Chapter 5
C++ Programming

Part 1 - Introduction

Required terminology and general information for this chapter:

Program – a series of instructions for a computer to execute.
Programming languages: Machine, Assembly, and High Level
How the first computers were programmed? By manually connecting wires.
Machine language: programs and data were entered into the computer using zeros and ones. All operation codes (**opcodes**) and **operands** were entered in binary numbers. Machine language is the machine "understandable" language. It does not matter what other languages are used to program, eventually all programs and data will have to be translated into a "machine executable" format.
Assembly Language: An easier programming language than machine language, where programs are written using familiar symbols (for example A for add). The operands could be in **binary**, **hexadecimal** or decimal format. For every machine language code, there needs to be one assembly language code. The program was not shorter, just easier to write. Once the program was written in assembly language, it had to be compiled to produce the machine executable code in order to run the program. The compiler used is called an **ASSEMBLER**.
High Level Language: Included are languages such as Fortran (Formula Translator), COBOL (Common Business Oriented Language), BASIC (Beginner's All purpose Symbolic Instruction Code), Pascal, C, Ada, and many more. Programs are much shorter than lower level languages. Each instruction is converted to several machine executable codes. This conversion is done in three steps: **preprocessing, compiling and linking**. An original program written using an EDITOR in a high level language is known as the **Source Code**. A compiled program is called the **Object Code** and the linked program is called the **Executable Code**. Examples of files created in each stage are: **Carpet.CPP, Carpet.OBJ, and Carpet.EXE**
Syntax: Rules for constructing a legal sentence in a programming language. If a program has syntax errors, it will not compile. Therefore, it produces **compile time errors**.
Semantics: Rules that determine the meaning of instructions written in a programming language.
Runtime and Logical errors: A runtime error occurs when the program cannot execute what is being asked to do. For example, open a file when it does not exist. Logical errors are most difficult to fix; it is caused by faulty logic by the programmer. For example, suppose you wanted write a program to give discount if purchase is greater than $100.00, instead you programmed to give discount if purchase is less than $100.00.
C++: The C (Dennis Ritchie) language is a modification of another language called BCPL (Ken Thompson). C language was written for programming under the Unix operating system environment which continues to be a very popular operating system for universities and businesses. C++ (Bjarne Stroustrup) is an object oriented version of C.
Constant : a data item that does not change during the execution of the program. A constant could be a named constant or a literal constant. Example of a literal constant is given in the following program. Example of a named constant is:

```
const  float TAX_RATE = 8.025; //TAX_RATE  is a named constant
                               // 8.025 is a literal constant
```

First C++ program: Call up the C++ programming environment from your windows desktop and type the following program in. Compile, and run it.

Program 1-1

```
/*******************************
        Say Hello program
        By Dr. John Abraham
        Created for 1380 students
Teaching objective: program structure
```

```
*******************************/

#include <iostream.h>                  //this is preprocessor directive
int main ()                            //this is the main function
{
cout << "See mom! I wrote my first C++ program\n";
return 0;
}
```

Program run

```
See mom! I wrote my first C++ program
```

IMPORTANT: Did this program run so fast that you could not see the output? Go to the end of this chapter where I describe how to stop the screen while running. See Program 1-1A

A program in C++ is a collection of one or more functions. This program has only one function named main. The function *main* is a required function in all programs. If there are several functions in a program, each function carries out a different task. The *main* function will call other functions into action.

Description of the program: Lines beginning with /* and ending with */ are comments, C++ ignores these lines. These comments are used for internal documentation. // may be used for commenting one line.

Preprocessor library: C language is a small language and does not have inherent input/output support. Input/output support is provided by modules (library). This program outputs one line of text to the monitor. The monitor is the standard output device and the keyboard is the standard input device. A pound sign # indicates that the remainder of that line is an instruction (directive) to the compiler. #include <iostream.h> tells the compiler to add the code found in that file to the beginning of this program. This code handles all character stream inputs and outputs. Without this you cannot read from the keyboard or display anything to the monitor.

A function: int main () is a function. A function receives one or more parameters (arguments) from the calling module and returns none or one result to the calling module. The word **int** before the function name (main) indicates that the function will return an integer to the calling module. The calling module of the main is the operating system. This program returns a zero to the operating system, indicating normal execution. The **()** after the function name indicates that this function does not receive any parameters from the calling module. Later we will see variations to this.

Begin and End of a block: The beginning of a body or compound statement is indicated by **{** and the ending is indicated by **}**.

Statement separator: A statement is separated from another statement by placing a semicolon between them. End of the line does not indicate end of a statement.

cout and cin: cout displays (prints) to an output device such as a monitor, and cin receives input from an input device such as a keyboard. The << or >> indicates the direction of flow of the data. In this program, "`See mom! I wrote my first C++ program\n,` is the data (in this case the data is a literal) that flows into the output device. Notice that the string literal is enclosed in double quotes. The last two characters in the literal **\n** makes the cursor to go to a new line. The backslash is used as a symbol for

escape. An escape sequence is used to control output devices. We will see additional escape sequences later.

The return statement: return 0, returns zero to the calling module through the main. In this case, the operating system is given the value 0, indicating that the program had a normal execution. Perhaps, another procedure would have returned a result of a calculation to the calling module.

Modification of the program: Adding one line to the program stops the screen for you press an enter key.

Program 1-1A

```
/*********************************
        Say Hello program
        By Dr. John Abraham
        Created for 1380 students
USE WITH TURBO C++ ONLY
*********************************/

#include <iostream.h>    //this is preprocessor directive

int main ()          //this is the main function
{
cout << "See mom! I wrote my first C++ program\n";
getchar();   //wait for the enter key to be pressed
return 0;
}
```

Arithmetic in C++

When we write code in C++ to do calculations, it is important to remember that results of integer and integer calculations may be different than real and real calculations. We also need to know how mixed number calculations will be carried out. C++ will allow you to assign a number with decimal to an integer, however, the fractional part will be discarded. Program 1-2 explores various arithmetic calculations.

Program 1-2

```
/*****************************************
        c++ Arithmetic
        By Dr. John Abraham
        Created for 1380 students
   Instructional objective: Arithmetic
*****************************************/

#include <iostream.h>      //this is preprocessor directive

int main ()          //this is the main function
{
 int i,j,k,l, m,n;
 float a,b,c;
 //integer operations
 cout << "INTEGER OPERATIONS\n \n";
 i = 6 + 3;
 l = 6.5;
```

```
 m = 3.5;
 j = l + m;
 k = 10 /3;
 n = 10 % 3;
 cout << "6 + 3 = " << i << "\n";
 cout << "l = 6.5, m = 3.5 --------->l + m = " << j << "\n";
 cout << "10 / 3 = " << k << "\n";
 cout << "10 % 3 = " << n << "\n";

 //real and mixed operations
 cout << "\nREAL AND MIXED OPERATIONS \n \n";
 a = 10 / 3;
 b = 10.0 / 3.0;
 c = 10.0 /3;
 cout << "10 / 3 = " << a << "\n";
 cout << "10.0 / 3.0 = " << b << "\n";
 cout << "10.0 / 3 = " << c << "\n";
 getchar();

return 0;
}
```

INTEGER OPERATIONS

6 + 3 = 9
l = 6.5, m = 3.5 --------->l + m = 9
10 / 3 = 3
10 % 3 = 1

REAL AND MIXED OPERATIONS

10 / 3 = 3
10.0 / 3.0 = 3.33333
10.0 / 3 = 3.33333

If l=6.5 and m =3.5 then l+m should be 10, why is it 9? We assigned these numbers to integer variables, which discards the fractional part leaving 6 and 3, which give a total of 9. How about 10/9 yielding 3? This is called integer division. The next problem 10 % 3 gives a result of 1, which is the remainder of the integer division (also known as the modulus). In the next problem even though we assigned the result of 10/3 to a real variable (a), the variable only received the result of an integer division. The result of 10.0/3.0 is 3.333333; here both numbers are real numbers (float). However the last problem 10.0/3 also gives 3.33333, why? In a mixed operation like this the integer is converted to float first, then the operation is carried out.

Home work

The terminology given at the beginning of this chapter is very important. You must learn it thoroughly. Make q-cards and memorize the terms.

Write this program over and over again until you do not have to look at the notes. You should not continue to the next chapter until you mastered this chapter thoroughly.

This homework may appear surprisingly easy to you. Don't be fooled. Many students do not finish the course because they do not spend much time with the first two chapters.

Write a program to determine the number of thousands, hundreds, tens, and ones in a given number. Hint: use integer division and modulus. Example of a program run:

In 8532 there are
8 thousands
5 hundreds
3 tens
2 ones.

Sending the output to a printfile

You are required to submit hard copies of all programs and program runs for every assignment given to you. You are required to submit structure charts and pseudocodes aswell. This addendum Chapter 1 describes how to send the output to a text file which can be printed from any program or directly from DOS.

