

Programming Fundamentals

LAB MANUAL



**DEPARTMENT OF ELECTRICAL ENGINEERING,
FAST-NU, LAHORE**

Created by: Ms. Hina Tariq and Mr. Waqas Ur Rehman

Date: August, 2014

Last Updated by: Mr. Waqas ur Rehman

Date: Jan, 2017

Approved by the HoD:

Date: July, 2015

Table of Contents

Sr. No.	Description	Page No.
1	List of Equipment	04
2	Experiment No. 01, Static Arrays and File Streaming	05
3	Experiment No. 02, Pointers and Dynamic Arrays	09
4	Experiment No. 03, Double Pointer and 2-D Arrays	13
5	Experiment No. 04, Structure	17
6	Experiment No. 05, Nested Structure and Pointer to Structure	20
7	Experiment No. 06, Classes	24
8	Experiment No. 07, Separate Compilation, Friend Function and Friend Classes	29
9	Experiment No. 08, Composition	33
10	Experiment No. 09, Inheritance	38
11	Experiment No. 10, Polymorphism	43
12	Experiment No. 11, Operator Overloading	48
13	Experiment No. 12, Templates	53
14	Experiment No. 13, Project Session	57
15	Experiment No. 14, Project Session	58
16	Appendix A, Lab Evaluation Criteria	59
17	Appendix B, Safety around Electricity	60
18	Appendix C, Guidelines on preparing Lab Reports	62

List of Equipment

Sr. No.	Description
1	Workstations (PCs)
2	Visual Studio 2010 C++ (software)

ExperimentNo. 01

Static Arrays& File Streaming

OBJECTIVE:

Things that will be covered in today's lab:

- Introduction to Compiler
- Revision of Static Arrays

Introduction to Compiler (Visual Studio 2010)

1. Creating a new project:

Go to the *File* menu. Select *New*. In the *Project* Tab, select *Win32 Console Application*. Write the Project Name. You can change the default location of the project from the location box. Check the Win32 Platform. Press OK. A new window appears. Select *An Empty Project*. Press *Finish*. If a new window appears showing project details, press OK.

2. Source File (.cpp):

Go to *Project* menu. Select *Add to Project*. Select *New*. From File Tab, select a *C++ Source file*. Write name of file and press ok.

3. Compiling the code:

After writing the code, press *CTRL+ F7*, or from the *Build* menu, select *Compile (Building the executable file)*. After compiling, press *F7*.

4. Running the exe file:

Press *CTRL+F5* for *Debugging the code* at any line of the code. Press *F9* to insert a breakpoint. Then do the following to start debugging.

Build>Start Debug>Go OR Press *F5*. Debugging is started now. Press *F11* for *step into*, *F10* for *step over* and *shift+f11* for *step out*.

File Streaming:

We have been dealing with input and output streams the whole time (*cin* and *cout*). The input/output stream is used with `<iostream>` library.

In C++ and any other programming language, you can read and write from a text file (.txt extension). We have to include a library in the same way as we include a library for *cin* and *cout*.

A five step process:

1. Include the header file <i>fstream</i>	<code>#include<fstream></code>
2. Declare the file stream variables	<code>Ifstream inDATA; Ofstream outDATA;</code>
3. Open input output files	<code>inDATA.open ("Infile.txt"); outDATA.open ("Outfile.txt");</code>
4. Read or write data from files	<code>inDATA>>varName1; outDATA<<varName2;</code>
5. Close the file	<code>InDATA.close(); outDATA.close();</code>

Exercise 1:(20 points)

A selection sort code searches an array looking for the smallest element in the array. Then, the smallest element is swapped with the first element of the array. The process is repeated for the sub-array beginning with the second element of the array. Each pass of the array results in one element being placed in its proper location. When the sub-array being processed contains one element, the array is sorted. Write C++ code for this selection sort and output must be stored in "output.txt" file (having all passes).

Example:

Unsorted list	23 78 45 08 32 56
First Pass	08 78 45 23 32 56
Second Pass	08 23 45 78 32 56
Third Pass	08 23 32 78 45 56
Fourth Pass	08 23 32 45 78 56
Fifth Pass	08 23 32 45 56 78
Sorted List	08 23 32 45 56 78

Exercise 2: (15 points)

Write a program that reads students' names followed by their test scores. The program should output each student's name followed by the test scores and the relevant grade. It should also find and print the highest test score and the name of the students having the highest test score.

Student data is stored in a "student.txt" file and are arranged in the following order.

- *Student First Name* of type string
- *Student Last Name* of type string
- *Test Score* of type int (testScore is between 0 and 100)
- *Grade* of type char.

Suppose that we have data of 20 students in a file. Your program must contain at least the following functions:


- A function to read the student's data into the array.
- A function to assign the relevant grade to each student.
- A function to find the highest test score.
- A function to print the names of the students having the highest test score.

Your program must output each student's name in this form: last name followed by a comma, followed by a space, followed by the first name; the name must be left justified. Moreover, other than declaring the variables and opening the input and output files, the function main should only be a collection of function calls.

Post Lab:(10 points)

Write a program which takes 10 values as input and stores them in an array. Calculate the average of five values starting from index zero. Increment the index by one till the end of array. On every increment, calculate average of next five values starting from incremented index and store the calculated average in a separate array.


Example:



5	12	7	6	45	10	11	56	45	12
---	----	---	---	----	----	----	----	----	----

$$\text{Average} = (5+12+7+6+45)/5 = 15$$

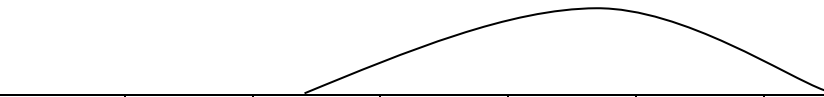
15									
----	--	--	--	--	--	--	--	--	--



5	12	7	6	45	10	11	56	45	12
---	----	---	---	----	----	----	----	----	----

$$\text{Average} = (12+7+6+45+10)/5 = 16$$

15	16								
----	----	--	--	--	--	--	--	--	--




5	12	7	6	45	10	11	56	45	12
---	----	---	---	----	----	----	----	----	----

$$\text{Average} = (7+6+45+10+11)/5 = 15.8$$

15	16	15.8							
----	----	------	--	--	--	--	--	--	--

•
•
•



5	12	7	6	45	10	11	56	45	12
---	----	---	---	----	----	----	----	----	----

$$\text{Average} = (10+11+56+45+12)/5 = 13.8$$

15	16	15.8	25.6	33.4	26.8				
----	----	------	------	------	------	--	--	--	--

ExperimentNo. 02

Pointers and Dynamic Arrays

OBJECTIVE:

Things that will be covered in today's lab:

- Pointers
- Dynamic Arrays

THEORY:

Pointer variable: A variable whose content is an address (i.e., a memory address). In C++, you declare a pointer variable by using the asterisk symbol (*) between the datatype and the variable name. The general syntax to declare a pointer variable is as follows:

```
datatype * identifier;
```

In C++, the *ampersand* (&), address of the operator, is a unary operator that returns the address of its operand. Similarly “*” is a dereferencing operator, refers to the object to which its operand (pointer) points. For example, given the statements:

```
int x;
int *p;
p = &x;           // assigns address of x to p
cout << *p << endl; // pointer p points to x
```

The arrays discussed in last lab are called **static arrays** because their size was fixed at compile time. One of the limitations of a static array is that every time you execute the program, the size of the array is fixed. One way to handle this limitation is to declare an array that is large enough to process a variety of data sets. However, if the array is very big and the data set is small, such a declaration would result in memory waste. On the other hand, it would be helpful if, during program execution, you could prompt the user to enter the size of the array and then create an array of the appropriate size.

Dynamic Array: An array created during the execution of a program. To create a dynamic array, we use *new* operator.

```
int size;
int *p;
p = newint [size];
```

If you are not in need of dynamically allocated memory anymore, you can use *delete* operator, which de-allocates memory previously allocated by new operator.

```
delete [] p;
```

Exercise 1: (5 points)

Write the output of the following C++ codes without running the code in Visual Studio.

a)

```
int x;
int y;
int *p=&x;
int *q=&y;
x=35;
y=46;
p=q;
*p=78;
cout<<x<<" "<<y<<" ";
cout<<*p<<" "<<*q;
```

b)

```
int x[3]={0,4,6};
int *p,t1,t2;
p=x;
(*p)++;
cout<<*p;
cout<<*(p+1);
```

c)

```
int x,*p,*q;
int arr[3]={0};
p=arr;
q=p;
*p=4;
for (int j=0;j<2;j++)
{
    x=*p;
    p++;
    *p=(x+j);
}
for (int k=0;k<3;k++)
{
    cout<<*q;
    q++;
}
```

Exercise 2: (10 points)

Write a function **resize()** that takes as arguments: *a pointer pointing to the array of integers*, *its size*, and a *new_size*. *New_size* can be any number greater than 0. This function should change the size of the array. If the new size is greater than the previous one, then insert zeroes in new cells.

