can be a powerful and elegant tool for certain needs.

NOTE: Recursion is a very tricky part of advanced programming. It is presented here because it can be very useful to understand the fundamentals of how it works, but don't worry too much if you don't fully understand all the details.

The Stack and Functions

Here's what happens when a program, running on a PC under DOS, branches to a function:

1. The address in the instruction pointer is incremented to the next instruction past the function call. That address is then placed on the stack, and it will be the return address when the function returns.

2. Room is made on the stack for the return type you've declared. On a system with two-byte integers, if the return type is declared to be `int`, another two bytes are added to the stack, but no value is placed in these bytes.

3. The address of the called function, which is kept in a special area of memory set aside for that purpose, is loaded into the instruction pointer, so the next instruction executed will be in the called function.

4. The current top of the stack is now noted and is held in a special pointer called the stack frame. Everything added to the stack from now until the function returns will be considered "local" to the function.

5. All the arguments to the function are placed on the stack.

6. The instruction now in the instruction pointer is executed, thus executing the first instruction in the function.

7. Local variables are pushed onto the stack as they are defined.

When the function is ready to return, the return value is placed in the area of the stack reserved at step 2. The stack is then popped all the way up to the stack frame pointer, which effectively throws away all the local variables and the arguments to the function.

The return value is popped off the stack and assigned as the value of the function call itself, and the address stashed away in step 1 is retrieved and put into the instruction pointer. The program thus resumes immediately after the function call, with the value of the function retrieved.

Some of the details of this process change from compiler to compiler, or between computers, but the essential ideas are consistent across environments. In general, when you call a function, the return address and the parameters are put on the stack. During the life of the function, local variables are added to the stack. When the function returns, these are all removed by popping the stack.

In coming days we'll look at other places in memory that are used to hold data that must persist beyond the life of the function.