Suppose you created a text file called carpet.txt from a program, this file then can be printed by opening it under Word, WordPerfect, or any other program you like and then printing from it. You can also print it by going to DOS prompt and typing the following: **type carpet.txt > prn**

Program OneA_1.

```
/*******************************************************
        Printing to a file
        By Dr. John Abraham
        Created for 1380 students
   Instructional objective: create print file One_2.txt
*******************************************************/

#include <iostream.h>
#include <fstream.h>    //to handle files

int main ()
{
 int i,j,k,l, m,n;
 float a,b,c;
 ofstream outfile;          //output file stream handle is outfile
 outfile.open("a:One_2.txt");  //assign a DOS name to the handle
                               //creates a file called One_2.txt in floppy drive A:
 //integer operations
 cout << "INTEGER OPERATIONS\n \n";
 i = 6 + 3;
 l = 6.5;
 m = 3.5;
 j = l + m;
```

```
k = 10 /3;
n = 10 % 3;
cout << "6 + 3 = " << i << "\n";
cout << "l = 6.5, m = 3.5 --------->l + m = " << j << "\n";
cout << "10 / 3 = " << k << "\n";
cout << "10 % 3 = " << n << "\n";
outfile << "6 + 3 = " << i << "\n";
outfile << "l = 6.5, m = 3.5 --------->l + m = " << j << "\n";
outfile << "10 / 3 = " << k << "\n";
outfile << "10 % 3 = " << n << "\n";

//real and mixed operations
cout << "\nREAL AND MIXED OPERATIONS \n \n";
outfile << "\nREAL AND MIXED OPERATIONS \n \n";
a = 10 / 3;
b = 10.0 / 3.0;
c = 10.0 /3;

cout << "10 / 3 = " << a << "\n";
cout << "10.0 / 3.0 = " << b << "\n";
cout << "10.0 / 3 = " << c << "\n";
outfile << "10 / 3 = " << a << "\n";
outfile << "10.0 / 3.0 = " << b << "\n";
outfile << "10.0 / 3 = " << c << "\n";
outfile.close();


return 0;
}
```

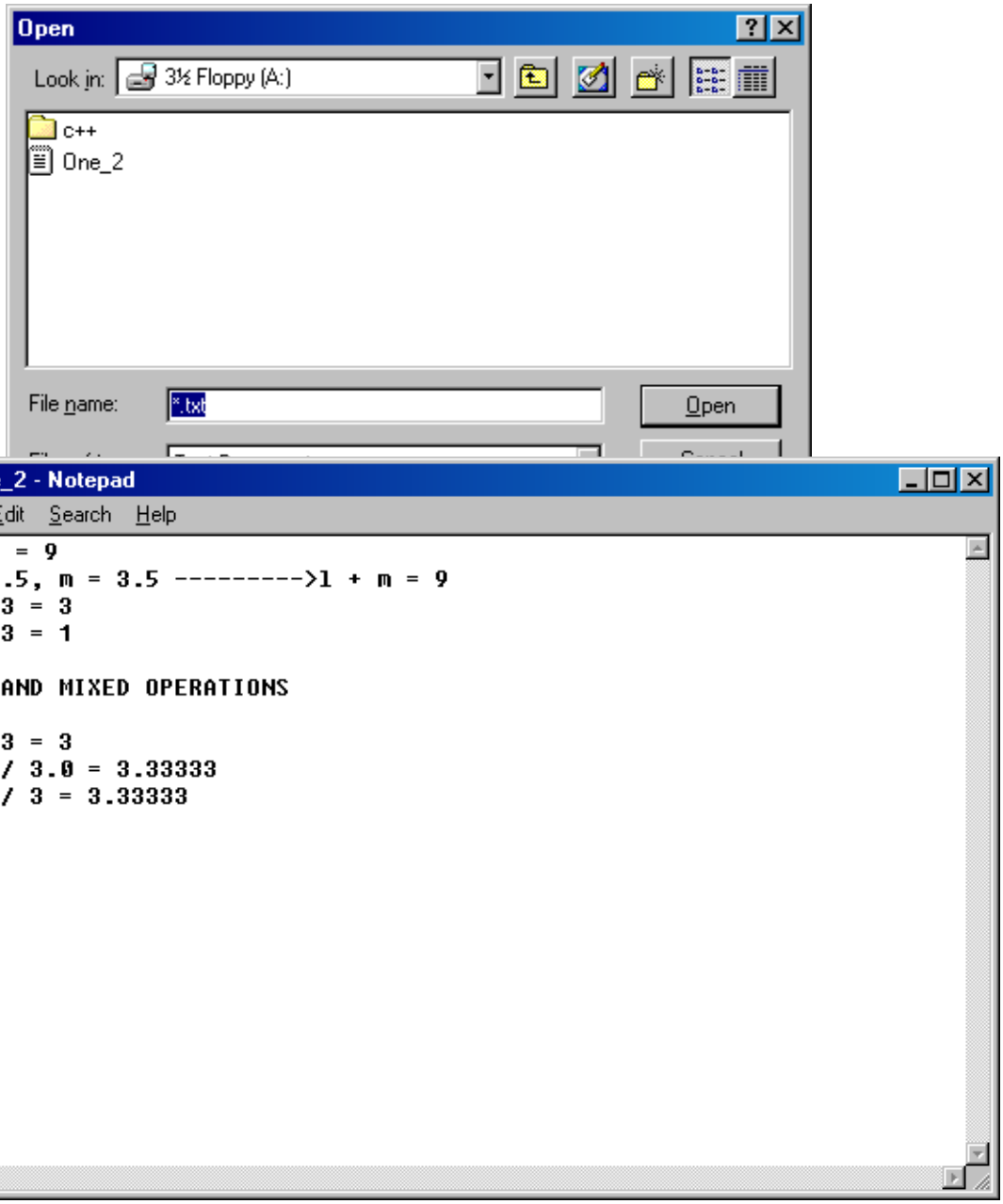
In order to create a textfile follow these steps:

1. Required preprocessor directive: #include <fstream.h>
2. Give a file handle such as outfile, printfile, textfile, or anything you like. Now on the file will be referred to using this name in the program.
3. Assign this handle to an actual DOS file name. DOS file names are 8 character long, a dot, and 3 character extension. You may include the drive letter and path in front of the file name. For example: **C:\mydocuments\cprograms\printfiles\carpet.txt**
4. Place any character stream you want in this file by using the << flow director. The easiest way to do this is to make copies of every **cout lines** in the program and change the cout to the file handle name. See example Program OneA_1.
5. Close the file.

To print the textfile created by this program, follow these steps:

1. Open Notepad from Start→Programs→Accessories
2. Open the file you created
3. From file option, choose print

You will be required to follow these steps for all program assignments now on.



Part 2 - Functions

When we build a house why do we make rooms? In our daily life why do we use so many different people who specialize in different professions, such as physicians, lawyers, accountants, etc? Same reasoning can be applied to programming. When we write a good program, it must be divided into modules; each module is designed to do a particular task. In C++ there are two types of modules, functions and classes. In this chapter we will deal with functions. You can write functions that can be used over and over again in a program or in several programs. In fact, the C++ standard library comes with many functions for mathematical, string, and character manipulations.

A large programming job needs to be broken down to manageable smaller modules; this breaking down process is called top down design. Here is one of my favorite hymns written using functions. I have broken down the hymn into five functions, four verses and one chorus. The main function simply calls the other functions. Even though the program is given in two text boxes for clarity, please combine them into one program.

Program 2-1

```
/*****************************
 Program: Function Calls
Written by: Dr. John P. Abraham
Instructional objective: Functions
*****************************/

#include <iostream.h>
void chorus();
void verse1();
void verse2();
void verse3();
void verse4();
void verse5();

int main()
{
  verse1();
  chorus();
  verse2();
  chorus();
  verse3();
  chorus();
  verse4();
  chorus();
  verse5();
  chorus();
return(0);
```

```
}
```

Program 2-1 continued:

```
void chorus()
  {
   cout<< "Hosts of angels anxiously ready and waiting\n";
   cout<< "They stand ready to bid me welcome\n";
   cout<< "All my robes  made so gleaming bright\n";
   cout<< "I will sing Hallelujah at His sight\n\n";
  }

void verse1()
{
        cout<< "I ran my race on this earth\n";
        cout<< "Stand ready for the price before me\n";
        cout<< "Transfigured I will fly home, heavenly home\n";
        cout<< "To stand in the presence of my Lord\\n\n";
}

void verse2()
{
        cout << "For years I hoped and longed\n";
        cout << "To see my King face to face\n";
        cout << "Soon I will behold him in Glory\n" ;
        cout << "I'll rest in His embracing arms\n\n";
}

void verse3()
{
        cout << "I long toiled and  labored\n";
        cout << "For my Master; now He honors me\n";
        cout << "With accolades and showers of\n";
        cout << "Gifts to decorate my crown\n\n";
}

void verse4()
{
        cout << "His saints and righteous servants\n";
        cout << "Those who gave their life for The Lord\n";
        cout << "As they play their strings to the Lord\n";
        cout << "I will sing and join that crowd\n\n";
}

void verse5()
{
        cout << "In the mansion made not by hand\n";
        cout << "In the city of new Jerusalem\n";
        cout << "Evermore as a bride in hand\n";
```

```
        cout << "I'll reign with my savior dear\n\n";
}
```