Example:

Case 1: (new_size > size)

new_size=7, size=5

Before calling resize function:

Array =>

2	32	4	34	51
---	----	---	----	----

After calling resize function:

Array =>

2	32	4	34	51	0	0
---	----	---	----	----	---	---

Case 2: (new_size < size)

new_size=3, size=5

Before calling resize function:

Array =>

2	32	4	34	51
---	----	---	----	----

After calling resize function:

Array =>

2	32	4
---	----	---

Exercise 3:(10 points)

Write a code that merges two arrays. Create two dynamic arrays of sizes *size_1* and *size_2* respectively. Take input in these arrays from the user. Now create a third array of *size* (*size_1+size_2*) and insert all the elements of both arrays in this array. Remove the duplicate elements from this array and resize the array to a smaller size.

Example:

Array 1=>

1	2	3	4
---	---	---	---

Array 2=>

3	4	5	6	7
---	---	---	---	---

After merging Array1 and Array2:

Array 3=>

1	2	3	3	4	4	5	6	7
---	---	---	---	---	---	---	---	---

After removing duplicate elements, this array should be of size 6:

Array 3=>

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Post Lab: (10 points)

Consider following main function:

```
void main()
{
    char input[100];
    cin.getline(input,100);
    //For example, user enters National University.

    char *q=input;
    ReverseSentence(q);
    cout<<input<<endl;

    // Now input array should be changed to lanoitaNytisrevinU.
}
```

Write the implementation of the function **void ReverseSentence(char*)**. Assume that each sentence ends with a full stop. You should use the following function **ReverseWord()** to reverse each word of the whole sentence. You are not allowed to use any static array. You are only allowed to use simple character pointers.

```
void ReverseWord(char *p, int len)
{
    char temp;
    for(int i=0; i<len/2; i++)
    {
        temp=p[i];
        p[i]=p[len-i-1];
        p[len-i-1]=temp;
    }
}
```

“p” is a pointer pointing to the first location of *char* array and length is the number of characters in the array. For example, if p is pointing to “HELLO” then the length is 5. After calling this function, p is pointing to “OLLEH”.

ExperimentNo. 03

Double Pointers and 2-D Dynamic Arrays

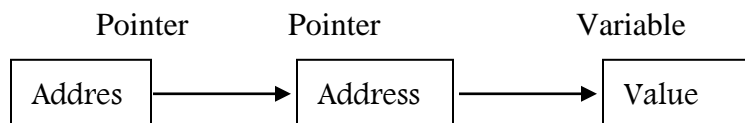
OBJECTIVE:

Things that will be covered in today's lab:

- Double Pointers
- Dynamic Two Dimensional Arrays

THEORY:

Double Pointer: A double pointer is a pointer to pointer. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below:



In C++, you declare a double pointer variable by using two asterisks (**) between the data type and the variable name. The general syntax to declare a pointer variable is as follows:

```
datatype ** identifier;
```

```

int var=3000;
int *ptr;
int **pptr;

ptr = &var;           // assigns the address of var to ptr
pptr = &ptr;          // assigns the address of ptr to pptr

cout << var <<endl;
cout << *ptr <<endl;  // pointer points to var
cout << **pptr<<endl; // Pointer points of ptr
  
```

Dynamic 2-D Array: First, we will allocate memory for an array which contains a set of pointers. Next, we will allocate memory for each array which is pointed by pointers. The de-allocation of memory is done in the reverse order of memory allocation.

```

int **Array = 0;
Array = new int *[ROWS]; //memory allocated for elements of
// row.
for(int i=0; i<ROWS; i++) //memory allocated for each col.
Array[i] = new int [COLUMNS];

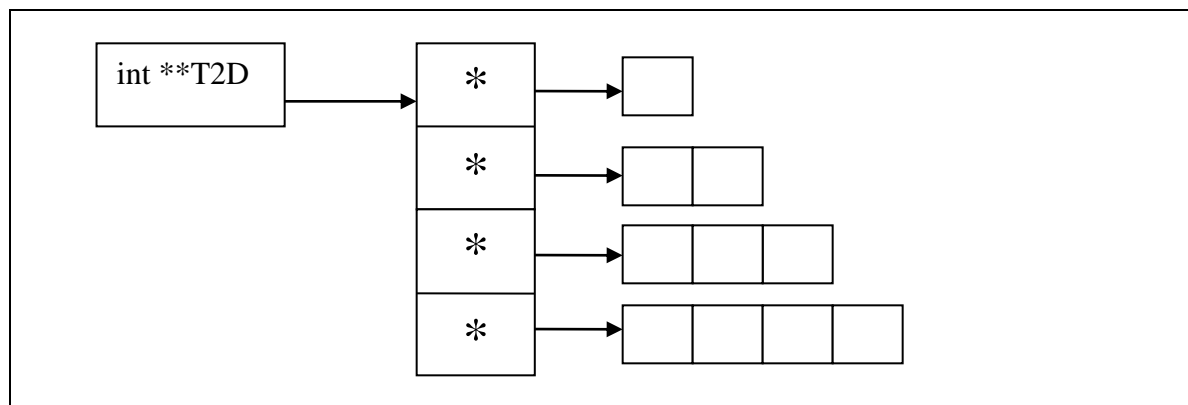
for(int i=0; i<ROWS; i++) //free the allocated memory
delete [] Array[i] ;
delete [] Array ;
  
```

Exercise 1:(10 points)

A double pointer is used for declaring two dimensional arrays dynamically. For example,

```
int **p;
p=newint *[rows];
for(int i=0; i<rows; i++)
p[i]=newint [cols];
```

We want to implement a **triangular 2D** array in which each row has one column greater than the previous one. i.e., the first row has one column, the second one has two columns, and the third one has three columns and so on. You have to take the total number of rows from the user. Following is an example of a triangular 2D array with four rows.



Write following functions with given prototype:

a. `void AddColumns (int * &, int size)`

This function takes a single pointer by reference and dynamically allocates memory to it. You will call this function in main to allocate number of columns to each row turn by turn.

```
int main( ){
    int **p;
    p=newint *[rows];
    for(int i=0; i<rows; i++)
        //call function...
}
```

b. `void RowWiseInput(int *, int size)`

This function simply takes a pointer as an argument and takes input in it from the user. The second argument is the size of 1D array pointed by pointer.

c. `void RowWisePrint(int *, int size)`

This function takes a pointer as argument and prints its contents. The second argument is the size of 1D array pointed by pointer.

d. `int main()`

In main function you have to do the following tasks:

- 1) Ask the user to enter the number of rows.
- 2) Declare a 2D array.
- 3) Allocate memory for its columns in this array using **AddColumns** Function defined above.
- 4) Take input in the 2D array using **RowWiseInput** function.
- 5) Print the 2D array using **RowWisePrint** function.
- 6) You may use loops in main.
- 7) No global variables are allowed.

Exercise 2: (10 points)

(Airplane Seating Assignment) Write a program that can be used to assign seats for a commercial airplane. The airplane has 13 rows, with six seats in each row. Rows 1 and 2 are first class, rows 3 through 7 are business class, and rows 8 through 13 are economy class. Your program must prompt the user to enter the following information:

- Ticket type (first class, business class, or economy class)
- Desired seat

Output the seating plan in the following form:

	A	B	C	D	E	F
Row 1	*	*	X	*	X	X
Row 2	*	X	*	X	*	X
Row 3	*	*	X	X	*	X
Row 4	X	*	X	*	X	X
Row 5	*	X	*	X	*	*
Row 6	*	X	*	*	*	X
Row 7	X	*	*	*	X	X
Row 8	*	X	*	X	X	*
Row 9	X	*	X	X	*	X
Row 10	*	X	*	X	X	X
Row 11	*	*	X	*	X	*
Row 12	*	*	X	X	*	X
Row 13	*	*	*	*	X	*

Here, * indicates that the seat is available; X indicates that the seat is occupied. Make this a menu-driven program; show the user's choices and allow the user to make the appropriate choices.

Post Lab:(10 points)

Write a C++ program to take the transpose of a matrix. A transpose of the matrix, is the matrix obtained by interchanging the rows and columns of the original matrix. Consider following examples:

1. Given Matrix

1	2	3
4	5	6
7	8	9

Transpose:

1	4	7
2	5	8
3	6	9

2. Given Matrix

1	2
3	4
5	6

Transpose:

1	3	5
2	4	6

A matrix is represented by a 2D array. Use a double pointer to declare this 2D array dynamically. The number of rows and columns should be taken as input from the user. Your code should be generic, i.e., it should work for any number of rows and columns.

You will have to create a new matrix of reversed size. For example, if the input matrix is of size (2x3), the new matrix should be of a size (3x2). Create the new matrix dynamically.

ExperimentNo. 04

Structure

OBJECTIVE:

Things that will be covered in today's lab:

- File Input and Output Streams
- Structures

THEORY:

Structs:

We have already discussed array which is a structured data type having all elements of the same type. Another structured data type is a *struct* (or *record*) which allows you to group related values that are of different types.

A *struct* is a collection of fixed number of components, members, in which the members are accessed by name. The members may be different data types.

Defining a Structure:

A *struct* is a user defined data type that requires the keyword *struct* followed by the name chosen for the *struct*. The data members that define *struct* are contained between a set of curly brackets. *Struct* definition must be ended with a semicolon.

```
struct struct_name
{
    //
    // Member variables
    //
};
```

Accessing Structure Members:

To access any member of a structure, we use the *member access operator* (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the *struct* keyword to define variables of structure type.

