IMS Engineering College, Ghaziabad

Department of Computer Science and Engineering

Session 2016-17 (Odd Sem)

LAB MANUAL

ADVANCE PROGRAMMING LAB (NCS-355)

IMS Engineering College, Ghaziabad
Department of Computer Science & Engineering
Session 2016-17

Subject Name: Advance Programming Lab Subject Code: NCS-355

Year and Branch: 2nd yr/CSE

As Per the University Syllabus:

Experiment No	Name /Objectives	Outcomes	Co-relation with POs	Co-relation with PSOs
1	**Aim:** Programs using Functions and Pointers in C **Objective:** To make students understand implementation of Functions and Pointers in C	Student will be able to understand concept of pointer.	1,2,3,4,5,6,11,12	1,2,3,4
2	**Aim:** Programs using Files in C **Objective:** To make students understand implementation of Files in C	Student will be able to implement file handling in C	1,2,3,4,5,6,11,12	1,2,3,4
3	**Aim:** Programs using Classes and Objects. **Objective:** To make students understand implementation of Classes and Objects.	Student will be able to understand class and object.	1,2,3,4,5,6,11,12	1,2,3,4
4	**Aim:** Programs using Operator Overloading **Objective:** To make students understand implementation of Operator Overloading	Student will be able to understand different overloading operators	1,2,3,4,5,6,11,12	1,2,3,4
5	**Aim:** Programs using Inheritance, Polymorphism and its types **Objective:** To make the students understand the concept of Inheritance, Polymorphism and its types	Student will be able to understand OOPS Concept	1,2,3,4,5,6,11,12	1,2,3,4
6	**Aim:** Programs using Arrays and Pointers. **Objective:** To enable the students to learn the concepts of arrays	Student will be able to understand array in pointer.	1,2,3,4,5,6,11,12	1,2,3,4

	and Pointers.			
7	**Aim:** Programs using Dynamic memory allocation **Objective:** To enable the students to learn the concepts of Dynamic Memory Allocation.	Students can write Memory allocation programs	1,2,3,4,5,6,11,12	1,2,3,4
8	**Aim:** Program using template and exceptions **Objective:** To enable the students to learn the concepts of template and exceptions.	Students can understand exception handling.	1,2,3,4,5,6,11,12	1,2,3,4
9	**Aim:** Programs using Sequential and Random access files **Objective:** To enable the students to learn the concepts of sequential and random access file.	Students can understand file handling in C++.	1,2,3,4,5,6,11,12	1,2,3,4

Experiments beyond syllabus

Experiment No	**Objectives#**	**Outcomes**	**Co-relation with POs**	**Co-relation with PSOs**
1	Program Using Constructor and destructore	Students can get basic idea about Java programming	1,2,3,4,5,6,11,12	1,2,3,4
2	Program using friend function		1,2,3,4,5,6,11,12	1,2,3,4
3	Program using virtual function		1,2,3,4,5,6,11,12	1,2,3,4

Course Name: ADVANCE PROGRAMMING LAB (NCS -355)

Year of Study: 2nd Yr (IIIRD Sem.)

Course Outcomes:

NCS – 355	<Statement>
NCS – 355.1	Understanding C Language (Advanced)
NCS – 355.2	Understanding Advanced C++
NCS – 355.3	Understand Array and Structures concepts.
NCS – 355.4	Understand Pointers.

Mapping of Course Outcome with Program Outcome:

Course Outcome	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
NCS -355.1	2	2	3	1	-	1	-	-	-	-	1	3
NCS -355.2	2	2	3	2	-	1	-	-	-	-	1	3
NCS -355.3	2	2	3	1	-	1	-	-	-	-	1	3
NCS -355.4	2	2	3	2	-	1	-	-	-	-	1	3

Mapping of Course Outcome with Program Specific Outcome:

Course Outcome	PSO1	PSO2	PSO3	PSO4
NCS -355.1	3	2	1	1
NCS -355.2	3	2	2	1
NCS -355.3	3	2	2	1
NCS -355.4	3	2	2	1

Faculty Name:

Signature:

Experiment -01

Objective:Program Function and pointer in C

Function-A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Defining a Function

The general form of a function definition in C programming language is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function −

Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − The function body contains a collection of statements that define what the function does.

Calling a Function

1-Call by value

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

2-Call by reference

This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Example:WAP in C find out maximum value among two numbers.

Algorithm

step 1 : start

step 2 : input number num1 & num2

step 3 : if (num1>num2)

print "number even"

else

print "number odd"

endif

step 4 : stop

Code:

```
/* function returning the max between two numbers */
#include<stdio.h>
#include<conio.h>
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
  return result;
}
```

Output- Enter any two number

10

20

Num2=20 is greater

Real Time Aplication-1.C functions are used to avoid rewriting same logic/code again and again in a program.