Program Run 2-1

```
I ran my race on this earth
Stand ready for the price before me
Transfigured I will fly home, heavenly home
To stand in the presence of my Lord

Hosts of angels anxiously ready and waiting
They stand ready to bid me welcome
All my robes  made so gleaming bright
I will sing Hallelujah at His sight

For years I hoped and longed
To see my King face to face
Soon I will behold him in Glory
I'll rest in His embracing arms

Hosts of angels anxiously ready and waiting
They stand ready to bid me welcome
All my robes  made so gleaming bright
I will sing Hallelujah at His sight

I long toiled and  labored
For my Master; now He honors me
With accolades and showers of
Gifts to decorate my crown

Hosts of angels anxiously ready and waiting
They stand ready to bid me welcome
All my robes  made so gleaming bright
I will sing Hallelujah at His sight

His saints and righteous servants
Those who gave their life for The Lord
As they play their strings to the Lord
I will sing and join that crowd

Hosts of angels anxiously ready and waiting
They stand ready to bid me welcome
All my robes  made so gleaming bright
I will sing Hallelujah at His sight

In the mansion made not by hand
In the city of new Jerusalem
```

```
Evermore as a bride in hand
I'll reign with my savior dear

Hosts of angels anxiously ready and waiting
They stand ready to bid me welcome
All my robes  made so gleaming bright
I will sing Hallelujah at His sight
```

When we run this program, the hymn is displayed verse1, chorus, verse2, chorus, and so on. Look at the main, it is not cluttered. In the main, the functions were **called**. The function *chorus* was called four times; all other functions were called only once. The main function **returned** a zero to the operating system indicating all is well; all other procedures returned nothing. When a function does not return anything, it is called a **VOID** function. Before calling a function, either that function must be defined, or the function **prototype** must be declared. In this particular example, if all functions were defined before the main there is no need to have the prototypes.

Please underline the prototypes just above the main in order for you to remember to do them when you write your own programs. A function prototype contains the following information: (1) name of the function, (2) the type of data returned by the function (or VOID), and (3) number, type, and order of parameters passed into the function. Let us write another program to find centigrade given Fahrenheit.

Program 2-2

```
/*************************************
 Program: Function Calls
Written by: Dr. John P. Abraham
Instructional objective: Passing Parameters
*************************************/



#include <iostream.h>  /* in newer versions of C++ replace this one line with the
                          following two lines:
                           #include <iostream>
                           using namespace std; */

float celsius(float);  //prototype ends with a semicolon.

int main ()
{
  float c, f;  //Centigrade and Fahrenheit
  cout << "Enter today's temperature in degrees of Fahrenheit ";
  cin >> f;
  c = celsius(f);
  cout << "That equals " << c << " degrees of celsius!\n" ;
  return(0);
```

```
  cout << "Press the Enter key to stop the Program!";
}

float celsius (float f)

{
 return ((f-32)*5/9);
}
```

Program Run 2-2

```
Enter today's temperature in degrees of fahrenheit 92
That equals 33.3333 degrees of celcius!
```

The function prototype is given immediately after the include statement. It can be said that this prototype is declared globally. A globally declared function or variable is available (visible) to all functions below it. The prototype indicates that the function-celsius will receive one float parameter and will return one float value back. In the main variable c is assigned the result of the function-celsius. It is very important that you understand this concept of passing parameters and returning results. Please type this program in as many times as required until you can do it without referring to the notes.

Try to answer these questions about this program. What are the similarities and differences between the prototype and the function heading? Is prototype required in this case? Why or why not? Why could this function not be a VOID function? Where is the function call found? What happens if the function is not called? Is there any difference between calling a VOID function and a value returning function like this example? Explain. **Please do not proceed until you are able to answer these questions.**

Next we need to concentrate on the **scope of variables**. This world has lots of people called John. So when someone calls John, which John should answer? In a program, you may use the same identifier name in different functions. There needs to be some rules about the scope of the variables. *The scope of variables concept is very important, please pay attention to every sentence in the following few paragraphs*. When a parameter is passed in, as in the example in program 2-2, the contents of that memory location (referred to as f in this example) is sent to the called function (celsius). This variable exists in that function only during the execution of that function. If that function is called again, the old value is gone and whatever is passed now is placed there. Run the following program to understand this concept.

Program 2-3

```
/*****************************
 Program: Function Calls
Written by: Dr. John P. Abraham
Instructional objective: Scope of variables
*****************************/
```

```
#include <iostream.h>

void changex (int);
int main ()
{
  int x;
  x = 5;
  cout << "In the main, value of x before the first call ----> " << x <<"\n";
  changex(x);
  cout << "In the main, value of x after the first call -----> " << x <<"\n";

  x = 1;
  cout << "In the main, value of x before the second call ----> " << x <<"\n";
  changex(x);
  cout << "In the main, value of x after the second call -----> " << x <<"\n";
 return (0);
}

void changex (int x)
{
 cout << "    in the function, value of x as it came in----> " << x << "\n";
 x = x+10;
 cout << "    in the function, value of x after adding 10---> " << x << "\n";
}
```

Program Run 2-3

```
In the main, value of x before the first call ----> 5
    in the function, value of x as it came in----> 5
    in the function, value of x after adding 10---> 15
In the main, value of x after the first call -----> 5
In the main, value of x before the second call ----> 1
    in the function, value of x as it came in----> 1
    in the function, value of x after adding 10---> 11
In the main, value of x after the second call -----> 1
```

When running this program please notice that x is first assigned the value of 5 and this value is passed into the function as a copy (now x has a copy). The x is changed in the function to 15 by adding a 10. However after the function is executed, everything in the function including the x in the function, is erased. The original x in the main still has the value of 5. There is a way to keep the values of variables in a function. This is done by declaring the variables as **static**. This will not be discussed further here.

When a function is called and copies of the parameters are passed in as discussed in the previous paragraph, it is known as **call-by-value or pass-by-value**. Another way to pass a parameter is to **call-by-reference or pass-by-reference**. When a parameter is

passed using **call-by-reference,** the **address** (not the copy) of the **original** memory location is passed to the function. The function may change the contents of that memory location. The advantage here is that by not duplicating large blocks of data, memory can be saved. A disadvantage is accidentally changing (**side effect**) the value of a variable. However, this can be an effective way of passing more than one value back and forth between a called and calling module.

Now let us study the concept of **global variables**. A variable that is visible to a lower level module is called a global variable. In the program 2-4, x is declared as a global variable. This variable is visible by all modules and any module can change its value. Global variables are also susceptible to side effects.

Program 2-4

```
/*****************************************
 Program: Function Calls
Written by: Dr. John P. Abraham
Instructional objective: Global variables
*****************************************/

#include <iostream.h>

int x;
void getvalue ();
int main ()
{
  x = 1;
  getvalue();
  cout << "x contains a value of " << x <<"\n";
  return(0);
}

void getvalue()
 {
  cout << "enter a value for x ";
  cin >> x;
 }
```

Program Run 2-4

```
enter a value for x 8
x contains a value of 8
```

Whatever you entered for the x in the function was displayed in the main; the function placed a value in x and the main was able to access that value. You can see the problems global variables can cause if you are not extremely careful. Therefore, use of global variables should be avoided unless you are a seasoned programmer.

It is time for us to examine the concept of **local variables**. Suppose a global variable x exists, and you declared another x in a function, and now you want to assign a value to x. Which x should get the value? Let us modify program 2-4 to include a local variable. When a module refers to a variable, it is located in this hierarchy: (1) check local declarations; if the variable is not found in the local declarations, (2) check the parameter list in the heading of that function; if it is not found there, (3) check globally declared variables.

Program 2-5

```
/*****************************
 Program: Function Calls
Written by: Dr. John P. Abraham
Instructional objective: local variables
*****************************/

#include <iostream.h>

int x;
void getvalue ();
int main ()
{
  x =1;
  getvalue();
  cout << "x contains a value of " << x <<"\n";
  return(0);
}

void getvalue()
 {
  int x;
  cout << "enter a value for x ";
  cin >> x;
 }
```

Program Run 2-5

```
enter a value for x 15
x contains a value of 1
```

The x in the function getvalue is a local variable. Now you have an x global and an x local. When you assign a value to x, the local gets it and the global is not changed. The global was assigned a value of 1 in the main and the program displays a one.

Reference Parameters

As we studied earlier in the module there are parameters in the function call and function heading. Those parameters found in the function call are called **actual**

parameters or **arguments**. Those found in the function heading are called **formal parameters**. Formal parameters can be passed by value or passed by reference.

Let us write a program to illustrate call-by-reference. When a parameter is passed by reference, any changes made to that variable in the function actually is changing the original variable. Both variables (in calling and called modules) refer to the same memory location. In the following example (Program 2-6), length and width are passed by reference. In order to pass by reference we use an **ampersand (&)** in front of the variable name. Note how the prototype is declared; here also we use the ampersands. Another interesting feature of functions is that you do not have to use the same identifier in the called and calling modules. For example, in the calling module the identifiers were length and width; in the called module the identifiers were l and w. The names do not have to match, but their order (position) has to.