Example: What should be the output of this program?

```
struct uni_t {  
    char * title ;  
    int year;  
};  
int main () {  
    uni_t uni;  
  
    uni.title= "FAST University";  
    uni.year = 2014;  
  
    cout << uni.title;  
    cout <<" ("<< uni.year <<") \n";  
    return 0;  
}
```

Exercise 1: (10 points)

Write a program that reads a telephone number from a file *phnum.txt* (available in lab folder) in the form xxx-xxxx. The first three digits represent an area code and the next 4 digits represent the phone number. Your task is to print these numbers into another file (*outFile.txt*) without any space or '-' between the area code and the number. Your program should define a function that has input and output file streams as arguments.

For example,

If the phone number is 042-5610, the output should be 0425610.

Exercise 2: (15 points)

Define a struct, **Item**, with two components: *name* and *price* of the item.

Write a program to help FAST-NUCES restaurant automate its breakfast billing system. The program should do the following:

Show the customer different breakfast items offered by the restaurant. Assume that the restaurant offers the following breakfast items (the price of each item is shown to the right of the item):

- Bacon and Egg \$2.45
- French Toast \$1.99
- Fruit Basket \$2.49
- Cereal \$0.69
- Coffee \$0.50
- Peanut butter: \$0.80

Allow the customer to select more than one item from the menu. Calculate and print the bill. Your program must contain at least the following functions:

1. **getData**: This function loads the data into the array *menuList* and number of items may vary.
2. **showMenu**: This function shows the different items offered by the restaurant and tells the user how to select the items.
3. **printCheck**: This function calculates and prints the check.

(Note that the billing amount should include a 5% tax.) The customer can select multiple items of a particular type. A sample output in this case is:

Welcome to FAST-NUCES Restaurant

1. Bacon and Egg \$2.45
2. Muffin \$1.98
3. Coffee \$0.50

Tax \$0.25

Amount Due \$5.18

Post Lab:(15 points)

Write a program that allows two players to play the tic-tac-toe game. Your program must contain the struct **TicTacToe** to implement a *TicTacToe* object. Include a 3-by-3 two-dimensional array, as a member variable, to create the board. If needed, include additional member variables. Some of the operations on a *TicTacToe* object are

- Printing the current board
- Getting a move
- Checking if a move is valid
- Determining the winner after each move.

Add additional operations as needed.

ExperimentNo. 05

Nested Structure and Pointer to Structure

OBJECTIVE:

Things that will be covered in today's lab:

- Nested Structures (A struct within a struct)
- Pointer to Structure

THEORY:

A nested structure in C++ is nothing but a structure within a structure. One structure can be declared inside the other structure as we declare structure members inside a structure. A structure variable can be a normal structure variable or a pointer variable to access the data.

Declare two separate structDeclare a nested struct:

```
struct xyz{  
    // xyz member variables  
};  
struct abc  
{  
    xyz var;  
    // abc member variables  
}a;  
  
// Access member variables  
a.abc_member  
a.var.xyz_member
```

```
struct abc{  
    // abc member variables  
    struct xyz  
    {  
        // xyz member variables  
    }var;  
}a;  
  
// Access member variables  
a.abc_member  
a.var.xyz_member
```

Access nested members:

- You can access the members of “*abc*” struct using dot operator.
- A “*xyz*” structure is nested within the “*abc*” struct so the members of “*xyz*” can be accessed using “*abc*” struct object.

Pointer to structure

You can define pointers to structures in a very similar way as you define a pointer to any other variable. Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place “&” operator before the name of the structure.

Here variable is an object of structure type *struct_name* and *p_var* is a pointer to point to objects of structure type variable. Therefore, the following code would also be valid:

```
struct_name * p_var;
struct_name variable;
p_var = & variable;
p_var->struct_members;
```

The value of the pointer *p_var* would be assigned the address of the object variable. The arrow operator (*->*) is a **dereference** operator that is used exclusively with pointers to objects that have members. This operator serves to access the member of an object directly from its address.

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed to by a	(*a).b

Example: What should be the output of this program?

```
struct movies_t
- {
char * title ;
int year;
};
int main ()
{
    movies_t amovie;

    movies_t * pmovie;
    pmovie = & amovie;

    pmovie->title= "MATRIX";
    pmovie->year = 1999;
    cout << pmovie->title;
+   cout <<" ("<< pmovie->year <<")\n";
}
```

Exercise 1: (15 points)

Write a program to help a university system to store records for its employee. You have to perform the following tasks:

1. Define a struct **facultyMember** with the following attributes:
 - ID number (int)
 - First Name (string)
 - Last Name (string)
 - Designation (string) e.g. Assistant professor, Lecturer etc.
2. Implement a function “**void newrecord(facultyMember & fm)**”, which will take an argument of *FacultyMember* type, input values for all the attributes from user, and store it in the argument variable.
3. Implement a function “**void printdetails(facultyMember fm)**”, which will print the values of the variable fm passed as an argument.
4. Implement your main function. Declare a variable of *FacultyMember* type. Assign values to it using *NewRecord* function. Print its values using *PrintDetails* function.
5. Now, declare a *FacultyMember* type array of size 3 in *main()*. Fill the values using *newRecord* function. (Note that your *newRecord* can assign value to a single *FacultyMember* type variable and you cannot change the prototype).
6. Print the values of the above array using *PrintDetails* Function without changing the prototype.
7. Implement a function “**void sortid(facultyMember fm [], int size)**”, which takes a *FacultyMember* type array as an argument and its maximum size. You have to sort this array in ascending order with respect to the ID number (use any sorting algorithms). Be careful while swapping the two locations of array.

```
// Point 5
facultyMember f1[3];
for( int i=0; i<3; i++)
newrecord(f1[i]);
for( int j=0; j<3; j++)
printdetails(f1[j]);

// Point 7
sortid(f1,3);
for( int k=0; k<3; k++)
printdetails(f1[k]);
```

Exercise 2: (15 points)

Implement another struct **University** with the following details:

- Name of University (string)
- Address (string)
- List of faculty Members (*FacultyMember* type array of size 3)

In main (), declare a variable *myUni* of the *University* type. Set any values to name and address of *myUni*. Copy the values of the array you declared in step5 into the *FacultyMember* type array of *myUni*.

Print the details of the *myUni*. You should use the *PrintDetails* function for printing the details of Faculty member list of *myUni*.

```
university uni;
uni.name="National University";
uni.address="Block-B,Faisal Town";
for( int b=0; b<3; b++) {
// Assign value university faculty member
}
for( int a=0; a<3; a++) {
    cout<<uni.name<<"    "<<uni.address<<"    ";
    printdetails(uni.FM[a]);
}
```

Post Lab: (15 points)

Write a C++ program to read data and display the author's record. The structure definition of authors should include the following details:

- Author's ID
- Author's Name
- Book List
 - Book Code
 - Book name
 - Subject
 - Book Price
 - Edition

The record of each author should contain details of three books.

ExperimentNo. 06

Classes

OBJECTIVE:

Things that will be covered in today's lab:

- Classes
- C++ Structs

THEORY:

Classes are an expanded concept of *data structures*: like data structures, they can contain data members, but *classes* can also contain functions as members.

Classes have the same format as plain *structures*, except that they can also include functions and have these new things called *access specifiers*. An *access specifier* is one of the following three keywords: **private**, **public** or **protected**. These specifiers modify the access rights for the members that follow them:

- Private members of a class are accessible only from within other members of the same class (or from their "*friends*").
- Protected members are accessible from other members of the same class (or from their "*friends*"), but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before any other *access specifier* has private access automatically.

Class Definition: Classes are defined using either keyword *class*, with the following syntax:

```
class class_name
{
    access_specifier:
    //data member;
                                //member functions;
    access_specifier:
    //data member;
                                //member functions;
};
```


Access Class Members:

To access any member variable or a function of the class, we use the *member access operator* (.). The member access operator is coded as a period between the class object and member variable or function of class.

Define class member function:

In outside definition, the operator of scope (::) is used to specify that the function being defined is a member of the class and not a regular non-member function.

```
return_type class_name:: function_name( arguments )
```

C++ *structs* are almost similar to classes and have the same syntax for definition. The only difference is that the reserve word *struct* is used and all functions and data member within a class are *pubic* by default instead of *private* as in classes.

Exercise 1: (5 points)

Copy and paste the following code in a text editor and check output. What is the difference between C++ Structs and Classes?

```
struct facultyMember
{
private:
    int id;

public:
    void rec(int value);
    void printdetails();
};
```

```
int main()
{
    facultyMember f;
    f.rec(10);
    f.printdetails();
    return 0;
}

void facultyMember::rec(int val)
{
    id=val;
}
void facultyMember::printdetails()
{
    cout<<"ID : "<<id<<endl;
}
```

```
class facultyMember
{
private:
    int id;

public:
    void rec(int value);
    void printdetails();
};
```

```
int main()
{
    facultyMember f;
    f.rec(10);
    f.printdetails();
    return 0;
}

void facultyMember:: rec(int val)
{
    id=val;
}
void facultyMember::printdetails()
{
    cout<<"ID : "<<id<<endl;
}
```

Exercise 2: (15 points)

Design and Implement a class **dateType** that manages a calendar. The class **dateType** should store *day, month, and year* and display it in the following format **(01-May-1998)**. For any object of **dateType**, your program should be able to perform the following operations on that object.

