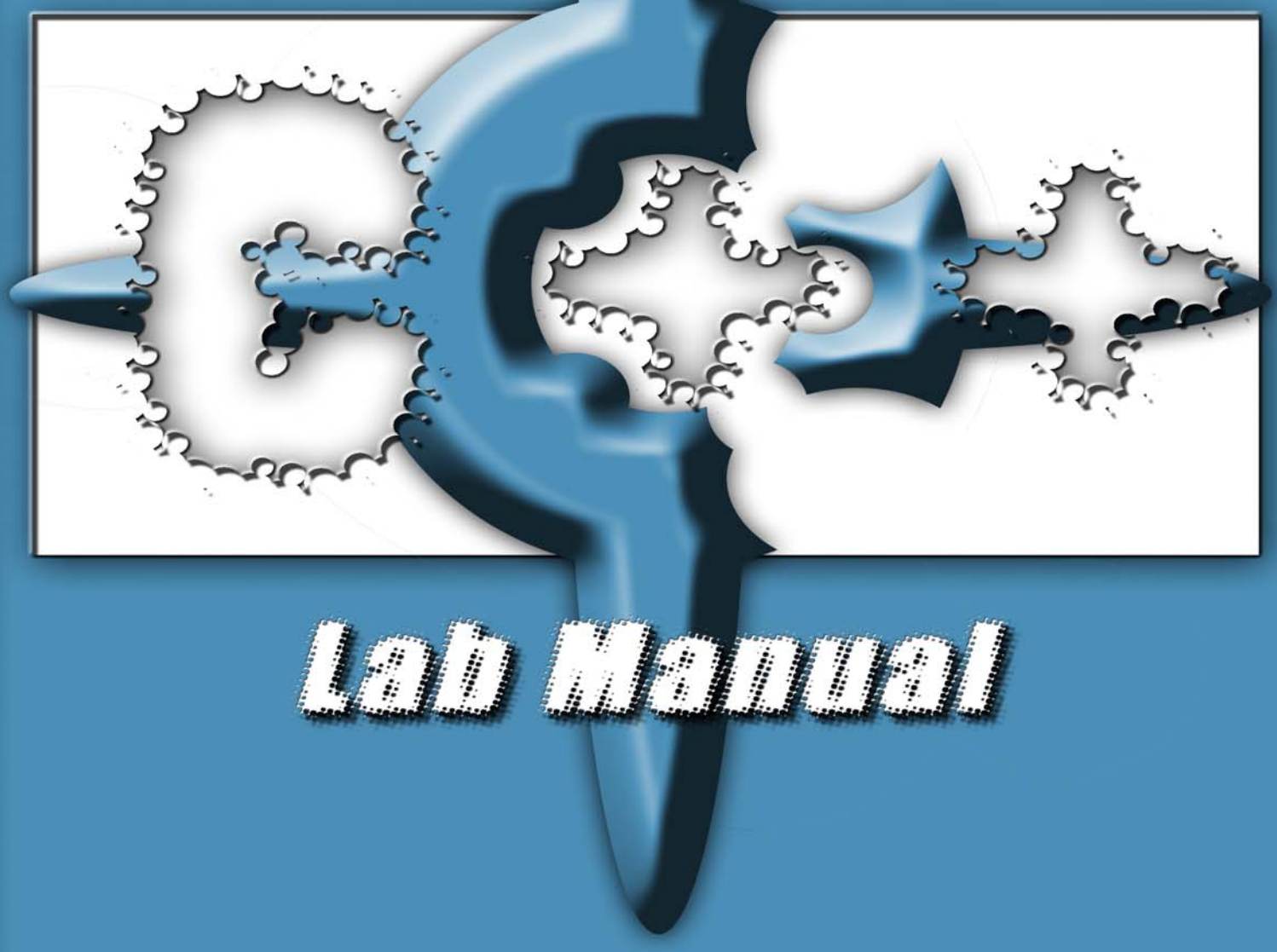
CSE - 101 L

Introduction to Computing Lab



Lab Manual

PREFACE

Ghulam Ishaq Khan Institute of Engineering Sciences & technology has had been for the past decade, a university of high repute and has maintained its high standards during this period. Faculty of Computer Science and Engineering has produced excellent engineers, which have proved their credibility the world over. Through the four years of studies that I myself undertook in this institute, I, along with my fellow students have felt a need of lab manuals for each and every lab that we perform during our degree program. When I myself joined here as a Lab Engineer, I found it hard to follow a set lab pattern or lab course because my predecessors had taught the labs according to their own liking. This although helps the instructor to mould the course according to his/her liking but, at the expense of the amount of information that they fail to deliver. Apart from this, a large number of students are always finding a source of information which they can follow to be better prepared for the labs in the coming weeks. The Labs were previously handed out to students only on the lab day which resulted in a lot of waste of time, paper and money. This forced me to work on a lab manual that the faculty could easily print and distribute to every student so that the above mentioned problems do not occur. This Lab manual is important for the people who are beginners in the field of programming as it provides them with the precise information that they need to get themselves acquainted with computing. Some of the labs present in this manual are actually a simplified version of the labs made by Mr. Ahmed Bilal who in the past has worked assiduously to bring the standard of programming much higher. This manual not only makes a person proficient in C++ programming, but it can also help him/her achieve a high standard in a very short amount of time hence, it is ideal for conducting labs in one semester. Since this manual focuses on C++ programming only, thus, it does not contain the introductory labs for introducing the students to computer's environment.

INDEX

ABOUT THE AUTHOR

Umair Azfar Khan, Registration Number 980153 (GIK Institute) is a former student and employee of Ghulam Ishaq Khan Institute of Engineering Science and Technology. He was a part of the Eighth batch that graduated from this institute and later became the part of the faulty for a period of ten months or two semesters as a Teaching Assistant. He did his majors in Computer Systems Engineering and graduated on May 29, 2002. He joined GIK Institute on 22nd August, 2002 and served as a Teaching Assistant there till 1st June, 2003. He has always been very ambitious about his work and expects the same from his colleagues and students. His Senior Design Project in distributed programming is an example of this zest. Apart from his studies, he takes immense interest in 3D graphics and is always working on one of his self assigned projects. His future aims contain the ambition of achieving a doctorate degree and serve his motherland by providing it with the knowledge that he himself has gained or will gain in the coming future.

CSE 101 – Introduction to Computers & Programming
Programming Lab # 1
Basics of C/C++ Programming

Introduction of the language called C++

C++ Language is a superset of the programming language C whose first name was "C with Classes". The reason that the C language is called C is simply because it is a successor to the language called B which in itself was a successor to a language called BCPL. C++ is called C plus plus for the reason that it provides more extension to the already known language C. C++ was developed by Bjarne Stroustrup in 1983.

Basics of C/C++ Programming

C++ is a high level language with certain low-level features as well. Now is the time to explore basic structure of C++ program and to learn a few fundamental concepts in C++ programming. Remember that C++ is a **case-sensitive** language. When we talk about programming, we mean that how we want the computer to perform certain operations. Computer responds to a double click on “My Computer” icon by opening a window of “My Computer”. This is because it was programmed that way. A C++ program is actually a collection of statements and data on which various operations can be performed. Kindly refer to *listing 1.1* to see what a simple C++ program looks like.