Program 2-6

```
/*******************************
  Program to Find square feet
  Instructional objective: Passed-by-reference.
  By Dr. John Abraham
  Created for 1380 students
*******************************/


#include <iostream.h>


void getmeasurements (int &, int &);
float calc_sqyards(int, int);
int main()
 {
        int length, width;
        float sqyards;
        getmeasurements(length, width);
        sqyards = calc_sqyards(length, width);
        cout << "The area in square yards is " << sqyards;
        return (0);
 }

void getmeasurements(int &l, int &w)
 {
  cout << "Enter length of the room in feet ";
  cin >> l;
  cout << "Enter width of the room in feet ";
  cin >> w;
 }
```

```
float calc_sqyards (int length, int width)

{
 float area;
 area = length * width /9;
 return(area);
}
```

Program Run 2-6

```
Enter length of the room in feet 18
Enter width of the room in feet 15
The area in square yards is 30
```

Assignment

Write a carpet estimation program to

1. accept the length and width of a room in feet.
2. accept price of the carpet per square yard.
3. find the area of the room in square yards.
4. find the cost of the carpet for the room.
5. calculate sales tax using globally declared tax rate constant.
6. display all pertinent information for the estimate on the screen.

More on Functions

In this chapter we will review functions, follow an example program from planning to completion, and learn two new concepts, namely inline functions and function overloading. A function is a subprogram that specializes in doing one specific task. A function by definition may receive none, one, or many input parameters and output none or one parameter.

None or One output argument ß **| Function** | ß None, one or many input argument(s)

```
float findAverage (int grade1, int grade2, int grade3)
```

In this example the function **findAverage** receives 3 input arguments and returns one argument. The input arguments here are **passed by value**, meaning only copies of the three original variables appearing in the calling function are passed to the function. Even if the values of these variables are changed by the function, the original variables will

remain unchanged. If a function does not return any result back just indicate so by stating **void**, example: void print (int grade1, int grade2, int grade3, float average).

Remember that the function only exists during its execution. Upon return from the function nothing remains of the function. A function is made upon calling it and destroyed upon returning. In order to call the above function, we can write the following statement:

average = findAverage(grade1, grade2, grade3);

Since we know that this function returns a float we must put this returned value somewhere; here we receive the result into **average**. We could have displayed the result directly on the screen by the statement:

cout << findAverage(grade1, grade2, grade3);

It is often necessary to use functions to read multiple variables and return them to the calling module. Based on the above definition it is impossible to send back more than one value. Here is where argument **passing by reference** comes in handy. Instead of returning the result, the function writes directly to the original memory locations of the reference variables. In the following is example, even though nothing is returned by the function, the calling function's three variables were changed with the read values.

void getGrades (int &grade1, int &grade2, int &grade3)

Suppose we want to write a program to find the average of three grades. We can generate a structure chart with three modules, getGrades, findAverage, and print. The Module getGrades receives three arguments by reference and returns none. The findAverage module receives one argument by value and returns a float. The last module receives four arguments by value and returns none. Draw a structure chart using these directions. The next step is to develop the prototypes. Translating a properly designed structure chart to prototypes is rather easy. The prototype for these three functions would be:

void getGrades (int&, int&, int&);

float findAverage (int, int, int);

void print (int, int, int, float);

Finally, let us write the whole program. The main should only declare variables that it uses. Do not declare variable a function might use as local variables. Convert the prototypes to functions and function calls.

```
/************************************

Program to find average of 3 grades

objective: Review functions
```

```
Program by Dr. Abraham
************************************/



#include <iostream.h>
#include <fstream.h>


void getGrades (int&, int&,int&); //function prototypes
float findAverage (int, int, int );
void print (int, int, int, float);
ofstream outfile;                       //outfile is declared globally
int main ()
{
        outfile.open ("a:grade.dat");
        float average;
        int grade1, grade2, grade3;
        getGrades(grade1, grade2, grade3);
        average = findAverage(grade1, grade2, grade3);
        print(grade1, grade2, grade3, average);
        outfile.close();
        return(0);
}
void getGrades (int &c, int &b, int &a) // variable names changed purposely
{
        cout << "Enter first grade ----------> ";
        cin >> c;
```

```
        cout << "Enter second grade ----------> ";
        cin >> b;
        cout << "Enter third grade ---------- > ";
        cin >> a;
        outfile << "\nEnter first grade ----------> " << c;
        outfile << "\nEnter second grade ----------> " << b;
        outfile << "\nEnter third grade ---------- > " << a;
}
float findAverage (int one, int two, int three) //variable names changed purposely
{
int total;
float a;
total = one+two+three;
a = (total)/3.0;
return a;
}



void print (int grade1, int grade2, int grade3, float average)
{
cout << "\nThree grades entered are: " << grade1 << " " <<grade2 << " " << grade3;
cout << "\nAverage is " << average <<endl;
outfile << "\nThree grades entered are: " << grade1 << " " <<grade2 << " " << grade3;
outfile << "\nAverage is " << average <<endl;
}
```

```
Enter first grade ----------> 92
Enter second grade -----------> 88
Enter third grade ---------- > 86
Three grades entered are: 92 88 86
Average is 88.6667
Press any key to continue
```

Inline Functions:

We can write a function that is so small that it will fit in one line. For example the findAverage function can be written like this:

inline float findAverage (int a, int b, int c) {return (a+b+c)/3.0;};

```
/************************************
Program to find average of 3 grades
objective: Review functions
Program by Dr. Abraham
************************************/



#include <iostream.h>
#include <fstream.h>


void getGrades (int&, int&,int&); //function prototypes
inline float findAverage (int a, int b, int c) {return (a+b+c)/3.0;};
```

```
void print (int, int, int, float);
ofstream outfile;                          //outfile is declared globally
int main ()
{
        outfile.open ("a:grade.dat");
        float average;
        int grade1, grade2, grade3;
        getGrades(grade1, grade2, grade3);
        average = findAverage(grade1, grade2, grade3);
        print(grade1, grade2, grade3, average);
        outfile.close();
        return(0);
}
void getGrades (int &c, int &b, int &a) // variable names changed purposely
{
        cout << "Enter first grade ----------> ";
        cin >> c;
        cout << "Enter second grade -----------> ";
        cin >> b;
        cout << "Enter third grade ---------- > ";
        cin >> a;
        outfile << "\nEnter first grade ----------> " << c;
        outfile << "\nEnter second grade -----------> " << b;
        outfile << "\nEnter third grade ---------- > " << a;
}
```

```
void print (int grade1, int grade2, int grade3, float average)
{
cout << "\nThree grades entered are: " << grade1 << " " <<grade2 << " " << grade3;
cout << "\nAverage is " << average <<endl;
outfile << "\nThree grades entered are: " << grade1 << " " <<grade2 << " " << grade3;
outfile << "\nAverage is " << average <<endl;
}
```

Function Overloading:

It is possible for a function to calculate average of real numbers yielding a real number or to calculate integer average yielding an integer. To do this we have to write two functions with different set of parameters but using the same function name, **average**. Such functions are called overload functions. When an overloaded function is called, C++ compiler chooses appropriate function based on parameters passed. This will be dealt with at later chapters.

More on Functions

In this chapter we will review functions, follow an example program from planning to completion, and learn two new concepts, namely inline functions and function overloading. A function is a subprogram that specializes in doing one specific task. A function by definition may receive none, one, or many input parameters and output none or one parameter.

None or One output argument ← **| Function** | ← None, one or many input argument(s)

float findAverage (int grade1, int grade2, int grade3)

In this example the function **findAverage** receives 3 input arguments and returns one argument. The input arguments here are **passed by value**, meaning only copies of the three original variables appearing in the calling function are passed to the function. Even if the values of these variables are changed by the function, the original variables will remain unchanged. If a function does not return any result back just indicate so by stating **void**, example: void print (int grade1, int grade2, int grade3, float average).

Remember that the function only exists during its execution. Upon return from the function nothing remains of the function. A function is made upon calling it and destroyed upon returning. In order to call the above function, we can write the following statement:

average = findAverage(grade1, grade2, grade3);

Since we know that this function returns a float we must put this returned value somewhere; here we receive the result into **average**. We could have displayed the result directly on the screen by the statement:

cout << findAverage(grade1, grade2, grade3);

It is often necessary to use functions to read multiple variables and return them to the calling module. Based on the above definition it is impossible to send back more than one value. Here is where argument **passing by reference** comes in handy. Instead of returning the result, the function writes directly to the original memory locations of the reference variables. In the following is example, even though nothing is returned by the function, the calling function's three variables were changed with the read values.

void getGrades (int &grade1, int &grade2, int &grade3)

Suppose we want to write a program to find the average of three grades. We can generate a structure chart with three modules, getGrades, findAverage, and print. The Module getGrades receives three arguments by reference and returns none. The findAverage module receives one argument by value and returns a float. The last module receives four arguments by value and returns none. Draw a structure chart using these directions. The next step is to develop the prototypes. Translating a properly designed structure chart to prototypes is rather easy. The prototype for these three functions would be:

void getGrades (int&, int&, int&);

float findAverage (int, int, int);

void print (int, int, int, float);