1. Set the date
2. Get the date
3. Display the date
4. Increment the day by 1
5. Increment the month by 1
6. Increment the year by 1
7. Create another object of **dateType** and compare it with the object you created earlier; determine whether they are equal or not.
8. Increment the day, month or year by certain number, suppose the date is 29-4-1976 and we add 4 days to it, now the date should be 3-5-1976.

Your class should be defined like this

```
class dateType
{
private:
    int day, month, year;
    string mon; //Month name

public:
    //define functions here with appropriate parameters and //return
    type.
}
```

Sample main function:

```
void main()
{
    dateType D;
    dateType C;
    int d,y,m;

    cout<<"Enter date"<<endl;
    cin>>d>>m>>y;
    cout<<endl;
    D.setdate(d,m,y);
    D.printdate();
    cout<<endl;
    cout<<"*****"<<endl;
    d=31,
    m=9;
    y=1995;
```

```

    cout<<"Set date:"<<d<<"-"<<m<<"-"<<y<<endl;
    D.setdate(d,m,y);
    cout<<endl;
    D.printdate();
    cout<<endl;
    cout<<"*****"<<endl;

    d=28, m=2, y=1998;
    cout<<"Set date:"<<d<<"-"<<m<<"-"<<y<<endl;
    D.setdate(28,2,1998);
    D.printdate();
    cout<<endl;
    cout<<"*****"<<endl;

    cout<<"AFTER INCREMENTING BY 1 DAY"<<endl;
    D.incdays();
    D.printdate();
    cout<<endl;
    cout<<"*****"<<endl;

    cout<<"If "<<endl;
    D.setdate(30,11,2007);
    D.printdate();
    cout<<endl;
    cout<<"AFTER INCREMENTING BY 1 MONTH"<<endl;
    D.incmonth();
    D.printdate();
    cout<<endl;
    cout<<"*****"<<endl;
    cout<<"AFTER INCREMENTING BY 1 YEAR"<<endl;
    D.incyear();
    D.printdate();
    cout<<endl;
    cout<<"*****"<<endl;

    C.setdate(12,3,1999);
    cout<<"New object created:"<<endl;
    cout<<"DATE1:"<<endl;
    C.printdate();
    cout<<"DATE2:"<<endl;
    D.printdate();
    cout<<endl;
    D.equal(C);

    cout<<"*****"<<endl;
    cout<<"If ";
    D.printdate();
    cout<<endl;
    cout<<"Date after 9 days will be"<<endl;
    D.incdays(9);
    D.printdate();
}

```

Post Lab: (15 points)

Create a class **Rational** for performing arithmetic with fractions. Write a program to test your class.

Use an integer variable to represent the *Private* data of the class-- the *numerator* and the *denominator*. Provide a *constructor* that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializes are provided and should store the fraction in reduced form. For example, the fraction 2/4 should be stored in object as 1 in the numerator and 2 in the denominator. Provide *public* member functions that perform each of the following tasks:

1. Add two **Rational** numbers. The result should be stored in reduced form.
Two rational numbers a/b and c/d can be added as follows:
$$(a/b) + (c/d) = (a*d + c*b) / (b*d)$$
2. Multiply two **Rational** numbers. The result should be stored in reduced form.
The product of two rational numbers a/b and c/d can be found as follows:
$$(a/b) * (c/d) = (a*c) / (b*d)$$
3. Divide two **Rational** numbers. The result should be stored in reduced form.
Two rational numbers a/b and c/d can be divided as follows:
$$(a/b) \div (c/d) = (a*d) / (b*c)$$
4. Print **Rational** numbers in the form **a/b** where **a** is the numerator and **b** is the denominator.

ExperimentNo. 07

Separate Compilation, Friend Function and Friend Classes

OBJECTIVE:

Things that will be covered in today's lab:

- Separate Compilation
- Friend Function and Friend Classes:

THEORY:

You might be wondering why you need header files and why you would want to have multiple .cpp files for a program. The reasons for this are simple:

- It speeds up compile time.
- It keeps your code more organized
- It allows you to separate *interface* from *implementation*

In C++, the contents of a module consist of structure type (struct) declarations, class declarations, global variables, and functions. The functions themselves are normally defined in a source file (a “.cpp” file). Except for the main module, each source (.cpp) file has a header file (a “.h” file) associated with it that provides the declarations needed by other modules to make use of this module.

The idea is that other modules can access the functionality in module X simply by *#including* the “X.h” header file and the linker will do the rest. The code in X.cpp needs to be compiled only the first time or if it is changed; the rest of the time, the linker will link X's code into the final executable without needing to recompile it, which enables IDEs to work very efficiently.

Header files contain mostly declarations that are used in the rest of the program. The skeleton of a class is usually provided in a header file. Header files are not compiled, but rather provided to other parts of the program through the use of *#include*.

A typical header file looks like the following:

```
// Inside sample.h
#ifndef SAMPLE_H
#define SAMPLE_H
// Contents of the header file.
//...
#endif/* SAMPLE_H */
```

Source File: An implementation file includes the specific details, that is the definitions, for what is done by the program

```
#include"sample.h"
//Definition of function
```

Friend Function and Friend Classes:

C++ programs are built in a two stage process. First, each source file is *compiled* on its own. The compiler generates intermediate files for each compiled source file. These intermediate files are often called *object files* -- but they are not to be confused with objects in your code. Once all the files have been individually compiled, it then *links* all the object files together, which generates the final binary (the program).

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "*friends*". *Friends* are functions or classes declared with the *friend* keyword.

Friend functions: A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword *friend*. The general form is

```
friend data_type function_name( ); // (inside the class)
```

A friend function is preceded by the keyword *friend*. Properties of *friend* functions are given below:

- *Friend* function is not in the scope of the class to which it has been declared as a *friend*. Hence it cannot be called using the object of that class.
- Usually it has objects as arguments.
- It can be declared either in the public or the private part of a class. It cannot access member names directly. It has to use an object name and dot membership operator with each member name. (e.g., A . x)

Let's take an example:

```
class Box{
double width;
public:
friend void printWidth( Box box );
void setWidth(double w) { width=w; }
};
void printWidth( Box box ){
cout << box.width << endl;
}
void main( ){
Box box;
box.setWidth(10.0);
printWidth( box );
}
```

Friend classes: Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class. A class can be made a friend of another class using keyword *friend*.

```
class A{
friend class B;      // class B is a friend class
...
}
class B{
...
}
```

When a class is made a friend class, all the member functions of that class becomes *friend* functions. In this program, all member functions of class B will be *friend* functions of class A. Thus, any member function of class B can access the private and protected data of class A.

If B is declared a *friend* class of A then, all member functions of class B can access private data and protected data of class A but, member functions of class A cannot private and protected data of class B. Remember, friendship relation in C++ is granted not taken.

Example: What should be the output of this program?

```
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j){
a = i;
b = j;
}
int sum(myclass x){
return x.a + x.b;
}
int main(){
myclass n;
n.set_ab(3, 4);
cout << sum(n);
return 0;
}
```

Exercise 1:(15 points)

Define a class **Student** that has the following attributes:

- Name: allocated dynamically by a character pointer.
- Roll no: an integer.
- Marks: dynamic array (double type).
- Percentage: a double

Include a constructor that takes values of the *Name*, *Rollno* and *Marks* from user as input. Also include the following methods:

CalculatePercentage: that adds all elements of array *Marks* and calculate percentage and stores the result in the member variable *Percentage*.

Grade: that calls *CalculatePercentage* method and displays the grade accordingly

Write a driver program to test your class.

Exercise 2:(15 points)

Create two classes **DM** and **DB** that store the value of distances. DM stores distance in *meters* and *centimeters* and DB in *feet* and *inches*.

Write a program that can read values for the class objects and add one object of DM with another object of DB. Use a *friend* function to carry out the addition operation. The object that stores the results maybe a DM object or DB object, depending on the units in which the results are required. The display should be in the format of feed and inches or meters and centimeters depending on the object on display.

Post Lab:(15 points)

Write a program with a class **integer** that contains an array of integers. Initialize the integer array in the constructor of the class. Then create *friend* functions to the class

- Find the largest integer in the array.
- Find the smallest integer in the array.
- Find the repeated elements in array.
- Sort the elements of array in ascending order.
- Create a destructor that sets all of the elements in the array to 0.

ExperimentNo. 08

Composition

OBJECTIVE:

Things that will be covered in today's lab:

- Composition

THEORY:

Composition (aggregation) is another way to relate two classes. In composition (aggregation), one or more members of a class are objects of another class type. Composition is a “has a” relation; for example “every person has a date of birth”.

```
class DOB
{
    // member variables/ functions
};
class person
{
    DOB date; // Declare object of date of birth
    // member variable/ functions
}student;
```

Access nested members:

- You can access the members of “*person*” class using dot operator.
- As “*date*” is the object of “*DOB*” which is a member of “*person*” class, so Members of “*DOB*” can be accessed using “*person*” class object.

```
// Access member variables
Student.person_members
Student.date.DOB_members
```

Now, let us discuss how the constructors of the objects of date are invoked. Recall that a class constructor is automatically executed when a class object enters its scope. Suppose that we have the following statement:

```
person student;
```

When the object “student” enters its scope, the objects “date”, which is the member of the student, also enters their scope. As a result, one of their constructors is executed. We, therefore, need to know how to pass arguments to the constructors of the member objects “date”, which occurs when we give the definitions of the constructors of the class. Recall that constructors do not have a type and so cannot be called like other functions. The arguments to the constructor of a member object are specified in the heading part of the definition of the constructor of the class.