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

void main( )
{
        //This program shows text
        clrscr( );
        cout<<”Great power gives great responsibilities\n”;
        getch();
}
```

Listing 1.1

In this code the first to lines are “include statements”. In C++ we will be using many pre-defined program modules (or functions). Definitions of these functions are present special kind of files, called header files. Include statements cause those definitions to be included in your program so that you may use those particular functions. For example in this program, we are using functions **clrscr()** and **getch()**. Definitions of these two functions are given in the header file conio.h. So, to use these functions, first we have to include the file **conio.h** in our program.

Next thing is **main()**, which is a user defined function. This is the function from where every C++ program starts execution. The brackets “**( )**” indicate to the compiler that "main" is a function and as the brackets are empty, we are not giving any input to the program. Opening and closing “**{ }**” brackets indicate the start and end of the program block. We will talk more about functions and the significance of keyword “**void**” in a later lab. For now, let us get on with the statements in this C++ program.

The first line in the **main** function is a comment. In this case, anything following a “**//**” denotes a comment in C++. This is used for documentation of the program. The second line is a function **clrscr()** which clears the screen. This function is very much like the **cls** command in DOS. Note that a **semicolon (;)** terminates each statement in C++. It is similar to the full-stop that we use in English, but in C++, a full-stop has other function, so the developers of this language used “**;**” for this purpose.

Next statement in this program uses something called an output stream to display the string "**Great power gives great responsibilities**" on the screen. Here "**<<**" is the stream insertion operator. Note that we have to use inverted commas in C++ to specify a string just like we use inverted commas in English language to specify what a person is saying. Here "**\n**" is used to insert a new-line character, which means, everything following this text will appear on a new line.

Finally, we have a statement calling **getch()** function. This function makes our C++ program get a character from keyboard before quiting. We are using it to make sure that we see output of our program on the screen, and then pressing any key to quit.

Data Types

Following are the few basic data types used in C++.

Data Type	Keyword Used	Example	Declarations
Integer (Numbers)	int	21	int a; int b = 21;
Floating Point (Decimal Numbers)	float	1.56	float a; float b = 1.56;
Character (Characters)	char	z	char a; char a ='z';

Now consider this sample program for declaring some and printing some data in C++:

```
//**********************A Value Swapping Program*********************

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

void main( )
{
        clrscr();
        int a;              //declaring interger variable called a
        int b, c;           //another way of declaring integers

        cout<<"\nEnter the value of a="; //prompting the user to enter an integer
        cin>>a;                          //reading in that integer value into the variable called a
        cout<<"\nEnter the value of b=";
        cin>>b;

        cout<<"value of a & b before swapping\n"
        cout<<a<<"\n";
        cout<<b<<"\n";

        c = a;
        a = b;
        b = c;

        cout<<"value of a & b after swapping\n"
        cout<<"\n"<<a<<"\n";
        cout<<"\n"<<b<<"\n";
}
```

listing 1.2

This program swaps the values of both variable a and variable b, using the temporary variable c.

Relational Operators

C++ also uses some relational operators to perform comparison of different values. Some of these are:

Operation	Operator
Equal to	==
Not equal to	!=
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=

Now consider the following example:

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

void main( )
{
        clrscr();
        int a,b;
        cin>>a>>b;
        if (a==b)
        {
                cout<<"\na is equal to b\n";
        }
        else if (a>b)
        {
                cout<<"a is greater than b\n";
        }
        else
        {
                cout<<"b is greater than a\n";
        }
        getch();
}
```

Listing 1.3

Arithmatic Operations

We can perform various arithmetic operations on the data we declare in C++. Arithmetic operators used in C++ are:

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

The following program shows how these operators work:

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

void main( )
{
        clrscr();
        int a,b,c;
        cin>>a>>b;
        cout<<"\nThe result of addition is=";
        c=a+b;
        cout<<c;
        cout<<"\nThe result of subtraction is=";
        c=a-b;
        cout<<c;
        cout<<"\nThe result of multiplication is=";
        c=a*b;
        cout<<c;
        cout<<"\nThe result of division is=";
        c=a/b;
        cout<<c;
        cout<<"\nThe remainder is=";
        c=a%b;
        cout<<c;
}
```

listing 1.4

Lab Exercises

Exercise 1: Write a program in which a user gives an integer as an input and the program finds out if it is even or odd.

Exercise 2: Write a program in which a user gives an integer as an input, which this program uses to find the area of a circle.

Exercise 3: Write a program which takes as an input the age of a person and his/her name and gives as an output like this "The age of Gandalf is = 63 years". The program shall also tell the user if the person is a child, a teenager, an adult or an old person. The program shall not accept values below 0 and above 100.

Lab Assignment # 1

Swap two integers namely "*alpha*" and "*beta*" and having the value 10 and 20 respectively, without using a third integer. It should be properly commented.

CSE 101 – Introduction to Computers & Programming
Programming Lab # 2
Data Types in C++

Basic Data Types

Name	C++ Syntax	Describes
Integer	int	numeric data in the range of -32768 to 32768
Floating-Point	float	floating-point numeric data in the range $8.43x10^{-37}$ to $3.37x10^{38}$
Double	double	floating-point numeric data in the range $2.225x10^{-308}$ to $1.7976x10^{308}$
Character	char	character specified by character codes -128 to 127
Boolean	bool	has only two values, either true (1) or false (0)
Void	void	A non existent value

Type Qualifiers

Name	C++ Syntax	Description
Long Form	long	It requests a long form of an item. Can be used with both int and double
Short Form	short	It requests a short form of an item. Can be only used with int and not double
Signed Number	signed	It describes a variable from its maximum negative to its maximum positive value
Unsigned Number	unsigned	It describes a variable from 0 to a maximum positive value. Valid only with int and char data types
Constant Value	const	It describes value of a variable to be unchangeable

To use type qualifiers, consider the example given in **listing 2.1**:

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

void main( )
{
        unsigned int positive;          //Unsigned integer
        unsigned long int longest;      //Longest positive integer that C++ can handle
        const int  constant = 20;

        clrscr( );

        cin>>positive;
        cin>>longest;
        cout<<"\n"<<positive<<" "<<longest<<" "<<constant;
        getch();
}
```

Listing 2.1

Type Casting

Sometimes, we need to change data type of a variable from one type to another. Most common of this thing can be its use in doing calculations in integers, and getting results in floating-point numbers.

For example, to perform a division operation on two integers, and to get result as a floating-point number, we would use the following piece of code as given in **listing 2.2**:

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

void main( )
{
        clrscr( );
        int num1, num2;
        float result;
        cin>>num1>>num2;
        result=(float)num1/num2;
        getch();
}
```

Listing 2.2

Operators

C++ has a variety of operators to perform various tasks. You came across a few in the previous lab and a few new operators will be discussed in this lab.

Logical Operators

Logical Operators are used to perform logical operations on data. These operators are typically useful to see whether certain conditions are satisfied or not. Logical Operations used in C++ are:

And (&&)

This operator is used to evaluate an expression for logical AND operation. The truth table on the right explains what a logical And (&&) really means.

x	y	Ans
1	1	1
1	0	0
0	1	0
0	0	0

Example:

```
If (a>b && a>c)
{
        cout<<"a is greater than both b & c";
}
```

Or (||)

This operator is used to evaluate an expression for logical OR operation. The truth table on the right explains what a logical And (&&) really means.

x	y	Ans
1	1	1
1	0	1
0	1	1
0	0	0

Example:

```
If (a>b || c>b)
{
        cout<<"Either a or c or both are greater than b";
}
```

Negation (!)

This operator is used to evaluate an expression for logical negation operation. Also, there is an operator for the condition "Not equal to" (!=). An example can be:

```
If (!(a>b) && c!=b)
{
        cout<<"here a is not greater than b, and c is not equal to b";
}
```

Increment & Decrement (++ and --)

These operators are used to increment or decrement value of a variable. For Example:

```
a++;     //This is same as a=a+1;
a - -;   //This is same as a=a -1;
```

Assignment (=) and Compound Assignemnts (operator=)

```
A = 1;   //Simple assignment operator
```

If a mathematical operator is used in conjunction with the assignment operator we can make the code better.

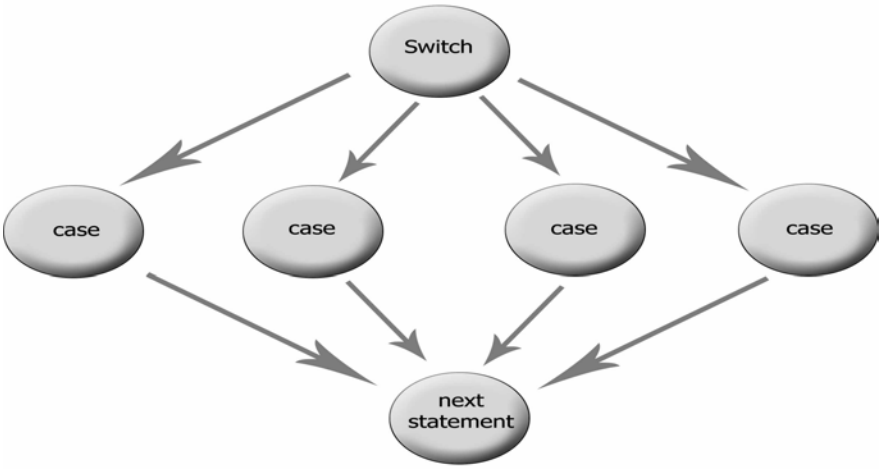
```
A += 1; //Same as A=A+1;
```

Flow Control

We studied flow control in the last lab while using IF ELSE statement. Today we are going to practice with a new type of flow control called:

SWITCH CASE Statement

The flow graph of switch case statement is like this:



The following example in listing 2.3 illustrates the use of the switch case statement:

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

void main( )
{
        clrscr( );
        int num1, num2, option;
        cout<<"Enter first number=";
        cin>>num1;
        cout<<"\nEnter second number=";
        cin>>num2;
        clrscr();
        cout<<"Enter a choice from below:\n"
                "1.     Add\n"
                "2.     Subtract\n"
                "3.     Divide\n"
                "4.     Multiply\n";
        cin>>option;

        switch(option)
        {
                case 1:
                        cout<<num1+num2;
                        break;
                case 2:
                        cout<<num1-num2;
                        break;
                case 3:
                        cout<<num1/num2;
                        break;
                case 4:
                        cout<<num1*num2;
                        break;
                default:
                        cout<<"Invalid Option";
        }
getch();
}
```