Finally, let us write the whole program. The main should only declare variables that it uses. Do not declare variable a function might use as local variables. Convert the prototypes to functions and function calls.

```
/************************************

Program to find average of 3 grades

objective: Review functions

Program by Dr. Abraham

************************************/




#include <iostream.h>

#include <fstream.h>


void getGrades (int&, int&,int&); //function prototypes

float findAverage (int, int, int );

void print (int, int, int, float);

ofstream outfile;                    //outfile is declared globally

int main ()
```

```
{
	outfile.open ("a:grade.dat");
	float average;
	int grade1, grade2, grade3;
	getGrades(grade1, grade2, grade3);
	average = findAverage(grade1, grade2, grade3);
	print(grade1, grade2, grade3, average);
	outfile.close();
	return(0);
}
void getGrades (int &c, int &b, int &a) // variable names changed purposely
{
	cout << "Enter first grade ----------> ";
	cin >> c;
	cout << "Enter second grade -----------> ";
	cin >> b;
	cout << "Enter third grade ---------- > ";
	cin >> a;
	outfile << "\nEnter first grade ----------> " << c;
	outfile << "\nEnter second grade -----------> " << b;
	outfile << "\nEnter third grade ---------- > " << a;
}
float findAverage (int one, int two, int three) //variable names changed purposely
{
int total;
float a;
total = one+two+three;
a = (total)/3.0;
return a;
```

```
}



void print (int grade1, int grade2, int grade3, float average)
{
cout << "\nThree grades entered are: " << grade1 << " " <<grade2 << " " << grade3;
cout << "\nAverage is " << average <<endl;
outfile << "\nThree grades entered are: " << grade1 << " " <<grade2 << " " << grade3;
outfile << "\nAverage is " << average <<endl;
}
```

```
Enter first grade ----------> 92
Enter second grade -----------> 88
Enter third grade ---------- > 86
Three grades entered are: 92 88 86
Average is 88.6667
Press any key to continue
```

Inline Functions:

We can write a function that is so small that it will fit in one line. For example the findAverage function can be written like this:

inline float findAverage (int a, int b, int c) {return (a+b+c)/3.0;};

```
/**********************************
Program to find average of 3 grades
objective: Review functions
```

```
Program by Dr. Abraham
**************************************/



#include <iostream.h>
#include <fstream.h>


void getGrades (int&, int&,int&); //function prototypes
inline float findAverage (int a, int b, int c) {return (a+b+c)/3.0;};
void print (int, int, int, float);
ofstream outfile;                       //outfile is declared globally
int main ()
{
        outfile.open ("a:grade.dat");
        float average;
        int grade1, grade2, grade3;
        getGrades(grade1, grade2, grade3);
        average = findAverage(grade1, grade2, grade3);
        print(grade1, grade2, grade3, average);
        outfile.close();
        return(0);
}
void getGrades (int &c, int &b, int &a) // variable names changed purposely
{
        cout << "Enter first grade ----------> ";
        cin >> c;
        cout << "Enter second grade -----------> ";
        cin >> b;
        cout << "Enter third grade ---------- > ";
```

```
        cin >> a;
        outfile << "\nEnter first grade ----------> " << c;
        outfile << "\nEnter second grade -----------> " << b;
        outfile << "\nEnter third grade ---------- > " << a;
}



void print (int grade1, int grade2, int grade3, float average)
{
cout << "\nThree grades entered are: " << grade1 << " " <<grade2 << " " << grade3;
cout << "\nAverage is " << average <<endl;
outfile << "\nThree grades entered are: " << grade1 << " " <<grade2 << " " << grade3;
outfile << "\nAverage is " << average <<endl;
}
```

Function Overloading:

It is possible for a function to calculate average of real numbers yielding a real number or to calculate integer average yielding an integer. To do this we have to write two functions with different set of parameters but using the same function name, **average**. Such functions are called overload functions. When an overloaded function is called, C++ compiler chooses appropriate function based on parameters passed. This will be dealt with at later chapters.

Part 3 - Branching

When instructions within a program are executed one after the other sequentially that program is said to have a linear structure. Decision making after examining all available options is very important in life as well as in programming. For example, it is the law that all males 18 or older should register with the selective service. If you are writing a program to send out reminders to enforce this law, the decision to send the letter should be based on if a person is male and if he is 18 or older. In this chapter you will learn how to write statements that make decisions. A simple program to look at an average grade of a student and display if that student passed or failed would look like this (Program 3-1):

Program 3-1.

```
/**************************************
```

```
Accept three grades, find the average
and display Passed or Failed.
Teach objective - making decisions
By Dr. John Abraham
Created for 1380 students
**************************************/
#include <iostream.h>
#include <iomanip.h> //to format input and output.
                                                    //Here setw and endl require it
float findave (int &, int &, int &);
int main ()
{
int one, two, three;
float average;
average= findave (one, two, three);
cout << "Three grades are: " <<one <<setw(4)<<two <<setw(4)<<three <<endl;
//endl inserts line feed and carriage return
cout << "The average is : " <<average <<endl;
if (average >= 70) cout << "Student passed the course!";
else cout << "Student failed the course!";
return (0);
}
float findave(int &one, int &two, int &three)
{
cout << "Enter three grades ";
cin >> one >> two >> three;
```

```
return (float(one+two+three) / 3.0);
// the total of three grades are first converted to float
}
```

Program Run 3-1

```
Enter three grades 88 44 99
Three grades are: 88 44 99
The average is : 77
Student passed the course!
```

The decision was made using the if-else statements:

```
if (average >= 70) cout << "Student passed the course!";
else cout << "Student failed the course!";
```

We could add compound statements under the if condition or the else condition or both like this:

```
if (average >=70)
                {
                cout << "Student Passed the Course!\n";
                 cout << "The student is allowed to take the next course.";
                }
else
                {
                 cout << "Student failed the course!\n";
                 cout << "This course must be repeated before taking the next course!";
```

}

The operator symbols we used here >= are called **Relational Operators. Relational operators are == (equal to), > (greater than), < (less than), != (not equal to), >= (greater than or equal to), and <= (less than or equal to).** The result of a relational operation is either **false** or **true**. Variables that hold **false** or **true are** called **bool** variables.

There were only two alternatives in the above situation, either passed or failed. In actual grading we want to go beyond pass/fail; we want to assign a letter grade of A, B, C, D, or F. Let us rewrite the above program to handle the multiple alternatives. See Program 3-2.

Program 3-2.

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

Accept three grades, find the average

and display the letter grade.

Teaching objective - multiple alternatives

By Dr. John Abraham

Created for 1380 students

****************************************/

#include <iostream.h>

#include <iomanip.h> //to format input and output.

                                                    //Here setw and endl require it

float findave (int &, int &, int &);

char getgrade(int);

int main ()

{

int one, two, three;

float average;

char grade;

average= findave (one, two, three);
```

```
cout << "Three grades are: " <<one <<setw(4)<<two <<setw(4)<<three <<endl;
//endl inserts line feed and carriage return
cout << "The average is : " <<average <<endl;
grade = getgrade(average);
cout << "The letter grade is : "<< grade << "\n";
getchar ();
}
float findave(int &one, int &two, int &three)
{
cout << "Enter three grades ";
cin >> one >> two >> three;
return (float(one+two+three) / 3.0);
// the total of three grades are first converted to float
}

char getgrade (int average)
{
if (average >=90)return ('A');
else if (average>= 80) return('B');
else if (average >=70) return ('C');
else if (average >= 60) return ('D');
else return('F');
}
```

In the getgrade function if the average is 90 or above the function returns an A and ends the function; it only continues if the average is not 90 or above. You may want to modify the program as follows to avoid many **return** statements. The program reads better when you only have one **return**.

```
char getgrade (int average)
{
char grade;
if (average >=90) grade = ‘A’;
else if (average>= 80) grade = 'B';
else if (average >=70) grade ='C';
else if (average >= 60) grade ='D';
else grade ='F';
return (grade);
}
```

Program Run 3-2.

```
Enter three grades 90 93 89
Three grades are: 90 93 89
The average is : 90.6667
The letter grade is : A
Enter three grades 66 58 52
Three grades are: 66 58 52
The average is : 58.6667
The letter grade is : F
Enter three grades 77 82 75
Three grades are: 77 82 75
The average is : 78
The letter grade is : C
```

Any time you write a program for multiple alternatives, the program should be run to check every alternative. In program Run 3-2 only three alternatives are tested. If you were to turn this program in for a grade, you should include all the alternatives.

Logical operators work with boolean values or results of relational operations. Logical operators are: **AND (&&)**, **OR (||)**, and **NOT (!)**. For each of this operation we can obtain a **truth table**.

T && T => T **T || T => T** **!T => F**

T && F => F **T || F => T** **!F =>T**

F && F => F **F || F => F**

Suppose the average score is 95. Let us try this statement:

```
If (average >=90 && average <= 100)
        cout << "Your grade is A \n";
```

The first relational operation of average >= 90 yields a T. The second operation of average <= 100 also yields a T. T && T gives a T. Since the entire operation yields a true then "Your grade is A" is displayed. If, on the other hand, the average score is 88, the first operation will yield a false and the second operation will yield a true (88 is less than 100); F && T is False, and the output will not be displayed. The concept of logical and relational operators will become more clear in the next chapter when we deal with repetitions.

Suppose you created a menu to choose one of the items from a list you may have to write some thing like this:

```
If (choice==1) AddClient ();

else if (choice == 2) EditClientInfo ();

else if (choice==3) LookUpClient();

and so on…
```

If the menu has many items there is a lot of coding you have to do and the code is hard to follow. There is alternative to the multiple if/else statements. We can use the **switch statement** as shown below.