2.There is no limit in calling C functions to make use of same functionality wherever required.

3.We can call functions any number of times in a program and from any place in a program.

4.A large C program can easily be tracked when it is divided into functions.

Pointer

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

type *var-name;

int *ip; /* pointer to an integer */

double *dp; /* pointer to a double */

float *fp; /* pointer to a float */

char *ch /* pointer to a character */

	Concept & Description
1	Pointer arithmetic

There are four arithmetic operators that can be used in pointers: ++, --, +, -

2	Array of pointers

You can define arrays to hold a number of pointers.

3	Pointer to pointer

C allows you to have pointer on a pointer and so on.

4	Passing pointers to functions in C

Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.

5	Return pointer from functions in C

C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

Ex-Code for Program to sort numbers in ascending order and use integer pointer to store numbers in C Programming

Algorithm:

Here we will use basic algorithm to sort arrays in ascending order:

Step 1: Read elements in array.

Step 2: Set i=0

Step 3: Set j=i+1

Step 4: If array[i] > array[j] then swap value of array[i] and array[j].

Step 5: Set j=j+1

Step 6: Repeat Step 4-5 till j<n (Where n is the size of the array)

Step 7: Set i=i+1

Step 8: Repeat Step 3-7 till i<n

Code:

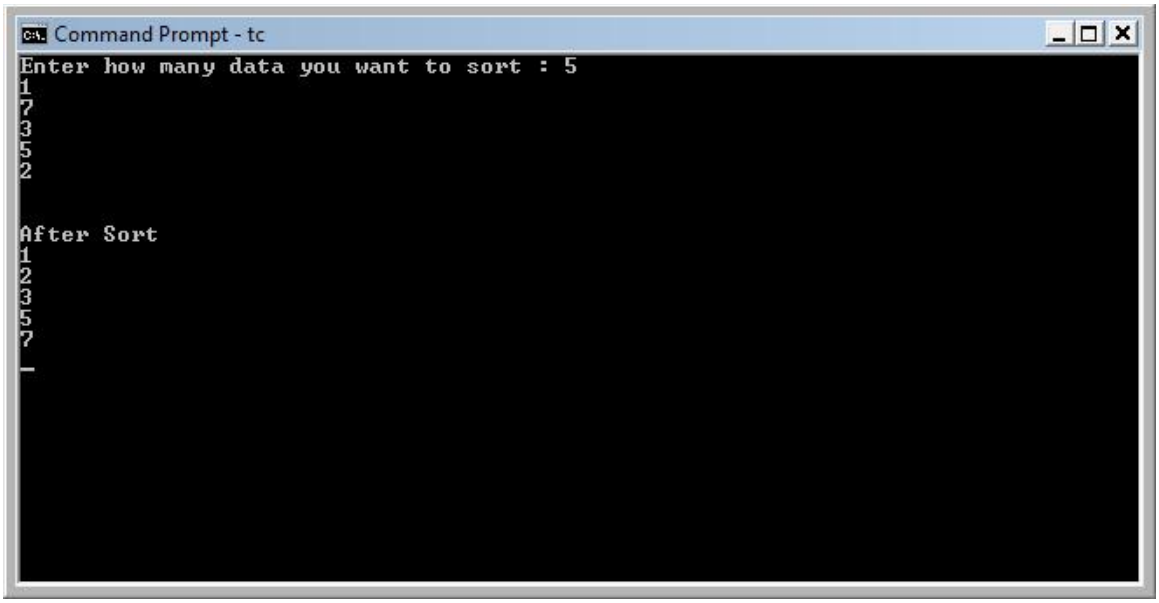
```
#include <stdio.h>
#include <conio.h>

void main(){
  int *arr,i,j,tmp,n;
  clrscr();
  printf("Enter how many data you want to sort : ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
    scanf("%d",arr+i);

  for(i=0;i<n;i++)
  {
   for(j=i+1;j<n;j++){
     if( *(arr+i) > *(arr+j)){
    tmp = *(arr+i);
    *(arr+i) = *(arr+j);
    *(arr+j) = tmp;
     }
   }
  }
```

```
    printf("\n\nAfter Sort\n");
    for(i=0;i<n;i++)
        printf("%d\n",*(arr+i));

        getch();
}
```



Real Time Application-1. Easy access

2.To return more than one value from a function.

3. To pass as arguments to functions.

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment -02

Objective: Program using file In C

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.