Listing 2.3

Exercise:
Write a menu driven program, in which a user enters the length of a line and then is prompted to choose from the following options: 1. Area of Square, 2. Area of a Circle, 3. Volume of a Sphere, 4. Volume of a cylinder, 5. exit. If a user enters the Area of circle, the Volume of the sphere should also be shown. If the volume of sphere is greater than 100, it should say,"small sphere" otherwise, "big sphere". If the volume of cylinder is less than 20, it should say, "glass". If its length is greater than 20 and less than 100, it should say,"a jug" otherwise, a tank. (if I forget, remind me of a while loop)

Assignment:
Write a menu driven program, which helps you take care of your virtual cat. The cat should have a name and age. You should be able to do the following actions: 1. Feed it, 2. Pat it and 3. Play with it. There should be levels that keep track of how hungry your cat is, how loving it is and how playful it is. When the user chooses to feed it, it should grow fat, but it should then become less playful and less loving and the same goes for the other two options. When the user wishes to quit, he should be told about the state of his/her cat before quiting. (should be commented).

Arrays

An array is like a list or table of any data type. We use arrays for a variety of programming tasks especially when we have to make a list of the same type of data.

UNIDIMENSIONAL ARRAYS

An array with a single dimension is like a list. That is how we define such arrays:

int list[10]; //Defining a list of 10 integers

This statement actually means that we are declaring a list of arrays from 0 to 9. This means that the starting array element will be referred to as **list[0]** and the last element will be referred to as **list[9]**.

Elements of this array can be referenced as:

```
list[2]=20;
list[5]=30;

cout<<list[2]<<"\n";
cout<<list[5];
```

Elements of an array can be initialized at the time of its declaration.

```
int list[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };    //Defining an array of ten integers
```

MULTIDIMENSIONAL ARRAYS

An array can also be multidimensional. To define a multidimensional array, we follow a similar approach:

```
int list[3][3];                //Define a 3x3 matrix or table
```

To reference various elements of this array, similar approach is used. For example:

```
list[1][2] = 34;               //Assigning 34 to row 1 and column 2
list[0][2] = 20;               //Assigning 20 to row 0 and column 2
```

Like the one-dimensional array, we can also initialize a multidimensional array like:

```
int list[3 ] [3 ] = { { 1, 2, 3} , {4, 5, 6} , {7, 8, 9} };    //Initializing a 3x3 matrix
```

You can write the same thing more elegantly as:

```
int list[3][3] = {  { 1, 2, 3 },
                    { 4, 5, 6 },
                    { 7, 8, 9 }  };
```

1	2	3
4	5	6
7	8	9

The initialized matrix will be like this

Strings

A text string in C++ is nothing but a one-dimensional array of characters terminated by a special character '\0'. Lets see what we can do with string in C++;

DEFINING A STRING

A string can be defined in a similar fashion as an array. That's how we define a string:

```
char str[30];       //Defining a string of 30 characters.
```

Similarly to access individual characters of a string, we use the following syntax:

```
str[0] = 'U';
str[1] = 'm';
str[2] = 'a';

cin>>str[3];
cout<<str[3];
```

We can also initialize a string while declaring it. This is how we do it:

```
char str[] = { 'a', 'b', 'c',  '\0' };          //Defining a string initialized to "abc"
```

making it more simple by:

```
char str[] = "abc";                             //Defining a string initialized to "abc"
```

To help us in string manipulation, we include two more header files:

string.h
stdio.h

STRING I/O

To get a value of the string from the user or to print a string on the monitor, it is not advisable to use cin>> and cout<< as was done previously. For this purpose, we use the functions defined in stdio.h

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

void main()
{
        clrscr();
        char myname[20];            //Define a string
        gets(myname);               //Get text for string from user
        puts (myname);              ??Display text of string on screen
        getch();
}
```

Listing 3.1

SOME INTERESTING OPERATIONS ON STRINGS

Following are a few interesting string manipulation functions defined in string.h. These functions make the life of a C++ programmer a lot easier and are a part of the standard C++ library.

To assign some text to a string, we can use the following function:
strcpy (char dest[], char source[])
Similarly, to concatenate two strings we use the following function:
strcat (char str1[], char str2[])
To compare two strings, we use the function:
strcmp (char str1[], char str2[])
If this function returns 0, that means both strings are same.
find the length of the string, we use the function:
strlen(str[]);

Now consider the following example to understand how these functions can be used to manipulate strings.

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

void main()
{
        char str1[20], str2[20];
        int length;
        clrscr;
        cout<<"Enter text for string 1:";
        gets(str1);
        cout<<"\nEnter text for string 2:";
        gets(str2);
        if(strcmp(str1,str2)==0)
        {
                cout<<"\nStrings are identical\n";
        }
        else
        {
                cout<<"\nStrings are not identical\n";
        }
        strcpy(str1,"Pink Floyd");          //Write some text to String 1
        puts(str1);                         //Display String 1
        strcat(str1,str2);                  //Concatenate String 1 and 2
        puts(str1);                         //Display String 1
        strcpy(str1,str2);                  //String 1 = String 2
        puts(str1);                         //Display String 1
        length = strlen(str1);              //get length of string 1
        cout<<length;                       //Print length
        getch();
}
```

Listing 3.2

EXERCISES:

Write a program to add, subtract and multiply a 2x2 matrix with another 2x2 matrix and storing their result in yet another 2x2 matrix. All the matrices should be identity matrices except the one storing the results.

ASSIGNMENT:

Write a program that has a default text string. The user enters three text strings which are concatenated by the program to form a single string and then are compared with the default string to see if the string is valid or not.

PRACTICE EXERCISE:

Write a program that asks the user to enter a password up to a certain number of characters, say 6 characters. If the password is wrong, it asks the user to re-enter the password, otherwise allows access and asks the user if he/she wants to change the password. If he/she wishes to change the password, he/she should be asked to enter the password twice to confirm it. Again, the new password should be constrained in some characters' number limit. In case the entry is wrong the second time, the original password should be retained. This exercise is a tough one. You will be required to use almost all of the functions provided by *string.h* header file plus a large number of nested IF – Else Statements and of course, a lot of thinking. It will be better on your part to write down the whole logic of the program on pen and paper before starting to code.

CSE 101 – Introduction to Computers & Programming
Programming Lab # 4
Repetetive Structures

This lab introduces a very important element of programming languages – repetitive structures or loops. These structures enable you to repeat the same lines of code over and over again and generally help in running a program continuously.

Random Numbers

While experimenting with various elements of programming language, its always helpful to know how to generate random numbers because, random numbers can provide us with sample data, which can be used to practice various programming language concepts.

There are two functions used to initialize and use random numbers. These functions are defined in the file stdlib.h. The function **randomize()** initializes random number generation, and the function **rand()** returns a random integer.

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

void main()
{
        int i;
        randomize();
        i = rand();
        cout<< i <<endl;
}
```

Listing 4.1

WHILE LOOP

General syntax of *while loop* is **while (expression) statement(s)**
The following program uses a while loop to print a string taken from a user as input. In this program the while loop used performs same operation as puts and strlen functions.

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

void main()
{
        char name[50]; int i = 0;
        clrscr();
        gets(name);
        while(name[i] != '\0')
        {
                cout<<name[i]<<" ";
                i++;
        }
        cout<<endl<<"Length of the string is:"<<i<<endl;
        getch()
}
```

Listing 4.2

DO – WHILE LOOP

General syntax of *do – while loop* is **do statement(s) while (expression)**.
Following example illustrates use of do – while loop to prompt user before quitting a program.