```
Switch (choice)

{

case 1: AddClient();
         break;

case 2: EditClientInfo();
         break;

case 3: LookUpCleint();
```

```
        break;
and so on..
default :
        cout << "invalid response";
}
```

We are essentially telling c++ to do something in case of choice is 1, 2, or 3 and so on. If a match is not found then it falls to the default and carries out that instruction. You need to remember that you cannot use any relational operators with the case statement such as case >3. What happens if you do not include the break statements? Every case statement will be executed until it finds the break statement or until the end of the block. Try deleting the break statements and see what happens.

Here is a complete example of a program. Let us write a program to recieve three grades, find its average, determine the letter grade and write a brief comment about the grade.

Program Three-3

```
/*****************************************
Accept three grades, find the average
and display the letter grade.
Teaching objective - multiple alternatives
By Dr. John Abraham
Created for 1380 students
*****************************************/

#include <iostream.h>
#include <iomanip.h> //to format input and output.
                                        //Here setw and endl require it
float findave (int &, int &, int &);
char getLetterGrade(int);
void Message(int one, int two, int three, float average, char grade);
```

```
int main ()
{
int one, two, three;
float average;
char grade;
average= findave (one, two, three);
grade = getLetterGrade(average);
Message (one, two, three, average, grade);
return 0;
}

float findave(int &one, int &two, int &three)
{
cout << "Enter three grades ";
cin >> one >> two >> three;
return (float(one+two+three) / 3.0);
// the total of three grades are first converted to float
}

char getLetterGrade (int average)

{
char grade;
if (average >=90) grade = 'A';
else if (average>= 80) grade = 'B';
else if (average >=70) grade ='C';
else if (average >= 60) grade ='D';
```

```
else grade ='F';
return (grade);
}


void Message (int one, int two, int three, float average, char grade)
{
        cout << "Three grades are: " <<one <<setw(4)<<two <<setw(4)<<three <<endl;
cout << "The average is : " <<average <<endl;
        cout << "\nThe letter grade is : "<< grade << endl;

        switch (grade)
        {
        case 'A' : cout << "Very impressive grade indeed!\n";break;
        case 'B' : cout << "A solid performance, congratulations!\n"; break;
        case 'C' : cout << "C++ is a tough course, but YOU MADE IT!\n";break;
        case 'D' : cout << "Made it eh? \n";break;
        case 'F' : cout << "Don't give up. Try keeping up with all the homework!\n";
        }

}
```

Program Run Three-3

Enter three grades 80 85 99

Three grades are: 80 85 99

The average is : 88

The letter grade is : B

A solid performance, congratulations!

Press any key to continue

Assignment

Write a program to display the name of the month, given its number. For example if you enter 4 for the month, it should display April. Write it using if/else and then modify it to use the case statement.

Explain purpose the else statement in the above program? What happens if you do not use else.

Include Files

Many programmers use same functions or code segments repeatedly in many of the programs they write. For example, you have been writing the same code to prepare a printer file every time you write a program. It would be convenient if we were able to save the necessary code to open and close a printer file so that we can use it with every program we write. In fact, C++ provides such a convenience through the include files.

To write a header file, launch the Visual C++ and choose to create a new file. From the Files Tab choose C++ header file. Write the code as shown in Program 5-1 and save it as printer.h file.

Program 5-1

```
/**************************

Include file for Printer output

Provided by Dr. John Abraham

For CSCI 1380 students
```

```
**************************/

#include <fstream.h>

ofstream printer;

void openPrinter()

{

        char string[20];

        cout <<"Enter name of the printer file-> ";

        cin >>string;

        printer.open(string);

}

void closePrinter()

{

        printer.close();

}
```

Once the header file is saved, you can include these files in any future programs you write. Let us modify the program we wrote in the previous chapter. Write the cpp program as it appears in Program 5-2. Compile and run the program.

Program 5-2

```
/****************************************

Display 1 to 100 on the screen

and send the output to a file.

Teaching objective - include files.

By Dr. John Abraham

Created for 1380 students

****************************************/

#include <iostream.h>

#include <c:\tempc\printer.h> //substitute with your file name

int main()

{
```

```
openPrinter();                          //call function from printer.h
int number;
number = 1;
while (number <= 100)
{
cout << number << " "; //display number and put some spaces
printer << number << "\n";
        number ++; //increment number by one
}
return (0);
closePrinter();
}
```

Part 4 - Iteration

Suppose you want to display numbers 1 to 100 on the screen. Based on what we studies so far, you will have to write one hundred cout statements. If you think about, all you are doing is **adding one to the previous number and displaying it** over and over again until 100 has been displayed. Let us write the steps to do it.

Number gets 1.
Display the number
Add one to it
Repeat these two statements.
Stop when 100 has been displayed.

Let us rework it.
Number = 1
Do the following statements (in brackets) while number less than or equal to 100.
{
display number
add one to number
}

Here are the steps:

1. **Initialize** the variable (this variable is also called the loop control variable or LCV), because this variable controls the loop. In this example the **LCV** is **number**.
2. Check for the condition to enter the loop. The condition should yield a **True** to **enter** the loop. If the condition yields a **false**, the loop is **not entered**.
3. Set up the **body** of the loop. You may have one or multiple things to do within the loop body. The body here appears within the brackets.
4. **Change** the value of the **LCV** within the body. In this example the number is changed by adding a one to it.

These steps will work in all programming languages. Let us write this program in c++. See program 4-1.

Program 4-1

```
/*****************************************
Display 1 to 100 on the screen
Teaching objective  - while loop
By Dr. John Abraham
Created for 1380 students
*****************************************/

#include <iostream.h>

int main()
{
  int number;
  number = 1;
  while (number <= 100)
    {
     cout << number << "   ";       //display number and put some spaces
     number ++;                      //increment number by one
    }
  getchar();
  return (0);
}
```

Program Run 4-1

```
  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18
 19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
 35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66
 67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82
 83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98
```

This while loop is a **count controlled loop**, since we wanted to repeat the loop a certain number of times. The count controlled loop may also be referred to as a **step controlled loop**. We can write a **sentinel controlled loop** as well. A sentinel value is a value that belongs to a type but does not belong to the set you are working with. For example suppose you are entering names. Names belong a type called string. When asked for the name what if you entered 'quit'?. 'Quit' also belongs to the string type, however, does not belong to the set of names. Parents do not name a child Quit! Let me illustrate it with Program 3-2.

Program 4-2

```
/*****************************************
Accept and display names
Teaching objective  - while loop/sentinel controlled
By Dr. John Abraham
Created for 1380 students
*****************************************/

#include <iostream.h>
#include <string.h>

int main()
{
  string name;
   cout << "Enter a name ";
   cin >> name;
   while (name !="quit")
     {
       cout << "Hello, " << name  <<"\n";
       cout << "Enter another name or type 'quit' to exit ";
       cin >> name ;
     }
   getchar();
   return (0);
}
```

Program Run 4-2

```
Enter a name Randy
Hello, Randy
Enter another name or type 'quit' to exit Sandy
Hello, Sandy
Enter another name or type 'quit' to exit James
Hello, James
Enter another name or type 'quit' to exit Roger
```

```
Hello, Roger
Enter another name or type 'quit' to exit Jill
Hello, Jill
Enter another name or type 'quit' to exit quit
```

What if you wanted to keep accepting names until ‘quit’ is entered, but do not want to accept more than 5 names? To do this we will make some modifications to the Program 4-2. First, we will change the test to include if the number of names is less than or equal to 5. Second we will increment a counter when a name is entered. See program 4-2A.

Program 4-2A

```
/*****************************************************************
Accept and display names
Teaching objective  - while loop/sentinel  and count ontrolled
By Dr. John Abraham
Created for 1380 students
*****************************************************************/

#include <iostream.h>
#include <string.h>        //to handle string functions.

int main()
{
  string name;
  int count;
   count = 1;
   cout << "Enter a name→ ";
   cin >> name;
   while (name !="quit" && count <= 5)
     {
       count ++;
       cout << "Hello, " << name <<"\n";
       cout << "Enter another name or type 'quit' to exit→ ";
       cin >> name ;

     }
   count --;  //actual number of names entered is count minus 1.
   cout << count << " names were entered " << "\n";
   getchar();
   return (0);
}
```

```
Enter a name--> James
Hello, James
Enter another name or type 'quit' to exit--> Mary
Hello, Mary
Enter another name or type 'quit' to exit--> Rose
Hello, Rose
Enter another name or type 'quit' to exit--> quit
3 names were entered