Opening Files

You can use the fopen( ) function to create a new file or to open an existing file. This call will initialize an object of the type FILE, which contains all the information necessary to control the stream. The prototype of this function call is as follows −

FILE *fopen(const char * filename, const char * mode);

Here, filename is a string literal, which you will use to name your file, and access mode can have one of the following values −

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is −

int fclose(FILE *fp);

Writing a File

int fputc(int c, FILE *fp);

The function fputc() writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error.

Reading a File

int fgetc(FILE * fp);

The fgetc() function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns EOF. The following function allows to read a string from a stream −

char *fgets(char *buf, int n, FILE *fp);

The functions fgets() reads up to n-1 characters from the input stream referenced by fp

Program:C Program to Append the Content of File at the end of Another

ALGORITHM:

STEP 1: Start the program.

STEP 2: Open File1 and File2 in read mode.

STEP 3: Open File3 in write mode.

STEP 4: while ((ch = fgetc(fsring1)) != EOF)

fputc(ch, ftemp);

while ((ch = fgetc(fsring2)) != EOF)

fputc(ch, ftemp);.

STEP 5: writing the file contents up to reach a particular condition.

STEP 6: Stop the program.

Code:

```
#include <stdio.h>

#include <stdlib.h>


main()

{

   FILE *fsring1, *fsring2, *ftemp;

   char ch, file1[20], file2[20], file3[20];
```

```
printf("Enter name of first file ");
gets(file1);
printf("Enter name of second file ");
gets(file2);
printf("Enter name to store merged file ");
gets(file3);
fsring1 = fopen(file1, "r");
fsring2 = fopen(file2, "r");
if (fsring1 == NULL || fsring2 == NULL)
{
   perror("Error has occured");
   printf("Press any key to exit...\n");
   exit(EXIT_FAILURE);
}
ftemp = fopen(file3, "w");
if (ftemp == NULL)
{
   perror("Error has occures");
   printf("Press any key to exit...\n");
   exit(EXIT_FAILURE);
}
while ((ch = fgetc(fsring1)) != EOF)
   fputc(ch, ftemp);
while ((ch = fgetc(fsring2) ) != EOF)
   fputc(ch, ftemp);
printf("Two files merged  %s successfully.\n", file3);
fclose(fsring1);
fclose(fsring2);
fclose(ftemp);
```

```
    return 0;
}
```

Output: Enter name of first file a.txt

Enter name of second file b.txt

Enter name to store merged file merge.txt

Two files merged merge.txt successfully.

Real Time Application:1. Platform-specific identifier of the associated I/O device, such as a file descriptor

2.stream orientation indicator (unset, narrow, or wide)

3.stream buffering state indicator (unbuffered, line buffered, fully buffered)

4.I/O mode indicator (input stream, output stream, or update stream)

5.binary/text mode indicator

6.end-of-file indicator

7.error indicator

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment-03

Objective: Program Using Classes And Objects

Class:A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword class as follows:

```
class Box

{

  public:

    double length;   // Length of a box

    double breadth;  // Breadth of a box

    double height;   // Height of a box

};
```

The keyword public determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in a sub-section.

Objects:A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1;        // Declare Box1 of type Box

Box Box2;        // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear:

```
 // box 1 specification

  Box1.height = 5.0;

  Box1.length = 6.0;

  Box1.breadth = 7.0;
```

Class member functions

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.

Class access modifiers

A class member can be defined as public, private or protected. By default members would be assumed as private.

Constructor & destructor

A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.

C++ copy constructor

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

C++ friend functions

A friend function is permitted full access to private and protected members of a class.

C++ inline functions

With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.

The this pointer in C++

Every object has a special pointer this which points to the object itself.

Program-C++ program to print student details using constructor and destructor

ALGORITHAM:

Step 1. Start the process

Step 2. Invoke the classes

Step 3. Call the read() function

a. Get the inputs name ,roll number and address

Step 4. Call the display() function

a. Display the name,roll number,and address of the student

Step 5. Stop the process

Code:

```
#include<iostream.h>

#include<conio.h>

class stu

{
```

```
                private: char name[20],add[20];
                        int roll,zip;
                public: stu ( );//Constructor
                        ~stu( );//Destructor
                        void read( );
                        void disp( );
};
stu :: stu( )
{
        cout<<"This is Student Details"<<endl;
}
void stu :: read( )
{
        cout<<"Enter the student Name";
        cin>>name;
        cout<<"Enter the student roll no ";
        cin>>roll;
        cout<<"Enter the student address";
        cin>>add;
        cout<<"Enter the Zipcode";
        cin>>zip;
}
void stu :: disp( )
{
        cout<<"Student Name :"<<name<<endl;
        cout<<"Roll no   is      :"<<roll<<endl;
        cout<<"Address is       :"<<add<<endl;
        cout<<"Zipcode is       :"<<zip;
```

```
}
stu : : ~stu( )
{
        cout<<"Student Detail is Closed";
}

void main( )
{
        stu s;
        clrscr( );
s.read ( );
s.disp ( );
getch( );
}
```