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

void main()
{
        clrscr();
        char chr;
        do {
                cout<<endl<<"Hello"<<endl;
                cout<<"Do you want to quit?";
                chr = getch();
        } while(chr != 'y' && chr != 'Y');
}
```

Listing 4.3

FOR LOOP

General syntax of *for loop* is: **for (expression_1; expression_2; expression_3) statement(s)**.
Following example uses a for loop to print integers from 0 to 9 on the screen:

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

void main()
{
        int i;
        clrscr();
        for(i = 0; i < 10; i++)
        {
                cout<<i<<endl;
        }
}
getch();
```

Listing 4.4

A more interesting example can be to use *for loop* to fill a 10x10 matrix (two – dimensional array) with random numbers. This example uses nested for loops.

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

void main()
{
        int mat[10][10], i, ,j;
        clrscr();
        randomize();
        for(i=0; i<10; i++) {
```

```
                for(j=0;j<10;j++) {
                        mat[i][j] = rand();
                        cout<<"Row:"<<i<<" Col:"<<j<<"="<<mat[i][j]<<" ";
                }
        }
        getch();
}
```

Listing 4.5

INTERRUPTING A LOOP

We can also use two special keywords of C++ to interrupt flow of a loop. The following program prints even numbers between 0 and 100. The break keyword works the same way as in switch – case statement, whereas continue interrupts execution of the loop and goes ahead with the next iteration.

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

void main()
{
        int i=0;
        clrscr();
        while(1) {
                i++;
                if( i%2 == 0){
                        cout<<i<<endl;
                }
                else {
                        continue;
                }
                if ( I == 100) {
                        break;
                }
        }
}
```

Listing 4.6

EXERCISES:

1. Using *while* or *do* – *while* loop, write a program to calculate number of occurrences of a specific character inside a string. This character and the string should be taken as input from the user.
2. Using nested *for* loops, write a program to fill two 10x10 matrices with random numbers and add these two matrices together.

CSE 101 – Introduction to Computers & Programming
Programming Lab # 5
Functions, Variable Scope and Debugging

A function is a basic programming unit of a C++ program. Every C++ program should have at least one function called main. In this lab, we will get to know about function declarations (prototypes), function definitions, and rest of the stuff related to functions. We will also take a look at scope of variables, and will learn how to use debugger.

Functions make decomposition of problem into various modules very easy. Let's take a few examples of functions and examine their use in C++ programs. Following example declares a simple function named PrintHello, which doesn't accept or return any data. This function just displays a string, "Hello World" on the screen.

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

void PrintHello( );                    //Function Prototype

void main( )
{
        PrintHello( );
}

void PrintHello( )                     //Function Definition and Body
{
        cout<<"Hello World!";
        getch( );
}
```

Listing 5.1

Following example declares a function named AddThem, which accepts two integers as input and prints their sum.

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

void AddThem( int, int);               //Function Prototype

void main( )
{
        int num1 = 3, num2 = 4;
        AddThem( num1, num2);
}

void AddThem( int n1, int n2 )         //Function Definition and Body
{
        cout<<n1 + n2;
}
```

Listing 5.2

In this example, function AddThem accepts two parameters, which are integers. In this case, the parameters declared as n1 and n2 are called *formal parameters* and are used as slots to send information from calling function to the function being called. Here num1 and num2 (arguments) are called *actual parameters* because these arguments denote the actual data that is passed to this function.

Following example declares another function named AddThem, which accepts two integers and returns their sum as an integer. Note that we are using the keyword "*return*" here:

```
#include <iostream.h>

int AddThem( int, int);                    //Function Prototype

void main( )
{
        int num1 = 3, num2 = 4,num;
        num3 = AddThem( 1, 2);             //Passing the numerical values 1 and 2
        cout<<num3<<endl;
        num3 = AddThem( num1, num2);
        cout<<num3<<endl;
}

int AddThem( int n1, int n2 )              //Function Definition and Body
{
        int n3;
        n3 = n1+ n2;
        return n3;                         //returning the resultant value of the addition
}
```

Listing 5.3

We can also pass an array to a function. Following example passes an array filled with random numbers to a function named MulArray. This function calculates product of elements of an array. First parameter specifies number of elements of this array:

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

int MulArray( int, int[ ] );               //Function Declaration

void main( )
{
        int list[100], prod,count;
        randomize( );
        for (count=0;count<100;count++)
        {
                list[count] = rand( )/1000;
        }
        prod = MulArray ( 100, list )
        cout<<prod<<endl;
}

int MulArray( int n, int array[ ] )        //Function Definition and Body
{
        int i, answer=1;
        for ( i = 0; i<n;i++)
        {
                answer *= array[i];
        }
        return answer;                     //returning the resultant value of the addition
}
```

Listing 5.4

Next example uses a very useful, but complex programming element, called recursion. Recursion is a mechanism when a function calls itself. Following is a classical example of recursion, that is, its use to calculate factorials:

```
#include <iostream.h>

unsigned int factorial ( int );                    //Function Prototype

void main( )
{
        unsigned int f=0;
        f = factorial (6)
        cout<< f <<endl;
}

unsigned int factorial( int n )                    //Function Definition and Body
{
        if ( n == 1 )
        {
                return 1;                          //Stopping condition
        }
        n = n * factorial ( n-1 );
        return n;
}
```

Listing 5.5

This program takes into consideration the fact that factorial of 1 is 1 and that a factorial of a number n is equal to factorial of n – 1. A properly made recursive function can make a life of a programmer easy.

Variable Scope:

The scope of a variable is the extent to which this variable is visible in a program. There can be two types of variables in this respect: *local variables* and *global variables*. Global variables are visible to the entire program, while local variables are visible only to the function they are defined in, for example:

```
#include <iostream.h>

const float PI = 3.14;                  //A global variable

float circArea (float);

void main( )
{
        float area;                     //A local variable of the main function
        area = circArea(21.5);
        cout<<area<<endl;
}

float circArea ( float rad )            //Function Definition and Body
{
        float area;                     //A local variable of the circArea function
        Area = PI * rad * rad;
        return Area;
}
```

Listing 5.6

In this program, PI is a global variable, visible to the entire program, where as all other variables are only visible to the functions they are defined in. Also note that the variable area in main is different from the variable area in circArea although the two variables have the same name.

Debugging a Program

Using a debugger is a very useful way to find logical errors inside a program, It allows a programmer to run a program one statement at a time, and to view values of variables after executing each statement. For a statement which calls a function, debugger can either execute that statement entirely, or step into code of that function for more detailed debugging. Function key F7 is used to step into a function and function key F8 is used to step over (or execute completely) a function. Instead of using Ctrl + F9, if you press F7 or F8, Borland C++ will compile and link the program and start debugger. To view value of a variable at a particular stage of execution, you can use Ctrl + F7 to add a variable name to the list of watches. And as you step through various statements of your program, you will see values of these variables in the watch window after every statement. You can also use breakpoints inside a program. While editing code, if you press Ctrl + F8 on a statement, it will be highlighted. This means, whenever you compile and run this program, this program will stop and invoke the debugger when it reaches this breakpoint. To remove a breakpoint, move cursor to that statement again press Ctrl + F8.

Lab Exercise:

Make a calculator that takes in a number, shows a menu thus asking the user as to what action does he/she want to take. The actions being:

+	add
-	subtract
*	multiply
/	divide
c	clear
e	exit

Once the user enters the first four actions, it asks the user to enter the second number. Even if the user presses the '=' sign instead of '+' and '8' instead of '*', it should work. Upon entering the second number, it performs the said function and shows you the answer. The calculator continues all the above mentioned operations to the result obtained until the user presses c to clear. Each operation should be performed in its own function which takes in the first number as an argument and returns the result of the calculation. All the functions should be declared in a header file named cal.h and declare it as **#include "cal.h"**. Make sure that both the cpp file and the header file are in the same directory.

Assignment:

No assignments from this day on meaning a life with fewer worries. From now on, your lab exercises will be marked both for marks and attendance, so show me the exercises only when you believe that they are perfect.

CSE 101 – Introduction to Computers & Programming
Programming Lab # 6
Handling of Records and Files

File handling is a very important aspect of C/C++ programming because; it enables the programmer to save data to permanent storage, and retrieves it later when needed.

A file is used to store some data on some permanent storage device like a floppy disk or a hard disk. There are four basic operations that you can perform on a file: -

1. Opening a file
2. Reading from a file
3. Writing to a file
4. Closing a file

To open a file, a function called **open** is used. This function accepts two arguments. First one is a file name and second one is a list of various options separated by symbol **|** . This function returns an integer called the handle. All other file-handling functions use this integer to identify a file. If this function returns -1, that means there is some error in opening this file. To close a file, a function called **close** is used. This function accepts file handle as an argument.

READING FROM A FILE

To read from a file, a function called **read** is used. This function accepts three arguments. First one is the file handle, second one is the address of variable to write to (&c in following listing means address of c). You can also use an array instead. And the third argument specifies size of data to read. For that a built-in function of C++ language, called **sizeof** can also be used. Following listing reads and displays contents of a file. Note that we are using O_RONLY as an argument of open function to specify that we want the file to be opened as read only.

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <fcntl.h>

void main()
{
        char filename[255], c;
        int handle;
        clrscr();

        cout<<"Enter name of file:";
        gets(filename);                         //Get file name from the user

        handle = open (filename, O_RDONLY);     //The return value of the 'open' function is stored in 'handle'

        if (handle == -1)                       //Checking if the handle returned is an error
        {
                cout<<"Could not open file!" \n";
                getch();
                return;
        }

        while( !eof (handle) )                  //Till the 'end of file' is not reached, continue
        {
                read ( handle, &c, sizeof(c));
                cout<<c;
        }
        close (handle);                         //Close file
        getch();
}
```