Furthermore, member objects of a class are constructed (that is, initialized) in the order they are declared (not in the order they are listed in the constructor's member initialization list) and before the containing class objects are constructed. Thus, in our case, the object "date" is initialized first and then "student".

The following statements illustrate how to pass arguments to the constructors of the member objects DOB:

```
person::person(string f_n, string l_n, int m, int d, int y)
    :date(m, d, y)
{
    // Definition of Constructor person
}
```

Example: What should be the output of this program?

```
class Birthday
{
public:
    Birthday(int cmonth, int cday, int cyear){
        month = cmonth;
        day = cday;
        year = cyear;
    }
    void printDate(){ cout<<month<<"/"<<day <<"/"<<year;}
private:
    int month;
    int day;
    int year;
};
class People
{
public:
    People(string cname, Birthday cdateOfBirth)
        :name(cname),dateOfBirth(cdateOfBirth){}
    void printInfo(){
        cout<<name <<" was born on: ";
        dateOfBirth.printDate();
    }
private:
    string name;
    Birthday dateOfBirth;
};
int main()
{
    Birthday birthObject(7,9,97);
    People infoObject("Lenny the Cowboy", birthObject);
    infoObject.printInfo();
}
```

Exercise 1: (15 points)

Every Circle has a center and a radius. Create a class **CircleType** that can store the center, the radius, and the color of the circle. Since the center of a circle is a point in the x-y plane, create a class **PointType** to store the x and y coordinate. Use class *PointType* to define the class *CircleType*.

Provide *constructors* that enable objects of these classes to be initialized when they are declared. The constructors should contain default values in case no initializes are provided.

The definition of class *CircleType* and class *PointType* is as under: (you may define additional functions if you require any).

```
class PointType
{
    int x;
    int y;
public:
    //default constructor
    //constructor so that objects are initialized
    //print the x and y- coordinates
    //determine the quadrant in which the point lies
    //getter and setter functions
};
class CircleType
{
    double radius;
    char * color;
    PointType center;
public:
    //default constructor
    //constructor so that objects are initialized
    //print the radius, center, color
    //calculate area
    //calculate circumference
    //getter and setter functions
    //destructor
};
```

Your program should run for the following main.

```
int main()
{
    CircleType C(21,2,3.5,"blue");
    cout<<"\n*****\n"<<endl;
    C.print();
    cout<<"\n*****\n"<<endl;
    cout<<" Area of circle is  "<<C.calc_area();
    cout<<"\n\n*****\n\n"<<endl;
```

```

    PointType P(-20,3);
    int p=P.checkquad();
    P.print();

    switch (p)
    {
    case 0:
        cout<<"Point lies at center"<<endl;
        break;
    case 1:
        cout<<"Point lies in I quadrant"<<endl;
        break;
    case 2:
        cout<<"Point lies in II quadrant"<<endl;
        break;
    case 3:
        cout<<"Point lies in III quadrant"<<endl;
        break;
    case 4:
        cout<<"Point lies in IV quadrant"<<endl;
        break;
    default:
        cout<<"INVALID";
        break;
    }

    int x;
inty;
    double r;
    char col[9];

    CircleType circ(2,5,4.89,"purple");
    cout<<"\n\n*****\n\n";
    circ.print();

    cout<<"\n Enter radius \n";
    cin>>r;
    cout<<"\n Enter the coordinates where the center lies \n";
    cin>>x>>y;
    cout<<"\n Enter color \n";
    cin>>col;

    circ.setparam(x,y,r,col);
    cout<<"\n\n*****\n\n";
    circ.print();
    cout<<"\n\n*****\n\n";
}

```

Post Lab: (15 points)

Given the following two classes:

```
class course
{
    int courseNumber;
    int creditHours;
public:
    void setCourse (int x,int y);
};

class section
{
    int secNumber;
    course c;//composition
public:
    void setSection (int,int,int);
};
```

- Provide a meaningful implementation for class course and section.
- Write a main program that declares an array of 7 objects of type section and set their values:
 - Course number: 117 for all sections.
 - Credit hours: 3 for all sections.
 - Section number: give each section a unique number from 1-7.

ExperimentNo. 9

Inheritance

OBJECTIVE:

Things that will be covered in today's lab:

- Inheritance

THEORY:

Inheritance: The process by which a class incorporates the attributes and behaviors of a previously defined class. The new classes that we create from the existing classes are called *derived* classes and the existing classes are called *base* classes. Derived classes inherit the properties of base classes. Therefore, rather than creating completely new classes from scratch, we can take advantage of inheritance and reduce software complexity. Each derived class, in turn, becomes a base class for a future derived class. Inheritance can be either single inheritance or multiple inheritances. In single inheritance, the derived class is derived from a single base class; in multiple inheritances, the derived class is derived from more than one base class.

```
class derived-class: access-Specifier base-class
{
    //member list
};
```

Where *access-specifier* is one of **public**, **protected**, or **private**, and *base-class* is the name of a previously defined class. If the *access-specifier* is not used, then it is private by default.

When deriving a class from a base class, the base class may be inherited through *public*, *protected* or *private* inheritance. The type of inheritance is specified by the *access-specifier* as explained above.

We hardly use *protected* or *private* inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

Public Inheritance: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

Protected Inheritance: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

Private Inheritance: When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Multiple Inheritances:

C++ class can inherit members from more than one class and here is the extended syntax:

```
class derived-class: access base-a, access base-b...
{
    //member list
};
```

Where *access* is one of **public**, **protected**, or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

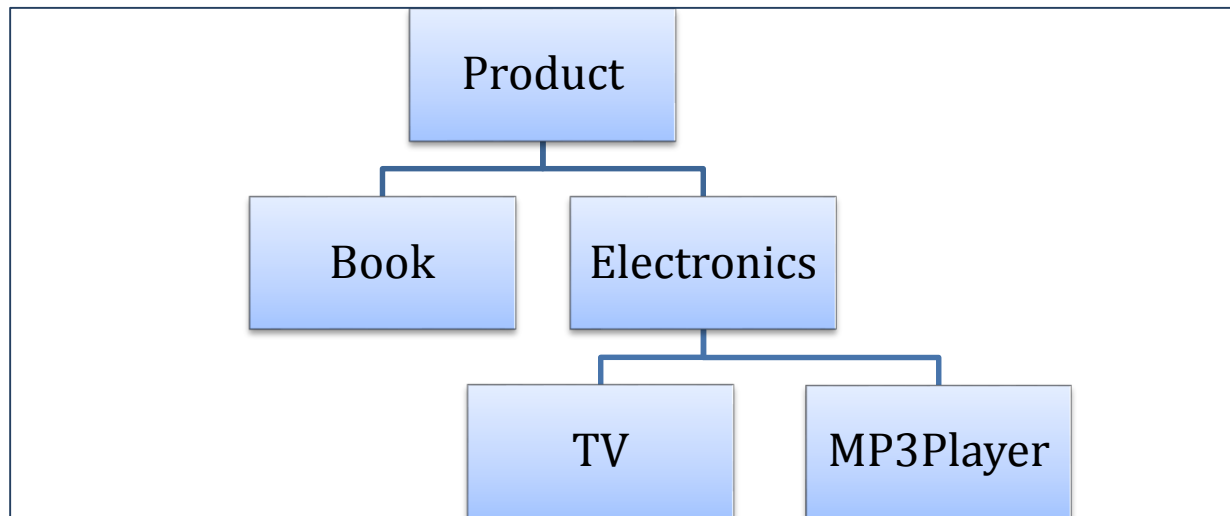
Example: What should be the output of this program?

```
class Shape
{
public:
void setWidth(int w) { width = w; }
void setHeight(int h) { height = h; }
private:
int width;
int height;
};
class Rectangle: public Shape
{
public:
int getArea() { return (width * height); }
};
int main()
{
    Rectangle Rect;
    Rect.setWidth(5);
    Rect.setHeight(7);

    cout <<"Total area: "<< Rect.getArea() << endl;
return 0;
}
```

Exercise 1: (15 points)

We want to store some details about an **online shop**. This shop has several kinds of products. Each product is represented by a class. The products are categorized into Electronics and Books. The Electronics is further subdivided into MP3Player and Television. The hierarchy is shown below:



For each product, system needs to store its regular price and discount. The system should provide a functionality to set or get a price for each product. For the products that are categorized as electronics, the name of the manufacturer should be stored. For MP3Players, system should store the *color* and *capacity* (in Giga Bytes). The system stores the size of each television. For each book, the following information is needed: Name of the book and Author's name. For each product, the system should calculate its discounted price.