Output:

```
Enter the student Name
James
Enter the student roll no
01
Enter the student address
Newyork
Enter the Zipcode
919108
Student Name : James
Roll no is : 01
Address is : Newyork
Zipcode is :919108
```

Real Time Application: 1.structures in c++ doesn't provide data hiding where as a class provides data hiding

2. classes support polymorphism(late binding), whereas structures don't

3.class and structure are very similar. the former is heavyweight while the latter is light weight. reference to the former rests on the heap..while the latter in whole (instance and data) rests on the stack. therefor care should be taken not to make a struct very heavy else it overloads the stack causing memory hogging. class needs to have an instance explicitly created to be used. A struct doesn't have to be explicitly initiated.

4.The "this" pointer will works only with class.

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment-04

Objective:Programs using Operator Overloading

Theory:Operator Overloading:

- An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
- When calling overloaded **function operator**, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions.electing the most appropriate overloaded function or operator is called **overload resolution**
- Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

To write a program to add two complex numbers using binary operator overloading.

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the class.

Step 3: Declare the variables and its member function.

Step 4: Using the function getvalue() to get the two numbers.

Step 5: Define the function operator +() to add two complex numbers.

Step 6: Define the function operator –()to subtract two complex numbers.

Step 7: Define the display function.

Step 8: Declare the class objects obj1,obj2 and result.

Step 9: Call the function getvalue using obj1 and obj2

Step 10: Calculate the value for the object result by calling the function operator + and operator -.

Step 11: Call the display function using obj1 and obj2 and result.

Step 12: Return the values.

Step 13: Stop the program.

PROGRAM:

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

```
class complex
{
        int a,b;
        public:
void getvalue()
cout<<"Enter the value of Complex Numbers a,b:";
cin>>a>>b;
};
complex operator+(complex ob)
{
complex t;
t.a=a+ob.a;
t.b=b+ob.b;
return(t);
complex operator-(complex ob)
{
complex t;
t.a=a-ob.a;
t.b=b-ob.b;
return(t);
}
void display()
{
cout<<a<<"+"<<b<<"i"<<"\n";
}
};

  void main()
{
 clrscr();
 complex obj1,obj2,result,result1;

 obj1.getvalue();
 obj2.getvalue();
```

```
 result = obj1+obj2;
 result1=obj1-obj2;

 cout<<"Input Values:\n";
 obj1.display();
 obj2.display();

 cout<<"Result:";
 result.display();
result1.display();

getch();
}
```

Output:

```
Enter the value of Complex Numbers a, b
4  5
Enter the value of Complex Numbers a, b
2  2
Input Values
4 + 5i
2 + 2i
Result
6 +  7i
2 + 3i
```

Applications

The follwing operators can be overloaded:

- Assignment Operator
- Input and Output Operators
- Function call operator
- Comparison operators
- Arithmetic Operators
- Array Subscripting
- Operators for Pointer-like Types

- Conversion Operators
- Overloading new and delete

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment 5 :

Objective:Programs using Inheritance, Polymorphism and its types

Theory:C++ Inheritance:

allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base**class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

To write a program to find out the payroll system using single inheritance.

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the base class emp.

Step 3: Define and declare the function get() to get the employee details.

Step 4: Declare the derived class salary.

Step 5: Declare and define the function get1() to get the salary details.

Step 6: Define the function calculate() to find the net pay.

Step 7: Define the function display().

Step 8: Create the derived class object.

Step 9: Read the number of employees.

Step 10: Call the function get(),get1() and calculate() to each employees.

Step 11: Call the display().

Step 12: Stop the program.