Enter a name--> Jack
Hello, Jack
Enter another name or type 'quit' to exit--> Jill
Hello, Jill
Enter another name or type 'quit' to exit--> Roger
Hello, Roger
Enter another name or type 'quit' to exit--> Ed
Hello, Ed
Enter another name or type 'quit' to exit--> Sandy
Hello, Sandy
Enter another name or type 'quit' to exit--> Reg
5 names were entered
```

While loop is a pre-test loop. It tests for the condition before entering the loop. We will be discussing a loop structure that checks for the condition at the bottom of the loop. If you want to use a pre-test loop controlled by a counter as seen in Program 4-1, you could either use a while loop or a **for loop**. **For loop** is variation of while loop specifically designed for count controlled loop. Program 4-3 is a modification of Program 4-1; the while loop has been changed to a for loop.

Program 4-3

```
/****************************************
Display 1 to 100 on the screen
Teaching objective  - For loop
By Dr. John Abraham
Created for 1380 students
****************************************/

#include <iostream.h>

int main()
{
   int number;
```

```
   for (number =1; number <= 100; number++)
     {
      cout << number << "   ";       //display number and put some spaces

     }
   getchar();
   return (0);
}
```

Program Run 4-3

```
  1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16   17   18
 19   20   21   22   23   24   25   26   27   28   29   30   31   32   33   34
 35   36   37   38   39   40   41   42   43   44   45   46   47   48   49   50
 51   52   53   54   55   56   57   58   59   60   61   62   63   64   65   66
 67   68   69   70   71   72   73   74   75   76   77   78   79   80   81   82
 83   84   85   86   87   88   89   90   91   92   93   94   95   96   97   98
 99   100
```

As you can see, the Program Run 4-3 looks identical to Program Run 4-1. Let us understand this statement: for (number =1; number <= 100; number++). The values of the variable number are given, the initial value, loop execution condition, and the step value. In Program 4-3A, we will change all three values and see how the execution changes.

Program 4-3A

```
/*************************************************************
Display all even numbers beginning with 2 and ending with 100
Teaching objective  - For loop
By Dr. John Abraham
Created for 1380 students
*************************************************************/

#include <iostream.h>

int main()
{
   int number;
   for (number =2; number <= 100; number=number+2)
     {
      cout << number << "   ";       //display number and put some spaces

     }
   getchar();
   return (0);
}
```

Program Run 4-3A

```
  2   4   6   8   10   12   14   16   18   20   22   24   26   28   30   32   34
 36   38   40   42   44   46   48   50   52   54   56   58   60   62   64   66
 68   70   72   74   76   78   80   82   84   86   88   90   92   94   96   98
```

We could modify this program to display the number backwards. See Program 4-3B. Note how the initial variable, loop execution condition, and the step values are changed.

Program 4-3B

```
/*************************************************************
Display all even numbers backward beginning with 100 and ending with 2
Teaching objective  - For loop
By Dr. John Abraham
Created for 1380 students
*************************************************************/

#include <iostream.h>

int main()
{
  int number;
  for (number =100; number >= 1; number=number-2)
    {
     cout << number << "   ";       //display number and put some spaces

    }
  getchar();
  return (0);
}
```

Program Run 4-3B

```
100   98   96   94   92   90   88   86   84   82   80   78   76   74   72   70
 68   66   64   62   60   58   56   54   52   50   48   46   44   42   40   38
 36   34   32   30   28   26   24   22   20   18   16   14   12   10   8   6   4
  2
```

Suppose you want to find the average grades for 20 students in a class on a quiz. You could either use the while loop or the for loop. However, if you want to write this program for a more general situation in which the number of students vary in different classes, you will have to use a **sentinel controlled while loop**. The sentinel value could be –99 for the grade input. Program 4-4 illustrates this.

Program 4-4

```
/*************************************************************
```

```
Find average of a set of exam scores
Teaching objective  - Sentinel controlled while loop
By Dr. John Abraham
Created for 1380 students
**********************************************************/

#include <iostream.h>

int main()
{
  int grade, total, count, average; //count will keep track of number of grades
  count = 1;  // initialize both count and total
  total = 0;
  cout << "Enter a grade or -99 to quit--> ";
  cin >> grade;
  while (grade != -99)
    {
     total += grade;
     count ++;
     cout << "Enter a grade or -99 to quit--> ";
     cin >> grade;
    }
  count--;
  average = total/count;
  cout << "Sum of " << count << " grades entered ---> " << total << "\n";
  cout << "The average grade is ---> " << average;

  getchar();
  return (0);
}
```

Program Run 4-4

```
Enter a grade or -99 to quit--> 100
Enter a grade or -99 to quit--> 88
Enter a grade or -99 to quit--> 71
Enter a grade or -99 to quit--> 56
Enter a grade or -99 to quit--> 88
Enter a grade or -99 to quit--> 65
Enter a grade or -99 to quit--> -99
Sum of 6 grades entered ---> 468
The average grade is ---> 78
```

We keep a running total by initializing total to 0 and adding all grades to the previous total. We do not want to add the –99 to the total nor should it count as a valid grade. The loop continuation condition clearly states to exit the loop if a –99 entered, therefore –99

is not added to the total. However, the count was already incremented, which should be negated. This is what we do with the statement: count--. This is a very important concept, we will be using this quite a bit in many of the future programs.

The last looping structure I want to mention is the do..while loop. Do while loop is a post test loop; the condition is tested at the bottom of the loop. Therefore, a do while loop will be executed at least once. Program 4-5 is a modification of the first program in this chapter.

Program 4-4

```
/*****************************************
Display 1 to 100 on the screen
Teaching objective  - do while loop
By Dr. John Abraham
Created for 1380 students
*****************************************/

#include <iostream.h>

int main()
{
  int number;
  number = 1;
  do
    {
     cout << number << "   ";
     number ++;
    }
  while (number <=100);
  getchar();
  return (0);
}
```

Program run 4-4

```
1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16   17   18
 19   20   21   22   23   24   25   26   27   28   29   30   31   32   33   34
 35   36   37   38   39   40   41   42   43   44   45   46   47   48   49   50
 51   52   53   54   55   56   57   58   59   60   61   62   63   64   65   66
 67   68   69   70   71   72   73   74   75   76   77   78   79   80   81   82
 83   84   85   86   87   88   89   90   91   92   93   94   95   96   97   98
 99   100
```

Part 5 – Arrays

So far we have looked at simple data types such as integer, float, etc. If we want to write a program to find the standard deviation of a set of test scores we need to have a variable that could hold multiple integers instead of assigning an identifier for each score. An array is the answer to our need. An array is used to store and process a collection of data of the same type. Let me emphasize that an array can hold **multiple values of the same type**; it cannot hold values of different types. An array may also be called a list.

First let us talk about declaring an array. Any program that we might write should be general enough to apply to different situations; the program we are about to write should not only let us calculate the standard deviation for exam #1 but also all subsequent exams. Therefore, we have decide what is the maximum number of scores we will have. For now, we will choose 90. This is a limitation of arrays; we are limited to this number once we code it in. To change it we will have to change the code and recompile. That is why we call an array a **static list**. Back to declaring the array. To declare an array of 90 scores declare it like any other integer variable, but put a [90] in the square brackets: **int scores[90].**

Just because we declared an array for 90 elements, there is no need to enter all 90 scores. We can enter any number of scores up to 90. Let us write a program to accept some scores into an array (terminate the entry when a negative number is entered). We will have to keep track of the number scores entered. Program Six_1 uses a global variable for the array to keep the example simple. Program Six_2 shows passing the array as an argument. Please pay special attention to the fact that the n is decreased by one to get the actual number of valid scores entered. The last grade (negative score), although counted, is not a valid score.

Program Six_1.

```
/************************************************
Accept some scores into an array.
Terminate Entry when a negative number is entered.
Keep track of number of scores entered.
Teaching objective  - Data entry into an array.
By Dr. John Abraham
Created for 1380 students
************************************************/

#include <iostream.h>

int scores[90]; //array is declared globally for this example.
int getscores (void);

int main ()
{
        int n, i; // n for number of scores. i is a counter.
        n = getscores(); //go get the scores and how many
        for (i=1; i<=n; i++) cout<<scores[i]<<endl;  //show scores.
        Return(0);
}


int getscores()
{
   int n=1;
        cout << "ENTER A SCORE AND PRESS ENTER. YOU QUIT ANY TIME BY ENTERING A
NEG NUMBER!\n";
        cout << "Enter score# " << n << " ";
        cin >> scores[n];
        while (scores[n] >= 0)
        {
```

```
                    n++;
                    cout << "Enter score# " << n <<" ";
                    cin >> scores[n];
          }
return n-1; //n-1 actual scores read.
}
```

Program Run Six_1.

```
ENTER A SCORE AND PRESS ENTER. YOU QUIT ANY TIME BY ENTERING A NEG NUMBER!
Enter score# 1 78
Enter score# 2 91
Enter score# 3 88
Enter score# 4 75
Enter score# 5 60
Enter score# 6 88
Enter score# 7 59
Enter score# 8 99
Enter score# 9 78
Enter score# 10 -1
78
91
88
75
60
88
59
99
78
Press any key to continue
```

Passing array as a parameter is not all that difficult. First, include a prototype which indicates that the parameter passed is an array, without actually showing the number of elements in the array - just put []. The function heading also has similar declaration. The variable declaration however, should show the dimension of the array. Please compare Program Six_1 to Program Six_2 to see the differences.

Program Six_2.