Listing 6.1

WRITING TO A FILE

To write a file, a function called **write** is used. This function accepts three arguments. First one is the file handle, second one is the address of variable to write (&c in following listing means address of c). You can also use an array instead. And the third argument specifies size of the data to be written. For that a built-in function of C++ language, called **sizeof** can also be used. Following listing writes some bytes to a file. In call to open, O_CREAT means that if this file does not exist, it should be created; O_TRUNC means that if the file exists, it should be truncated to zero length, and O_WRONLY means that we want this file to be opened for writing only.

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <fcntl.h>

void main()
{
        char filename[255], c;
        int handle, num;
        clrscr();

        cout<<"Enter name of file:";
        gets(filename);                                 //Get file name from the user

        handle = open (filename, O_CREAT | O_TRUNC | O_WRONLY);

        if (handle == -1)                               //Checking if the handle returned is an error
        {
                cout<<"Could not open file!" \n";
                getch();
                return;
        }

        cout<<"How many characters do you want to write to the file? \n"
        cin>>num;

        for(int i=0; i<num; i++)
        {
                c = getch();
                write ( handle, &c, sizeof (c) );       //Writing to a file till a certain number is reached
        }
        close (handle);                                 //Close file
}
```

Listing 6.2

WORKING WITH RECORDS

A record is basically an aggregation of some basic data items. In C++ structures are used to create records. A structure is like a single unified record. It is also referred to as composite data type. Once a structure is defined, a variable of its type can be declared just as any other variable. In the listing that follows, note that in open function we are using O_APPEND to specify that if the file exists, we want to append data to it, and O_RDWR to specify that we want to use this file for reading as well as writing. A new function lseek is used to go to a specific location inside a file. This function uses three parameters. First one is the handle, second one is the offset to move to, and the third one is the base from which this offset is calculated. SEEK_SET refers to start of the file. The following program writes some records to a file and reads a record from it.

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <fcntl.h>

struct student
{
	char reg[8];
	char name[30];
	int marks;
};

void main()
{
	student s;
	char filename[255];
	int handle, num;
	clrscr();

	cout<<"Enter name of file:";
	gets(filename);				//Get file name from the user

	handle = open (filename, O_CREAT | O_TRUNC | O_WRONLY);

	if (handle == -1)			//Checking if the handle returned is an error
	{
		cout<<"Could not open file!" \n";
		getch();
		return;
	}

	cout<<"How many records do you want to write to the file? \n"
	cin>>num;

	for(int i=0; i<num; i++)
	{
		cout<<"Enter the reg. no. of students:\n";
		gets(s.reg);
		cout<<"Enter name of student:\n";
		gets(s.name);
		cout<<"Enter marks of the student:\n";
		cin>>marks;
		write ( handle, &s, sizeof (student) );		//Writing to a file till a certain number is reached
	}

	cout<<"\nWhich record do you ant to see:\n";
	cin>>i;
	lseek(handle, (i-1) * sizeof (student), SEEK_SET);
	read(handle, &s, sizeof(student));
	cout<<"Reg. No. :";
	puts(s.reg);
	cout<<"Name:";
	puts(s.name);
	cout<<"Marks:"<<s.marks<<"\n";
	close (handle);				//Close file
	getch();
}
```

Listing 6.3

CSE 101 – Introduction to Computers & Programming
Programming Lab # 6
Handling of Records and Files

Lab Exercise

1. Type and execute all programs given in this lab exercise to make sure you know exactly how they work. Check out what happens if you replace O_TRUNC with O_APPEND in open function of listing 6.2.

2. Write a program to copy one file to another. Your program should take filenames of the two files as input from the user.

CSE 101 – Introduction to Computers & Programming
Programming Lab # 7
C++ Graphics

From the feedback received from the whole of batch 12, I have designed this lab handout a little differently so that it might help you understand the code better. Today's lab is centered on C++ graphics, animations and sounds. The good thing about graphics is that, they help you be creative and give you a better idea about deceiving a user with tricks such as the 3D buttons in windows. First of all, we will discuss a simple program and the basic C++ graphics functions and then we will discuss another program, in which we will discuss how we can make buttons.

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

We are using two new header files. One being ***graphics.h*** and the other being ***dos.h***. These header files are entered so that we can use the built in graphics functions of C++ and also the ***delay(int)*** function so that we can give a slight pause in between animations.

```
void main()

{
        clrscr();

        int grafixerror;                        //This variable checks for graphics error
        int grafixmode;                         //This variable checks for graphics mode
        int grafixdriver = DETECT;              //graphics driver is detected this way

        initgraph ( &grafixdriver, &grafixmode, "C:\\BORLANDC\\BGI") ;   //Initializing graphics
        grafixerror = graphresult();

        if (grafixerror != grOK)                //If the returned value  is not grOK
        {
                cout<<"Graphics Error:"<<grapherrormsg (gerr);
                getch();
                return;
        }
```

To start the graphics system, you must first call **initgraph**. initgraph initializes the graphics system by loading a graphics driver from disk (here we are using a pre-defined constant **DETECT** which detects the graphics driver. We can also put EGA, CGA, VGA etc if we know what driver we want to use) then putting the system into graphics mode. **graphresult( )** returns the error code (an integer in the range -15 to 0) for the last graphics operation that reported an error, then resets the error level to grOk. This means that if the returned value is **grOK** or **0** it means that the graphics were initialized without any error.

```
        setcolor(LIGHTBLUE);
        settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);
        outtextxy(20,20, "Making a green circle");
        setcolor(GREEN);
        circle(getmaxx( )/2. getmaxy( )/2, 100);
        setfillstyle(SOLID_FILL, YELLOW);
        floodfill(300, 300, GREEN);
        getch();
        closegraph();
}
```

settextstyle sets the text font, the direction in which text is displayed, and the size of the characters. **outtextxy** display a text string, using the current justification settings and the current font, direction, and size. The function **circle** draws a circle in the current drawing color and on the x- and y-coordinates given to it

getmaxx returns the maximum x value (screen-relative) for the current graphics driver and mode and similarly, **getmaxy** returns the maximum y value (screen-relative) for the current graphics driver and mode.For example, on a CGA in 320 x 200 mode, getmaxx returns 319 and getmaxy returns 199.

setfillstyle sets the current fill pattern and fill color and after doing that you flood the area bounded by the color border with the selected fill pattern and fill color.