Apply object oriented programming concepts and implement this system in C++. Implement setter and getter functions for each class and all other required functions. Use the following main().

```
void main() {
    int choice1=0,choice2=0,capacity,size;
    double price,discount;
    string manufacturer,color,author,name;

    cout<<"\n*****WELCOME TO ONLINE SHOP*****\n";
    cout<<"\nPress 1 for Electronics."<<"\nPress 2 for Books\n";
    cin>>choice1;
    if (choice1==1){
        cout<<"\nPress 1 for MP3Players."<<"\nPress 2 for TV\n";
        cin>>choice2;
        if (choice2==1) {
```



```

        cout<<"\nSet Attributes\n";
        cout<<"Price: ";
        cin>>price;
        cout<<"\nDiscount in %: ";
        cin>>discount;
        cout<<"\nManufacturer: ";
        cin>>manufacturer;
        cout<<"\nColor: ";
        cin>>color;
        cout<<"\nCapacity in GB: ";
        cin>>capacity;
        MP3player m1(price,discount,manufacturer,color,capacity);
        if (m1.getCapacity()==1)
            m1.setDiscount(10);
        else
            m1.setDiscount(50);
        cout<<m1.SalePrice();
    }
    elseif (choice2==2) {
        cout<<"\nSet Attributes\n";
        cout<<"Price: ";
        cin>>price;
        cout<<"\nDiscount in %: ";
        cin>>discount;
        cout<<"\nManufacturer: ";
        cin>>manufacturer;
        cout<<"\nSize: ";
        cin>>size;
        TV t1(price,discount,manufacturer,size);
        cout<<t1.SalePrice();
    }
    else
        cout<<"\nInvalid value... try again later";
}
elseif (choice1==2){
    cout<<"\nSet Attributes\n";
    cout<<"Price: ";
    cin>>price;
    cout<<"\nDiscount in %: ";
    cin>>discount;
    cout<<"\nAuthor: ";
    cin>>author;
    cout<<"\nName: ";
    cin>>name;
    Book b1(price, discount, author,name);
    cout<<b1.SalePrice();
}
else
    cout<<"\nInvalid value... try again later";
}

```

Post Lab:(15 points)

- a. A point in the x-y plane is represented by its x-coordinate and y-coordinate. Design a class, “*pointType*”, that can store and process a point in the x-y plane. You should then perform operations on the point, such as setting the coordinates of the point, printing the coordinates of the point, returning the x-coordinate, and returning the y-coordinate. Also, write a program to test various operations on the point.
- b. Every circle has a center and a radius. Given the radius, we can determine the circle’s area and circumference. Given the center, we can determine its position in the x-y plane. The center of the circle is a point in the x-y plane. Design a class, “*circleType*”, which can store the radius and center of the circle. Because the center is a point in the x-y plane and you designed the class to capture the properties of a point in Part a), you must derive the class “*circleType*” from the class “*pointType*”. You should be able to perform the usual operations on the circle, such as setting the radius, printing the radius, calculating and printing the area and circumference, and carrying out the usual operations on the center. Also, write a program to test various operations on a circle.
- c. Every cylinder has a base and height, wherein the base is a circle. Design a class, “*cylinderType*” that can capture the properties of a cylinder and perform the usual operations on the cylinder. Derive this class from the class “*circleType*” designed in Part b). Some of the operations that can be performed on a cylinder are as follows: calculate and print the volume, calculate and print the surface area, set the height, set the radius of the base, and set the center of the base. Also, write a program to test various operations on a cylinder.

ExperimentNo. 10

Polymorphism

OBJECTIVE:

Things that will be covered in today's lab:

- Polymorphism

THEORY:

Polymorphism is the term used to describe the process by which different implementations of a function can be accessed via the same name. For this reason, polymorphism is sometimes characterized by the phrase “one interface, multiple methods”.

In C++ polymorphism is supported both run time, and at compile time. Operator and function overloading are examples of compile-time polymorphism. Run-time polymorphism is accomplished by using inheritance and virtual functions.

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature.

```
class XYZ {
public:
void print() { cout<<"Parent class print:"<<endl; }
};
class ABC: public XYZ{
public:
void print() { cout<<"child class print:"<<endl; }
};
void main( )
{
    XYZ *xyz;
    ABC abc;
    xyz = &abc;           // store the address of abc
    xyz->print();          // call abc print .
}
```

When the above code is compiled and executed, it produces the following result:

```
Parent class print:
```

The reason for the incorrect output is that the call of the function *print()* is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the *print ()* function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of *print()* in the “abc” class with the keyword **virtual**. After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
child class print :
```

This time, the compiler looks at the contents of the pointer instead of its type. Hence, since address of object of “abc” class is stored in *xyz the respective *print()* function is called. As you can see, each of the child classes has a separate implementation for the function *print()*. This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function:

A virtual function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

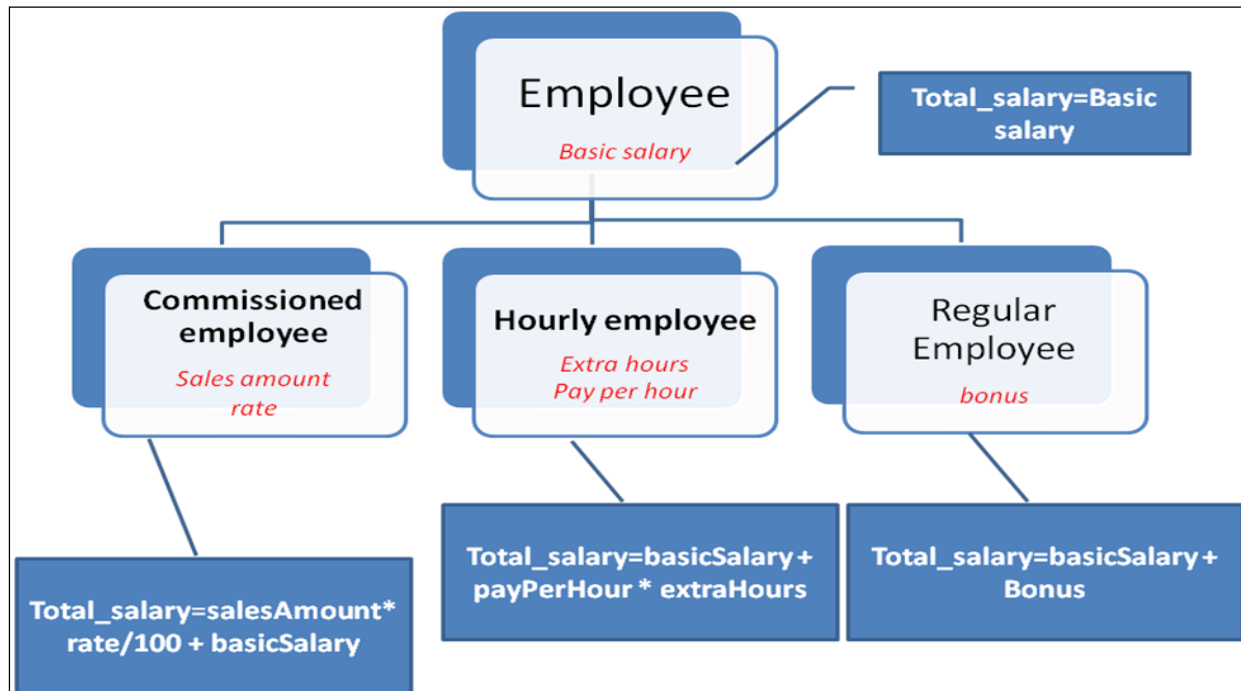
Example: What should be the output of this program?

```
class base{
public:
    virtual void who()      { cout<<"Base\n"; }
};
class first_d: public base{
public:
    void who()              { cout<<"First Derivation\n"; }
};
class second_d: public base{
public:
    void who()              { cout<<"Second Derivation\n"; }
};
int main() {
    base base_obj;
    base *p;
    first_d first_obj;
    second_d second_obj;
    p = &base_obj;
    p->who();
    p = &first_obj;
    p->who();
    p = &second_obj;
    p->who();
}
```

Exercise 1: (15 points)

We want to design a system for a company to calculate salaries of different types of employees.

Consider the following diagram:



Every employee has an employee ID and a basic salary. The Commissioned employee has a sales amount and rate. Hourly employee is paid on the basis of number of working hours. A regular employee may have a bonus.

You have to implement all the above classes. Write constructor for all classes. The main functionality is to calculate salary for each employee which is calculated as follows:

Commissioned Employee:	Total Salary=sales amount*rate/100+basic salary
Hourly Employee:	Total salary=basic salary + pay per hour*extra hours
Regular Employee:	Total salary= basic salary + bonus

You have to define the following function in all classes:

float calculateSalary() and run the given **main()** for the following two cases:

1. when the *calculateSalary()* in base class is not **virtual**
2. when the *calculateSalary()* in base class is made **virtual**

Use the following main().