```
/************************************************
Accept some scores into an array.
Terminate Entry when a negative number is entered.
Keep track of number of scores entered.
Teaching objective  - Pass Array as a parameter.
By Dr. John Abraham
Created for 1380 students
************************************************/

#include <iostream.h>

int getscores (int scores[]);      //prototype for passing an array
```

```
int main ()
{
        int scores[90];              //array of 90 integers named scores
        int n, i;                    // n for number of scores. i is a counter.
        n = getscores(scores);       //go get the scores and how many
        for (i=1; I<=n; i++) cout<<scores[i]<<endl;  //show scores.
        Return(0);
}


int getscores(int scores[])
{
   int n=1;
        cout << "ENTER A SCORE AND PRESS ENTER. YOU QUIT ANY TIME BY ENTERING A
NEG NUMBER!\n";
        cout << "Enter score# " << n << " ";
        cin >> scores[n];
        while (scores[n] >= 0)
        {
                n++;
                cout << "Enter score# " << n <<" ";
                cin >> scores[n];
        }
return n-1; //n-1 actual scores read.
}
```

Program Run Six-2.

```
ENTER A SCORE AND PRESS ENTER. YOU QUIT ANY TIME BY ENTERING A NEG NUMBER!
Enter score# 1 88
Enter score# 2 77
Enter score# 3 76
Enter score# 4 -1
88
77
76
Press any key to continue
```

It is important to note that the array parameter passed to the function is neither a **call-by-value**, or a **call-by-reference**, but a new kind of parameter known as an **array parameter**. It follows that when passing an array neither a copy of the entire array is passed nor every memory location of entire array are passed, instead the memory location of the first element of the array is passed. Therefore, it is not necessary to pass the size of the array, just include square brackets with nothing in it. Both the calling module and called module calculate the memory location of a particular element using abase and offset (calculated by index). More about addressing modes will be taught in future courses.

In developing this chapter I will use the standard deviation example. So let me explain what it is. In simple terms the standard deviation is a weighted average of differences of all the scores from the mean. In order to calculate it we follow the following steps:

1. Find the mean (average) of all the scores.
2. Find the difference of each score from the mean.

3. Square the differences (item #2).
4. Find the sum of all squared differences (item #3).
5. Find the variance by dividing the sum (item #4) by the number of scores minus one.
6. Find the square root of the variance (item #5).

1. Find the mean of the scores entered.

Program Six_3.

```
/************************************************
Accept some scores into an array.
Terminate Entry when a negative number is entered.
Keep track of number of scores entered.
Teaching objective  - Pass Array as a parameter.
By Dr. John Abraham
Created for 1380 students
************************************************/

#include <iostream.h>

int getscores (int scores[]);
float calcMean (int scores[], int);
void display(int, int scores[], float);

//mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
int main ()
{
        int scores[90];
        int n; // n for number of scores.
        float mean;
        n = getscores(scores); //go get the scores and how many
        mean = calcMean(scores, n);
        display(n,scores,mean);
        return(0);
}

//mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmm

int getscores(int scores[])
{
  int n=1;
        cout << "ENTER A SCORE AND PRESS ENTER. YOU QUIT ANY TIME BY
ENTERING A NEG NUMBER!\n";
        cout << "Enter score# " << n << " ";
        cin >> scores[n];
        while (scores[n] >= 0)
        {
                n++;
                cout << "Enter score# " << n <<" ";
                cin >> scores[n];
        }
return n-1; //n-1 actual scores read.
}

//fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
float calcMean(int scores[], int n)
        {
```

```
            int sum=0;
            int i;    //use for counter
            float m;  //local variable for mean
            for (i=1; i <=n; i++) sum += scores[i]; //add all scores
            m = float(float(sum)/n);
            return m;
           }

//ffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
void display (int n, int scores[], float mean)
           {
           int i;
           cout << n<< " Scores were entered. They are: \n";
           for (i=1; i<=n; i++) cout<<scores[i]<<endl;          //show scores.
           cout << "The mean is: "<<mean <<endl;
           }
```

Program Run Six_3.

```
ENTER A SCORE AND PRESS ENTER. YOU QUIT ANY TIME BY ENTERING A NEG
NUMBER!
Enter score# 1 88
Enter score# 2 77
Enter score# 3 83
Enter score# 4 85
Enter score# 5 96
Enter score# 6 68
Enter score# 7 -1
6 Scores were entered. They are:
88
77
83
85
96
68
The mean is: 82.8333
Press any key to continue
```

Standard deviation program continued.

2. **Find difference of each score from the mean.**
3. **Square the differences.**
4. **Add the squared differences.**
5. **Find the variance by dividing the sum by the number of scores minus one.**
6. **Find the square root of the variance.**

Program Six_4.

```
/***************************************************
Accept some scores into an array.
Terminate Entry when a negative number is entered.
Keep track of number of scores entered.
Find difference of each score from the Mean.
```

```
Square the differences and add the squares.
Teaching objective  - Pass Array as a parameter.
By Dr. John Abraham
Created for 1380 students
**************************************************/

#include <iostream.h>
#include <iomanip.h>
#include <math.h> //for square root function.

int getscores (int scores[]);
float calcMean (int scores[], int);
float sumSquares (int, int scores[], float diff[], float diffSq[],float);
float stdDeviation(int, float);
void Table(int, int scores[], float diff[], float diffSq[], float, float);
//mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
int main ()
{
        int scores[90];float diff[90], diffSq[90];
        int n; // n for number of scores.
        float mean, sumSq, stDev;
        n = getscores(scores); //go get the scores and how many
        mean = calcMean(scores, n);
        sumSq = sumSquares (n,scores,diff,diffSq,mean);
        stDev = stdDeviation(n, sumSq);
        Table(n, scores, diff,diffSq, mean, stDev);
        return(0);

}

//mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmm

int getscores(int scores[])
{
   int n=1;
        cout << "ENTER A SCORE AND PRESS ENTER. YOU QUIT ANY TIME BY
ENTERING A NEG NUMBER!\n";
        cout << "Enter score# " << n << " ";
        cin >> scores[n];
        while (scores[n] >= 0)
        {
                n++;
                cout << "Enter score# " << n <<" ";
                cin >> scores[n];
        }
return n-1; //n-1 actual scores read.
}

//fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
float calcMean(int scores[], int n)
        {
          int sum=0;
          int i;    //use for counter
          float m;  //local variable for mean
          for (i=1; i <=n; i++) sum += scores[i]; //add all scores
```

```
            m = float(float(sum)/n);
            return m;
          }


//ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

float sumSquares (int n, int scores[], float diff[],
                                        float diffSq[],float mean)

          {
                    int i; float ss=0.0;
                    for (i=1; i <=n; i++)
                    {
                              diff[i] = scores[i]-mean;
                              diffSq[i] = diff[i] * diff[i];
                              ss += diffSq[i];
                    }
          return ss;
          }

//ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
float stdDeviation(int n, float sumSq)

{
          float variance;
          variance = sumSq/(n-1);
          return (float(sqrt(variance)));
}

//ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

void Table(int n, int scores[], float diff[],
                      float diffSq[], float mean, float stDev)
{
          int i; float ss=0;
cout << "\nStadard Deviation for " << n<<" Scores.  Mean: "<<mean <<endl;
cout <<"-------------------------------------------------------\n";
cout <<setw(10) <<"Score"<<setw(10) <<"Dev" <<setw(10) << "Dev Sq"
            <<"  Sum of dev2)\n";
cout <<"-------------------------------------------------------\n";
cout << setiosflags(ios::fixed);

for (i = 1; i<=n; i++)
{
                    ss += diffSq[i];
                    cout << setw(10) << setprecision(2) << scores[i]
                              << setw(10) << setprecision(2) << diff[i]
                              << setw(10) << setprecision(2) << diffSq[i]
                              << setw(10) << setprecision(2) << ss << endl;

          }
cout <<"-------------------------------------------------------\n";
cout <<"Standard Deviation "<< setprecision(2) <<stDev <<endl;
}
```

Program Run Six-4.

```
ENTER A SCORE AND PRESS ENTER. YOU QUIT ANY TIME BY ENTERING A NEG
NUMBER!
Enter score# 1 88
Enter score# 2 79
Enter score# 3 75
Enter score# 4 96
Enter score# 5 84
Enter score# 6 92
Enter score# 7 77
Enter score# 8 71
Enter score# 9 94
Enter score# 10 88
Enter score# 11 82
Enter score# 12 -1

Stadard Deviation for 11 Scores.  Mean: 84.1818
-------------------------------------------------------------------------------
          Score        Dev        Dev Sq     Sum of dev2)
-------------------------------------------------------------------------------
             88       3.82      14.58       14.58
             79      -5.18      26.85       41.43
             75      -9.18      84.31      125.74
             96      11.82     139.67      265.40
             84      -0.18       0.03      265.44
             92       7.82      61.12      326.56
             77      -7.18      51.58      378.14
             71     -13.18     173.76      551.90
             94       9.82      96.40      648.30
             88       3.82      14.58      662.88
             82      -2.18       4.76      667.64
-------------------------------------------------------------------------------
Standard Deviation 8.17
```