MAKING A BUTTON

In any graphical user interface, a button plays a major role to make your life easy. The first reason being, a button is the best type of visual aid that a user can get. Buttons are so common nowadays that it is really easy to perceive that a certain button has certain functionality. We press a button to turn lights on and off, we turn our computer on with a button, we press buttons to enter values in a calculator. In short, if you show a button on the screen, it is really easy to perceive that what that function it will do. In the following listing, we make a white button that animates when the 'a' key is pressed. This code is helpful for those people who will be making projects like the piano and the typing tutor.

```
#include <iostream.h>
#include <conio.h>
#include <process.h>
#include <graphics.h>
#include <dos.h>

void grafixinit();
void whitebuttonup(int,int,int);
void whitebuttondown(int,int,int);

void main()
{
        int xcor=100;
        int ycor=100;
        char alphabet;
        clrscr();

        grafixinit();

        cout<<"press a button to make it animate, otherwise press any key to quit";
        while(1)
        {
        cleardevice();
        whitebuttonup(xcor, ycor, WHITE);

        alphabet=getch();

                switch(alphabet)
                {
                        case 'a':
                                whitebuttonup(xcor, ycor, BLACK);
                                whitebuttondown(xcor, ycor, WHITE);
                                sound(300);
                                delay(50);
```

```
                                        nosound();
                                        whitebuttondown(xcor, ycor, BLACK);
                                        break;

                                default:
                                        exit(0);
                                        closegraph();
                        }

                }

}

void grafixinit()
{
        int grafixerror;                        //This variable checks for graphics error
        int grafixmode;                                 //This variable checks for graphics mode
        int grafixdriver = DETECT;                      //graphics driver is detected this way

        initgraph ( &grafixdriver, &grafixmode, "C:\\BORLANDC\\BGI") ;   //Initializing graphics
        grafixerror = graphresult();

        if (grafixerror !=0)                            //If the returned value  is not grOK
        {
                cout<<"Graphics Error:"<<grapherrormsg (grafixerror);
                getch();
                exit(0);
        }
}

void whitebuttonup(int x, int y, int color)
{
        setcolor(color);
        rectangle(x,y,x+30,y+30);
        setfillstyle(SOLID_FILL, color);
        floodfill(x+15, y+15, color);

}

void whitebuttondown(int x, int y, int color)
{
        setcolor(color);
        rectangle(x+3,y+3,x+33,y+33);
        setfillstyle(SOLID_FILL, color);
        floodfill(x+18, y+18, color);

}
```

Listing 7.1

CSE 101 – Introduction to Computers & Programming
Programming Lab # 7
C++ Graphics

Exercise 1:

Type and execute the codes given in both the listings to see how Borland C++ graphics work.

Exercise 2:

Make an octave of a piano, where each key will be animated and every key should have a different sound.

CSE-101 L – INTRODUCTION TO PROGRAMMING
Programming lab No. 8
POINTERS

The most confusing and hard part of any programmer's life is the clear & complete understanding of the concept of a pointer. A pointer can be a programmer's delight or a cold grave for all his/her future projects. I would like to explain the concept of a pointer as best as I can. It is only natural if somebody doesn't understand it in the first instance, but I would advise each and every one of you to try their very best and pay utmost attention.

The lvalue (pronounced "el – value") and the rvalue (pronounced "aar-value"):

Each variable has two values; one being the **rvalue** (which is the value on the left hand side of the assignment operator, in other words, the value stored in the variable), while the other being the **lvalue** (which is the **address** of the variable).

Say, you are on the final clue of the treasure hunt. You know that your prize money, say, **3000** bucks, is present in the **café** and you had been living in a cave and don't know the **address** of the café.

From this example we can deduce the following data:

int café = 3000;

Here **café** is the name of the variable, and **3000** is the rvalue of this variable. But we do not know the **address** of this variable, which of course is the lvalue of this variable. We know that the address exists, so in order to extract it, we use **&** (address of) **operator**. So, in other words, we can extract the address of the variable using the following statement:

cout<<&café;

The Pointer Chips in:

Once this is explained, we come to the concept of pointer. To explain this, we again come to our treasure hunt which you are about to win. Now after a little finding, you come to a board, that has many arrows **pointing** at many directions and one of these arrows contains the **address** of café. This means that there is a variable by the name of **arrow** which has the **address** of the variable **café** as its rvalue or which stores the address of another variable.

Thus, we come to the conclusion that a pointer has three values associated with it:

1. The lvalue, which is its own address in memory
2. The rvalue, which is the address of another variable i.e., the lvalue of that variable
3. The rvalue of that variable, which can be accessed by the pointer by using the *** operator**.

Thus, we can write:

```
int *arrow;          //Assigning a pointer to an integer, by the name of arrow
arrow = &café;       //Assigning the address of café to arrow
cout<<arrow;         //Outputting the address of cafe
cout<<*arrow;        //Outputting the value stored in the variable array
```

The following listing helps you manipulate pointers so is quite useful for learning the concept of pointers:

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

void main( )
{
        clrscr( );

        int data = 20;
        int array[10] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

        cout<< ”Address of data is: “<< &data <<endl;
        cout<< ”Value of data is :”<< data <<endl;
        cout<< ”Address of array is: “<< array << endl;
        cout<< ”Elements of array are: “<<endl;

        for( int j = 0; j<10; j++)
        {
                cout<<array[ j ]<<” \ t ”;
        }

        int *ptr;               //Defining a pointer to an integer
        ptr = &data             //Assigning the address of data to the ptr

        cout<< ”Address of pointer is: “<< &ptr <<endl;
        cout<< ”Value of ptr is :”<< ptr <<endl;
        cout<< ”Value pointed to by pointer is: “<< *ptr << endl;

        cout<< ”Assigning the address of array to the pointer”<<endl;
        ptr = array;            //Referencing array[10] as array means we are referring to &array[0]
                                //which can be assigned as value of a pointer
        cout<< ”The current address of pointer becomes: “<<&ptr<<endl;
        cout<< ”Value of ptr is :”<< ptr <<endl;

        cout<< ”Elements pointed by pointer are: “<<endl;

        for( j = 0; j<10; j++ )
        {
                cout<<ptr[ j ]<<” \ t”;         //You can also write cout<<*(ptr + j)<<” \ t”
        }
}
```

Listing 8.1

Returning Multiple Values by Pointers:

Pointers can be used to pass data to functions and then getting the result back from the functions. You might recall the lab on functions, in which we used to return a single value back from the function (which was called **“passing by value”**. Using pointers, you can pass both multiple values and get multiple results back. This can be done in two ways, the first being, **“passing by reference”**. If a parameter is passed this way, it is passed as an alias of the variable being passed. Yet another way is “passing by address” or **“passing by pointer”**. In this case, a pointer to the variable is passed to a function. Here too, all the changes made are reflected in the calling function. Using any of the last two mechanisms, we can get multiple return values from a function. To do that, we would pass multiple variables to a function by address or reference, and assign the desired values to those variables after performing the required operations. The following listing gives examples of such operations:

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

int sqrbyval(int);
void sqrbyref(int, int &);
void sqrbyptr(int, int *);


void main( )
{
        clrscr( );

        int data , number;

        number = 2;
        cout<< ”Number is: “<< number <<endl;
        square = sqrbyval(number);              //Calculating square by value
        cout<< ”Square is :”<< square <<endl;

        number = 3;
        cout<< ”Number is: “<< number <<endl;
        sqrbyref(number, square);          //Calculating square by reference
        cout<< ”Square is :”<< square <<endl;

        number = 4;
        cout<< ”Number is: “<< number <<endl;
        sqrbyptr(number, &square);              //Calculating square by pointer
        cout<< ”Square is :”<< square <<endl;

        getch( );
}

int sqrbyval (int num)              //One parameter is passed by value to this function
{                                   //This function returns value of square of num
        return (num * num)
}

void sqrbyref(int num, int &sqr)    //Second parameter is passed by reference to this function
{                                   //Value of square is assigned to the second parameter
        sqr = num * num;
}

void sqrbyptr(int num, int *sqr)    //Second parameter is passed by pointer to this function
{                                   //Value of square is assigned to the second parameter
        *sqr = num * num;
}
```

Listing 8.2

Dynamically Allocating / De-allocting Memory:

By default, when we declare a variable, memory is automatically allocated to it. We can do the same manually as well by using pointers. We know that a pointer holds the address of a memory, so we can use the **new** keyword of C++ to allocate desired amount of memory for some data item and assign starting address of that memory block back to the pointer. Similarly, we can allocate an array of variable length using pointers. Once we allocate memory using the **new** keyword, we have to de-allocate that memory as well, using the **delete** keyword of C++. It also shows a way of passing an array to a function using pointers.