```
int main()
{
    CommissionedEmployee E1(25, 5000, 1000, 10);

    // CASE 1 - derived Class Pointer pointing to Derived class object

    CommissionedEmployee * ptr;
    ptr = &E1;
    cout<<" Commissioned Employee salary:"<<ptr->calculateSalary();
    cout<<endl;

    // CASE 2 - Base Class Pointer pointing to Derived class object

    Employee * eptr;
    eptr = &E1;
    cout<<" Commissioned Employee salary:"<<eptr->calculateSalary();
    cout<<endl;

    CommissionedEmployee E2 (25, 5000, 1000, 10);
    CommissionedEmployee E3 (26, 5000, 2000, 10);

    HourlyEmployee H1(27, 5000, 10, 100 );
    HourlyEmployee H2(28, 5000, 5, 100 );

    RegularEmployee R1(29, 5000, 1000 );
    RegularEmployee R2(29, 5000, 2000 );

    Employee * list [6];
    list[0] = & E2;
    list[1] = & E3;
    list[2] = & H1;
    list[3] = & H2;
    list[4] = & R1;
    list[5] = & R2;

    for(int i = 0 ; i < 6; i++)
    {
        cout<<"Employee "<<i<<" salary is : "<<list[i]>calculateSalary();
        cout<<endl;
    }

    return 0;
}
```

Post Lab: (15 points)

Define a class **Shape** having an attribute *Area* and a pure virtual function *Calculate_Area*. Also include following in this class.

- A constructor that initializes Area to zero.
- A method *Display()* that display value of member variable.

Now derive two classes from Shape; **Circle** having attribute radius and **Rectangle** having attributes Length and Breadth. Include following in each class.

- A constructor that takes values of member variables as argument.
- A method *Display()* that overrides *Display()* method of Shape class.
- A method *Calculate_Area()* that calculates the area as follows:

Area of Circle= $\text{PI} * \text{Radius}^2$

Area of Rectangle=Length*Breadth

Use following driver program to test above classes.

```
int main()
{
    Shape *p;
    Circle C1(5);
    Rectangle R1(4,6);
    p=&C1;

    p->Calculate_Area();
    p->Display();

    p=&R1;
    p->Calculate_Area();
    p->Display();
    return 0;
}
```

ExperimentNo. 11

Operator Overloading

OBJECTIVE:

Things that will be covered in today's lab:

- Operator Overloading

THEORY:

Operator Overloading provides the ability to use the same operator to perform different actions. In C++ the statement `c = a + b` will compile successfully if `a`, `b` and `c` are of "int" and "float" types and if we attempt to compile the statement when `a`, `b` and `c` are the objects of user-defined classes, the compiler will generate error message but with operator overloading, this can happen.

Operator overloading is done with the help of a special function, called operator function, which defines the operation that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword *operator*.

Operator functions can be either members or nonmembers of a class. Non-member operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and nonmember functions. The general format of member operator function is:

Function prototype: (with in a class)

```
return_type operator op(arglist)
```

Function definition:

```
return_type class_name :: operator op(arglist)
{
    function body // task defined
}
```

You can also overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a friend function is not a member of the class, it does not have a *this* pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter. Insertion and extraction operators, Operator function must be a nonmember function of the class. Here is the syntax.

Function prototype :(with in a class)

```
friend ostream&operator<<(ostream&, const class_name&);  
friend istream&operator>>(istream&, const class_name&);
```

Function definition:

```
ostream&operator<<( ostream& out, const class_name& obj )  
{  
    // ...  
    return out;  
}  
istream&operator>>( istream&in, const class_name& obj )  
{  
    // ...  
    return in;  
}
```

We can overload the entire C++ operator except following.

1. Class member access operator (.)
2. Scope resolution operator (::)
3. Size operator (sizeof)
4. Conditional operator (? :)

Example: What should be the output of this program?

```
class Distance  
{  
private:  
int feet, inches;  
public:  
    Distance(int f, int i) { feet = f; inches = i; }  
void operator=(const Distance &D )  
{  
    feet = D.feet;  
    inches = D.inches;  
}  
void displayDistance() {  
    cout <<"F: " << feet <<" I:" << inches << endl;  
}  
};  
void main()  
{  
    Distance D1(11, 10), D2(5, 11);  
    D1 = D2;  
    cout <<"First Distance :";  
    D1.displayDistance();  
}
```

Exercise 1: (15 points)

You had implemented class for Complex numbers in lab 7. Now, write overloaded functions for the following operators

1. *(Multiplication)
2. +(Addition)
3. -(Subtraction)
4. /(Division)
5. =(Assignment)
6. == (Equality Comparator)

The Class definition is given below:

```
class comp
{
    double real;
    double imag;
public:
    // default constructor
    // function that set real and imag part of complex no
    // function that print complex number
    // Operator "+" for addition
    // Operator "-" for Subtraction
    // Operator "*" for Multiplication
    // Operator "/" for Division
    // Operator "==" for check both complex are equal or not
    // Operator "=" for assignment
};
```

Your program should run for the following main().

```
void main()
{
    comp n1,n2,n3;

    n1.setpara(2,3);
    n2.setpara(1,4);

    n3=n1;
    n3.show();

    (n1+n2).show();

    n3=n1*n2;
    n3.show();

    n3=n1-n2;
```

```
n3.show();

n3=n1/n2;
n3.show();

if(n1==n2)
    cout<<"Both no have same real and imag part "<<endl;
else
    cout<<"Unequal !!!"<<endl;

}
```

Exercise 2: (15 points)

Write a class implementation for a class named **PhoneNumber** giving the specification below:

Data that is associated with this class are:

- First name of type string.
- Last name of type string.
- Phone number of type string.

Functions:

- Constructor: that initializes any object of type “*PhoneNumber*”.
- Overloaded function for the insertion operator (<<) to print any object of type *PhoneNumber*.
- Overloaded function for extraction operator (>>) to read in for any object of type *PhoneNumber* all the values of its data members.

Write a driver program that test the class as follow:

- Declare an object of type “*PhoneNumber*”, read in its values, and then print it (using operator<<, operator>>).
- Declare an array of three objects of type “*PhoneNumber*”, read in their values and then print their values (using operator<<, operator>>).

Post Lab: (20 Marks)

Write a class **Rational** which represents a numerical value by two double values *Numerator* and *Denominator*. Include the following public member functions:

- Constructor with no arguments (default).
- Constructor with two parameters.
- Reduce() function to reduce the rational number by eliminating the highest common factor between the numerator and the denominator.
- Overload the addition, subtraction, multiplication and division operators for this class
 - + (Addition)
 - – (Subtraction)
 - x (Multiplication)
 - /(Division)
- Overload >> operator to enable input through in.
- Overload << operator to enable output through out.
- Overload the relational and equality operators for this class.
 - <(Less than)
 - >(Greater than)
 - <=(Less than or equal to)
 - >=(Greater than or equal to)
 - !=(Not equal)
 - ==(Equal to)
- Overload pre-increment, pre-decrement, post-increment and post-decrement operator if denominator in a rational number is “1”.

Experiment No. 12

Templates

OBJECTIVE:

Things that will be covered in today's lab:

- Templates

THEORY:

Templates are a very powerful feature of C++. They allow you to write a single code segment for a set of related functions, called a function template, and for a set of related classes, called a class template. The syntax we use for templates is:

```
template<class Type>
declaration;
```

Function templates: A function template behaves like a function except that the template can have arguments of many different types. In other words; a function template represents a family of functions. The general format of a function template is as follows:

```
template<class T>
return_type functionname(argument T )
{
// body of function with Type T
};
```

Class templates: A class template provides a specification for generating classes based on parameters. Class templates are generally used to implement [containers](#).

```
template <class type>
class class_name
{
// (Body of the class)
};
```

Let's take an example, just declare an "integer" generic class. This class contains two member variables which are of Type T, and a member function "greater" to return greater number.

```
template<class T>
class number
{
T no_1, no_2;
public:
number (T n1, T n2)    { no_1=n1; no_2=n2;    }
T greater();
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type “int” with the values 115 and 36 we would write:

```
number<int> myobject1(1,2);
```

This same class could also be used to create an object to store any other type, such as:

```
number<double> myobject2(1.2,2.2);
```

If a member function is defined outside the definition of the class template, it shall be preceded with the template < . . . > prefix:

Notice the syntax of the definition of a member function “greater” is:

```
template<class T>
T number<T>::greater()
{
    // body of greater definition
}
```

Example: What should be the output of this program?

```
template<class T>
class number
{
    T no_1, no_2;
public:
    number (T n1, T n2)    { no_1=n1; no_2=n2;    }
    T greater();
};

template<class T>
T number<T>::greater()
{
    if(no_1>no_2)
        return no_1;
    else
        return no_2;
}

void main()
{
    number<int> myobject1(1,2);
    cout<<myobject1.greater()<<endl;

    number<double> myobject2(1.2,2.2);
    cout<<myobject2.greater()<<endl;
}
```

Exercise 1:**(10 points)**

Study the `myMAX` function provided below. You are required to create a C++ template based `myMAX` function and test it on different built-in data types.

```
//Make a template out of this function. Don't forget the return type.
int myMax(int one, int two)
{
    int bigger;
    if (one < two)
        bigger = two;
    else
        bigger = one;
    return bigger;
}
int main()
{
    int i_one = 3, i_two = 5;
    cout <<"The max of "<< i_one <<" and "<< i_two <<" is "
    << myMax(i_one, i_two) << endl;
    //Test your template on float and string types
    return 0;
}
```

Exercise 2:**(10 points)**

Write a C++ generic bubblesort function using templates that takes as parameters an array and its size. You may assume that the compared objects have a `<` operator already defined. Test your template function with a **built-in** type and a **user-defined** type.