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

void dispArray(int *, int );


void main( )
{
        clrscr( );

        int i, n;
        int *ptr;                       //Declaring pointer

        ptr = new int;                  //Allocating memory for ptr

        cout<< ”Enter a value to be pointed to by ptr: “<<endl;
        cin>>*ptr;
        cout<< ”Address of pointer is: “<< &ptr <<endl;
        cout<< ”Value of ptr is :”<< ptr <<endl;
        cout<< ”Value pointed to by pointer is: “<< *ptr << endl;

        delete ptr;                     //De-allocating Memory

        cout<< ”Enter the number of elements in array:  “<< *ptr << endl;
        cin>>n;

        ptr = new int [n];              //Allocating memory for desired number of elements for ptr

        for (i=0; i<n; i++)             //Assigning values to the elements of array
        {
                ptr [ i ] = i;
        }

        dispArray ( ptr, n );           //Display this array
        delete ptr;                     //De-allocating Memory
        getch( );
}

void dispArray(int *a, int num)
{
        for( int i = 0; i<num; i++)
        {
                cout<< a[ i ] << ” \ t “;
        }
}
```

Listing 8.3

Lab Exercise:

Type and execute all listings given in this lab exercise, and try to relate them to the concepts discussed.

Practice Exercises:

- Write a function that calculates sum, product, difference, quotient and remainder of two numbers. The function should return results back to the calling function using “Parameter passing by reference”.
- Write another function to perform the same operation using “Parameter passing by pointers”.

CSE 101 – Introduction to Computers & Programming
Programming Lab # 9
Object Oriented Programming

Function Overloading

Function overloading is an important concept to be understood before we touch classes as function overloading plays an important part in classes. Till now, we have known functions to perform one specific function. We can use the same function in different ways to perform the same function. In the following listing, we overload the same function to perform different functions. It is just like asking a person to add two numbers and then asking the same person to add three numbers. See the ***listing 9.1*** for further details.

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

void sum(int,int);
void sum(int,int,int);

void main()
{
        clrscr();
        sum(10,20);
        sum(10,20,30);
        getch();
}

void sum(int one, int two)
{
        cout<<"first function answer: "<<one+two;
}

void sum(int one, int two, int three)
{
        cout<<"\nsecond function answer: "<<one+two+three;
}
```

Listing 9.1

Classes and Structures

Comparison between a Class and a Structure

```
class Bird
{
        char name[10];
        int age;
        say();
        move();
};
```

```
struct Bird
{
        char name[10];
        int age;
        say();
        move();
};
```

Listing 9.2

If you compare the two declarations, you will not notice any difference accept the keyword "**struct**" or "**class**". The difference between them is in their implementation. A Structure's components are defined **public** by default while that of a class are defined **private** by default. Here **private** and **public** are called **access specifiers**.

Private Access Specifier:
A private access specifier indicates as to what code is to be kept private to the class, which means, only the functions defined in the class can only access the private data types and functions.

Public Access Specifier:
A public access specifier indicates as to what code is to be kept public for everyone, which means, the code is visible to both inside and outside the class.

These specifiers are there for data hiding normally known as **data encapsulation**. It is sometimes necessary to hide some data in order to avoid unnecessary errors. Programmers while programming make certain data available for the user which does not change the overall structure of the class. The most typical examples for such classes are the **MFCs** or the Microsoft Foundation Classes.

Functions/Methods in a Class:

Another name for functions is methods and a class can have many methods or no methods at all. Normally, most of the data of class is made private so that no unnecessary change might take place and the methods are made public so as to help the user to use the code as we like him/her to use. But all this declaration of data as private or public depends on the user. The member functions can either be defined within the class or out side the class according to the following listing.

```
Class data
{
        private:
                int data;
        public:
                getdata(int d)
                {
                        data=d;
                }
};
```

```
Class data
{
        private:
                int data;
        public:
                getdata(int);
};

void data::getdata(int d)
{
        data=d;
}
```

Listing 9.3

A function is declared inside the class when the function body is very small. When you declare a function outside the class, you use the **scope resolution operator ::** to specify that which function belongs to which class. Two or more classes might have a function with the same name, so it is imperative that we use the scope resolution operator to differentiate between the different functions of different classes.

In order to access the variables and functions defined in the public access specifier, we use the **dot operator**. It is the same dot operator that we used while we were accessing the members of a structure. So, in case we want to enter some value into the private data member of the class defined in ***listing 9.3***, we cannot use the dot operator in this case, but what we'll do is, in the main function body, we will first of all define an object of the class as

data mydata;

And then

mydata.getdata(100);
Objects and Constructors

A class is a user defined data type, so we can declare variables of this data type commonly known as objects. Calling them objects gives them more of a physical form such as a lever in a machine, it might have different dimensions and might perform different functions. Now say you want to make levers, then when you are

making levers for yourself, you will want them to have a fixed size but if you are going to export them, you must put in some relaxation as to what their dimensions should be. This can be done by using constructors in a class as shown in ***listing 9.4***.

```
class lever
{
        private:
                int length;
                int width;
                int height;
        public:
                lever()
                {
                        length=20;
                        width=10;
                        height=10;
                }

                lever(int len, int wid, int hgt)
                {
                        length = len;
                        width = wid;
                        height = hgt;
                }
};
```

Listing 9.4

here we have defined two constructors to initialize data as we like it to be. A constructor is a member function which has the same name as the class itself, does not return a value and is used to initialize data. Now if we want to make an object of the class having the default dimensions, we will write:

lever handle;

but if we want to define a handle of different dimensions we can write:

lever handle(30, 15, 20);

and it will define a lever according to these dimensions, thus overloading the constructors.

The Deafult Copy Constructor:

You can initialize an object with another object of the same type. Surprisingly, you don't need to create a special constructor for this; one is already built into the all classes. It's called the default copy constructor. It's one argument constructor whose argument is an object of the same class as the constructor. The following listing shows how it works.

```
#include <iostream.h>

class Distance
{
        private:
                int feet;
```

```
		float inches;
	public:
		Distance()
		{ }

		Distance(int ft, float in)
		{
			feet=ft;
			inches=in;
		}

		void getdist()
		{
			cout<<"\nEnter feet= ";
			cin>>feet;
			cout<<"\nEnter inches= ";
			cin>>inches;
		}
		void showdist()
		{
			cout<<"\nfeet = "<<feet<<"\ninches = "<<inches<<endl;
		}
};

void main()
{
	Distance dist1(10,11.5);
	Distance dist2(dist1);
	Distance dist3=dist1;

	cout<<"\nDistance 1 = "
	dist1.showdist();
	cout<<"\nDistance 2 = "
	dist2.showdist();
	cout<<"\nDistance 3 = "
	dist3.showdist();
}
```

Listing 9.5

Lab Exercise

Perform listings 9.1, 9.4 and 9.5 to understand the concepts of function overloading and classes

CSE 101 – Introduction to Computers & Programming
Programming Lab # 10
Pointers to Structures

The first week is dedicated to *arrays*, *pointers*, *structures* & *classes*. Before we start working on advance programs, it is extremely important that the concepts regarding the above mentioned keywords are absolutely clear. We will start this session from the introduction of pointers and structures and then we will combine these two together to check their utility.

Pointers:

A pointer points towards a data type of which, it is itself a pointer. Any variable has two values, an *rvalue*, which is the value stored by that data type and the other one is the *lvalue*, which is the address of the variable. Pointer also has both these values; an *lvalue* which is its address and an *rvalue* which is actually the address of the memory location towards which it is pointing. Since a pointer can point towards any memory location, so we don't really need to know that what is the name of a certain memory location as we can access it using the pointer. Similarly, we can assign any memory location to the pointer (called allocating memory) and when we have done our work, we can release that memory location (called de-allocating memory) at our own pleasure. This property is called *dynamically allocating* and *de-allocating* memory.

Example 10.1:

```
int Value = 5;
int *ptrValue = &Value;
```

Structures:

A structure is an aggregation of many data types. A structure acts as a record and is generally used to keep different data regarding a certain object together and intact. Since everything regarding that object is present in a structure, it becomes easy to use that data at your own pleasure. It makes programming easy as you don't need to memorize the names of the variables as in case of need, you can always go to the place where the structure has been defined and see the variables along with their respective data types constituting that structure.

Example 10.2:

```
struct employee
{
        char name[20];
        int age;
        float pay;
};
```

Combining the Two Together:

A structure is a user defined data type and a variable of this data type can be declared just like a variable of any other data type. Thus, we can also declare a pointer of this user defined data type and use it at our own convenience. In the following example, we are first going to declare a pointer to a structure of type employee and then, we are going to use this pointer to declare an array of structures of variable length. We will go through the code step by step clearly explaining what each line of code does. This example gives a very good idea as to why pointers and structures are so important. According to the following program, you can declare an array of structures of desired length, thus avoiding the possibility of declaring too many or too few variables of the structure data type. And once our job is done, we release these memory locations and ask the user if he/she wants to perform data entry and retrieval again.

Example 10.3:

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

struct employee
{
        char name[20];
        int age;
        float income;
};

void main()
{
        int num,count;
        char choice;
        employee *ptr;                  //Declaring a pointer of type employee
        do
        {
                clrscr();
                cout<<"Enter the number of Employees: ";
                cin>>num;
                cout<<endl;

                ptr=new employee[num];          //Allocating an array of memory locations

                for(count=0;count<num;count++)
                {
                        cout<<"\nEnter the name of the employee No. "<<count<<": ";
                        gets(ptr[count].name);
                        cout<<"\nEnter the age of the employee No. "<<count<<": ";
                        cin>>ptr[count].age
                        cout<<"\nEnter the income of the employee No. "<<count<<": ";
                        cin>>ptr[count].income;
                        cout<<"\n\n";
                }
                cout<<"The data is as following:\n\n";

                for(count=0;count<num;count++)
                {
                        cout<<"\nThe name of the employee No. "<<count<<": ";
                        puts(ptr[count].name);
                        cout<<"\nThe age of the employee No. "<<count<<": ";
                        cout<<ptr[count].age;
                        cout<<"\nThe income of the employee No. "<<count<<": ";
                        cout<<ptr[count].income;
                        cout<<"\n\n";
                }
                delete ptr;                             //De-allocating an array of memory locations
                cout<<"Do you want to enter data again? (y/n)";
                choice=getch();
        }while(choice!='n');
}
```

CSE 101 – Introduction to Computers & Programming
Programming Lab # 10
Pointers to Structures

In this example, we first define a structure of type *employee* which consists of three data types, a character, an integer and a float. These three data types combine together to form this structure and we name it as employee. Then in the main() function, we declare an integer by the name of *num* so that on runtime, we can tell the program that how many employees' data needs to be entered.

After doing so, we declare a pointer *ptr* of type employee and then we use the *do-while* loop so that our program runs for atleast one time. In the the loop we allocate an array of memory locations of type *employee* to the pointer *ptr* by using the statement: -

ptr = new employee[num];

As we know that the name of an array is actually a pointer to its first memory location, we are doing the same with *ptr* as we make it the same pointer which is pointing towards the first memory location of an array of structures of type employee. Putting *num* within the square brackets, we declare this array of structures of our desired length, thus avoiding assigning too much or too less memory than is required. Now we use this name of the pointer to work our way around the different structures and their respective constituents. Since all of these are structures, we use the dot operator to use their constituents as we like, for example: -

puts(ptr[count].name);

Once we are done with our data entry, the program displays the same data, de-allocates the memory and then asks the user if he/she wants to continue. The de-allocation of this memory is done by the following statement: -

delete ptr;

By doing so, we are actually telling the compiler that *ptr* hasn't got any memory allocated to itself, thus, the compiler is free to use those memory locations. In case we don't delete *ptr* and rerun the do-while loop, we are then making a memory leak as each time a new memory location gets allocated when the loop runs and the old memory location doesn't get de-allocated. Thus it is extremely necessary that whenever we allocate memory, we de-allocate it once its use is over. This is a major cause of bugs that appear in applications that use pointers.

Lab Work:

⇨ Type and execute the listing given in **Example 1.3** to understand the concept of pointers, structures, array of structures, dynamic memory allocation and memory de-allocation.

⇨ In order to further strengthen your concepts of memory allocation and de-allocation, data aggregation and data hiding, re-write the code in **Example 1.3** so that all the work is done using classes.

CSE 101 – Introduction to Computers & Programming
Programming Lab # 11
Introduction to Data Structures

Introduction:

Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a *data structure*. The choice of a particular data model depends on two considerations: First, it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary.

Arrays & Linked Lists:

Data structures are classified as linear or non linear. A data structure is said to be linear if its elements form a sequence, or, in other words, a linear list. There are two basic ways of representing such linear structures in memory. One way is to have linear relationship between the elements represented by means of sequential memory locations called *arrays*. The other way is to have linear relationship between the elements represented by means of pointers or links. These linear structures are called *linked lists*.

Stack:

Your past experience in arrays while working in C++ must have let you believe that you can enter and delete data at any place in the array. Although this might appear very useful, but sometimes, you require inputting data or getting the output of data from any end of the list and not from the middle. As an example, consider a stack of dishes. You keep putting the dishes one after the other, thus the dish that goes in last, comes out first. Such a collection of dishes is normally called a *stack*, which in turn, is also called *last-in first-out* (LIFO) list.

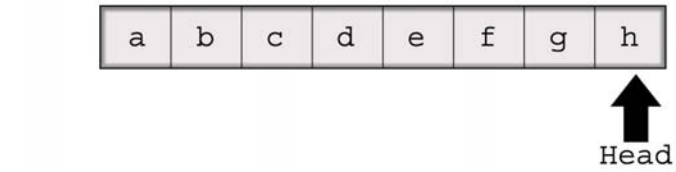
A *stack* is a list of elements in which an element may be inserted or deleted only at one end. This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

Special terminology is used for two basic operations associated with stacks:

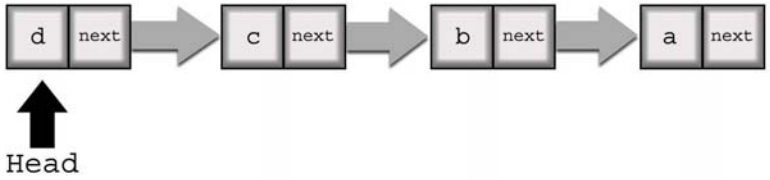
(a) "Push" is the term used to insert an element into a stack.
(b) "Pop" is the term used to delete an element from a stack.

The point from where data is entered or delete from a stack is called the *head* of that stack. So in a stack, the *head* becomes the focus of attention. When the data is entered into a stack, the *head* points towards the latest entry and when ever you pop an element, the *head* starts pointing towards the new *head* of that stack.

The problem with stack implementation while using an array is that, it has a fixed length. Apart from that, you can only store a fixed data type at one time and doesn't provide much room for multiple data types. In such a situation, we can use pointers to declare a linked list according to our own needs and store data according to our liking.



Stack implementation using an array



Stack implementation using a linked list

In order to work in a linked list, we will make a structure of type *node* to contain two data types; a character by the name of *data* and a pointer *next*, of type *node*. Characters will be stored in *data* and the pointer will be used to point towards a new memory location containing another node.

```
struct node
{
        char data;
        node *next;
}
```

During the push operation, the first character is entered into the node marked as *head*. Upon entering the next character, a new node is created by the name of *temp* and the character is added into the data of that new node. We then make its *next* pointer to point towards head so that we might not lose track of this node when the head is moved. Now we make the head point towards the temp node so that we might keep track of the top of the stack.

While using the member of a pointer of type *struct*, we use the indirect *member selection operator* -> or simply a *hyphen* followed by the *greater than* sign. Thus in order to access the data member of *head, we write:

head -> data

It is a convenient synonym for

(*head).data

The following code shows how a linked list can be used to make a stack of variable length:

Example 11.1:

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

class stack                                  //Class of a stack
{
        private:
                struct node                          //structure of a node
                {
                        char data;
                        node *next;
                };
                char value;
                node *head;
        public:
                stack()                              //Constructor
                {
                        head =  NULL;
                }
                void push(char);
                char pop();
};

void main()
{
        stack stk;
        char choice;
        char value;
        while(1)
        {
                clrscr();
                cout<<"Enter your choice:\n1.\tPush\n2.\tPop\n3.\tExit\n";
                choice=getch();

                switch(choice)
                {
                        case '1':
                                cout<<"Enter your data: ";
                                cin>>value;
                                stk.push(value);
                                break;
                        case '2':
                                cout<<stk.pop();
                                getch();
                                break;
                        case '3':
                                exit(0);
                        default:
                                cout<<"\Invalid Input";
                }
        }
}
```

```
void stack::push(char input)
{
        if(head == NULL)        //for first node
        {
                head = new node;
                head -> data = input;
                head ->next = NULL;
        }
        else
        {
                node *temp;
                temp = new node;
                temp->data = input;
                temp->next = head;
                head=temp;
        }
}

char stack::pop()
{
        if(head->next==NULL)
        {
                value=head->data;
                delete head;
                head=NULL;
                return value;
        }
        else
        {
                node *temp;
                temp = head;
                value=temp->data;
                head = head->next;
                delete temp;
                return value;
        }
}
```

Lab Work:

⇨ Make a program that implements a stack by using an array.

⇨ Write a function for the stack class of *Example 2.1* that shows you the current status of the stack, i.e., shows you the current data stored in the stack.