Exercise 3:**(10 points)**

Consider the class of points in the xy plane. The location of each point is determined by the real numbers (x, y) specifying the cartesian coordinates. The class definition is:

```
#include<iostream>
usingnamespace std;
class point{
public:
    point();
    point(double value_x, double value_y);
    double get_x() const;
    double get_y() const;
    void print() const;
    void move(double dx, double dy);
private:
    double x, y;
};
point::point(){
    x = 0.0; y = 0.0;
```

```

}
point::point(double a, double b){
    x = a; y = b;
}
void point::print() const{
    cout<<x<<" "<<y<< endl;
}
double point::get_x() const{
    return x;
}
double point::get_y() const{
    return y;
}
void point::move(double dx, double dy){
    x = x+dx;
    y = y+dy;
}

```

Generalize the class Point into a template and test your code using following `main` function.

```

int main()
{
    point<int> A = point<int>(1, 2);
    A.print();
    A.move(4, -5);
    A.print();

    point<float>B(3.2, 4.9);
    cout << B.get_x() <<" "<< B.get_y() << endl ;
    point<string> C("day", "young");
    C.print();
        C.move("s", "ster");
        C.print();
    return 0;
}

```

ExperimentNo. 13

Project Session

ExperimentNo. 14

Project Session

Appendix A:

Lab Evaluation Criteria

1. Experiments and their reports	50%
a. Experiment	60%
b. Lab report	40%
2. Quizzes (3-4)	15%
3. Final evaluation	35%
a. ProjectImplementation	60%
b. Project report and quiz	40%

Notice:

Copying and plagiarism of lab reports is a serious academic misconduct. First instance of copying may entail ZERO in that experiment. Second instance of copying may be reported to DC. This may result in awarding FAIL in the lab course.

Appendix B:

Safety around Electricity

In all the Electrical Engineering (EE) labs, with an aim to prevent any unforeseen accidents during conduct of lab experiments, following preventive measures and safe practices shall be adopted:

- Remember that the voltage of the electricity and the available electrical current in EE labs has enough power to cause death/injury by electrocution. It is around 50V/10 mA that the “cannot let go” level is reached. “The key to survival is to decrease our exposure to energized circuits.”
- If a person touches an energized bare wire or faulty equipment while grounded, electricity will instantly pass through the body to the ground, causing a harmful, potentially fatal, shock.
- Each circuit must be protected by a fuse or circuit breaker that will blow or “trip” when its safe carrying capacity is surpassed. If a fuse blows or circuit breaker trips repeatedly while in normal use (not overloaded), check for shorts and other faults in the line or devices. Do not resume use until the trouble is fixed.
- It is hazardous to overload electrical circuits by using extension cords and multi-plug outlets. Use extension cords only when necessary and make sure they are heavy enough for the job. Avoid creating an “octopus” by inserting several plugs into a multi-plug outlet connected to a single wall outlet. Extension cords should ONLY be used on a temporary basis in situations where fixed wiring is not feasible.
- Dimmed lights, reduced output from heaters and poor monitor pictures are all symptoms of an overloaded circuit. Keep the total load at any one time safely below maximum capacity.
- If wires are exposed, they may cause a shock to a person who comes into contact with them. Cords should not be hung on nails, run over or wrapped around objects, knotted or twisted. This may break the wire or insulation. Short circuits are usually caused by bare wires touching due to breakdown of insulation. Electrical tape or any other kind of tape is not adequate for insulation!
- Electrical cords should be examined visually before use for external defects such as: Fraying (worn out) and exposed wiring, loose parts, deformed or missing parts, damage to outer jacket or insulation, evidence of internal damage such as pinched or crushed outer jacket. If any defects are found the electric cords should be removed from service immediately.
- Pull the plug not the cord. Pulling the cord could break a wire, causing a short circuit.
- Plug your heavy current consuming or any other large appliances into an outlet that is not shared with other appliances. Do not tamper with fuses as this is a potential fire hazard. Do not overload circuits as this may cause the wires to heat and ignite insulation or other combustibles.
- Keep lab equipment properly cleaned and maintained.
- Ensure lamps are free from contact with flammable material. Always use lights bulbs with the recommended wattage for your lamp and equipment.
- Be aware of the odor of burning plastic or wire.

- ALWAYS follow the manufacturer recommendations when using or installing new lab equipment. Wiring installations should always be made by a licensed electrician or other qualified person. All electrical lab equipment should have the label of a testing laboratory.
- Be aware of missing ground prong and outlet cover, pinched wires, damaged casings on electrical outlets.
- Inform Lab engineer / Lab assistant of any failure of safety preventive measures and safe practices as soon you notice it. Be alert and proceed with caution at all times in the laboratory.
- Conduct yourself in a responsible manner at all times in the EE Labs.
- Follow all written and verbal instructions carefully. If you do not understand a direction or part of a procedure, ASK YOUR LAB ENGINEER / LAB ASSISTANT BEFORE PROCEEDING WITH THE ACTIVITY.
- Never work alone in the laboratory. No student may work in EE Labs without the presence of the Lab engineer / Lab assistant.
- Perform only those experiments authorized by your teacher. Carefully follow all instructions, both written and oral. Unauthorized experiments are not allowed.
- Be prepared for your work in the EE Labs. Read all procedures thoroughly before entering the laboratory. Never fool around in the laboratory. Horseplay, practical jokes, and pranks are dangerous and prohibited.
- Always work in a well-ventilated area.
- Observe good housekeeping practices. Work areas should be kept clean and tidy at all times.
- Experiments must be personally monitored at all times. Do not wander around the room, distract other students, startle other students or interfere with the laboratory experiments of others.
- Dress properly during a laboratory activity. Long hair, dangling jewelry, and loose or baggy clothing are a hazard in the laboratory. Long hair must be tied back, and dangling jewelry and baggy clothing must be secured. Shoes must completely cover the foot.
- Know the locations and operating procedures of all safety equipment including fire extinguisher. Know what to do if there is a fire during a lab period; “Turn off equipment, if possible and exit EE lab immediately.”

Appendix C:

Guidelines on preparing Lab Reports

Each student will maintain a lab notebook for each lab course. He will write a report for each post lab he performs in his notebook. A format has been developed for writing these lab reports. Separate formats are devised for hardware and programming stream labs.

Programming Stream Lab Report Format

For programming streams, the format of the report will be as given below:

1. **Introduction:** Introduce the new constructs/ commands being used, and their significance.
2. **Objective:** What are the learning goals of the experiment?
3. **Design:** If applicable, draw the flow chart for the program. How do the new constructs facilitate achievement of the objectives; if possible, a comparison in terms of efficacy and computational tractability with the alternate constructs?
4. **Issues:** The bugs encountered and the way they were removed.
5. **Conclusions:** What conclusions can be drawn from experiment?
6. **Application:** Suggest a real world application where this exercise may apply.
7. Answers to post lab questions (if any).

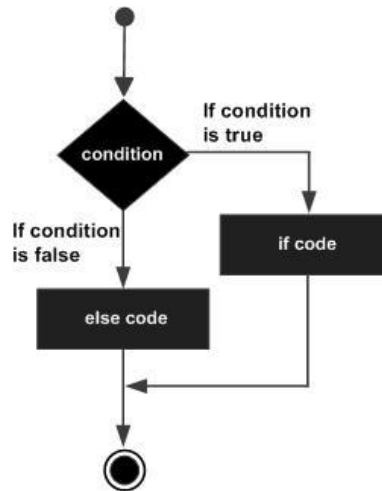
Sample Lab Report for Programming Labs

Introduction

The ability to control the flow of the program, letting it make decisions on what code to execute, is important to the programmer. The if-else statement allows the programmer to control if a program enters a section of code or not based on whether a given condition is true or false. If-else statements control *conditional branching*.

```
if ( expression )  
    statement1  
else  
    statement2
```

If the value of *expression* is nonzero, *statement1* is executed. If the optional **else** is present, *statement2* is executed if the value of *expression* is zero. In this lab, we use this construct to select an action based upon the user's input, or a predefined parameter.



Objective:

To use if-else statements for facilitation of programming objectives: A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. We have written a C++ program that reads in a five-digit integer and determines whether it is a palindrome.

Design:

The objective was achieved with the following code:

```

#include<iostream>

usingnamespace std;
int main()
{
int i,temp,d,revrs=0;

    cout<<"enter the number to check :";
    cin>>i;
    temp=i;
while(temp>0)
{
    d=temp%10;
    temp/=10;
    revrs=revrs*10+d;

}
if(revrs==i)
    cout<<i<<" is palindorme";
else
    cout<<i<<" is not palindrome";

}
}
  
```

Screen shots of the output for various inputs are shown in Figure 1:

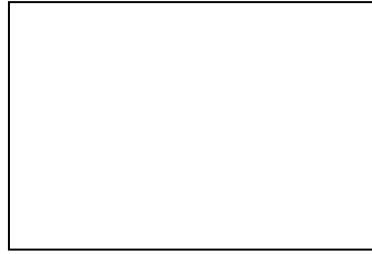


Fig.1. Screen shot of the output

The conditional statement made this implementation possible; without conditional branching, it is not possible to achieve this objective.

Issues:

Encountered bugs and issues; how were they identified and resolved.

Conclusions:

The output indicates correct execution of the code.

Applications:

If-else statements are a basic construct for programming to handle decisions.