PROGRAM:PAYROLL SYSTEM USING SINGLE INHERITANCE

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


class emp
{
 public:
```

```
 int eno;
 char name[20],des[20];
 void get()
 {
cout<<"Enter the employee number:";
cin>>eno;
cout<<"Enter the employee name:";
cin>>name;
cout<<"Enter the designation:";
cin>>des;
 }
};

    class salary:public emp
{
float bp,hra,da,pf,np;
public:
void get1()
 {
cout<<"Enter the basic pay:";
cin>>bp;
cout<<"Enter the Humen Resource Allowance:";
cin>>hra;
cout<<"Enter the Dearness Allowance :";
cin>>da;
cout<<"Enter the Profitablity Fund:";
cin>>pf;
}
 void calculate()
 {
np=bp+hra+da-pf;
 }
 void display()
 {
cout<<eno<<"\t"<<name<<"\t"<<des<<"\t"<<bp<<"\t"<<hra<<"\t"<<da<<"\t"<<pf<<"\t"<<
np<<"\n";
```

```
 }
};

void main()
{
int i,n;
char ch;
salary s[10];
clrscr();
cout<<"Enter the number of employee:";
cin>>n;
for(i=0;i<n;i++)
{
s[i].get();
s[i].get1();
s[i].calculate();
}
cout<<"\ne_no \t e_name\t des \t bp \t hra \t da \t pf \t np \n";
for(i=0;i<n;i++)
{
s[i].display();
}
getch();
}
```

Output:

Enter the Number of employee:1
Enter the employee No: 150

Enter the employee Name: ram

Enter the designation: Manager

Enter the basic pay: 5000

Enter the HR allowance: 1000

Enter the Dearness allowance: 500

Enter the profitability Fund: 300

Application:

Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably. If the return type of a method is superclass

Reusability -- facility to use public methods of base class without rewriting the same
Extensibility -- extending the base class logic as per business logic of the derived class
Data hiding -- base class can decide to keep some data private so that it cannot be altered by the derived class

Overriding--With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

(b.) Polymorphism

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

```
#include <iostream>
using namespace std;

class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    int area()
    {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape{
```

```
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
        cout << "Rectangle class area :" <<endl;
        return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
        cout << "Triangle class area :" <<endl;
        return (width * height / 2);
      }
};
// Main function for the program
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();

   return 0;
}
```

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

```
Parent class area
Parent class area
```

The reason for the incorrect output is that the call of the function area() 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 area() 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 area() in the Shape class with the keyword **virtual** so that it looks like this:

```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    virtual int area()
    {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). 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**.

Pure Virtual Functions:

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    // pure virtual function
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment: 6

Objective:Programs using Arrays and Pointers

C - Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

```
balance[4] = 50.0;
```

The above statement assigns the 5^{th} element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

```
#include <stdio.h>

int main () {

   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ ) {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
```

Element[8] = 108
Element[9] = 109

(b) Pointers

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined −

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

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

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type**is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently.**(a)**We define a pointer variable,**(b)**assign the address of a variable to a pointer and**(c)**finally access the value at the address available in the pointer variable. This is done by using unary operator*****that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

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

```
int main () {

   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

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

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment: 7

Objective:Programs using Dynamic Memory Allocation

Theory:

Allocating memory

There are two ways that memory gets allocated for data storage:

1. Compile Time (or static) Allocation
    - Memory for named variables is allocated by the compiler
    - Exact size and type of storage must be known at compile time
    - For standard array declarations, this is why the size has to be constant
2. Dynamic Memory Allocation
    - Memory allocated "on the fly" during run time
    - dynamically allocated space usually placed in a program segment known as the *heap* or the *free store*
    - Exact amount of space or number of items does not have to be known by the compiler in advance.
    - For dynamic memory allocation, pointers are crucial

Dynamic Memory Allocation

- We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"
- For this reason, dynamic allocation requires two steps:
    1. Creating the dynamic space.
    2. Storing its **address** in a **pointer** (so that the space can be accesed)
- To dynamically allocate memory in C++, we use the **new** operator.
- De-allocation:
    1. Deallocation is the "clean-up" of space being used for variables or other data storage
    2. Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
    3. It is the programmer's job to deallocate dynamically created space
    4. To de-allocate dynamic memory, we use the **delete** operator

Allocating space with new

- To allocate space dynamically, use the unary operator **new**, followed by the *type* being allocated.
- new int; // dynamically allocates an int
- new double; // dynamically allocates a double
- If creating an array dynamically, use the same form, but put brackets with a size after the type:
- new int[40]; // dynamically allocates an array of 40 ints

- new double[size]; // dynamically allocates an array of size doubles
- // note that the size can be a variable
- These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the new operator returns the starting address of the allocated space, and this address can be stored in a pointer:
- int * p; // declare a pointer p
- p = new int; // dynamically allocate an int and load address into p
-
- double * d; // declare a pointer d
- d = new double; // dynamically allocate a double and load address into d
-
- // we can also do these in single line statements
- int x = 40;
- int * list = new int[x];
- float * numbers = new float[x+10];

Notice that this is one more way of *initializing* a pointer to a valid target (and the most important one).

Accessing dynamically created space

- So once the space has been dynamically allocated, how do we use it?
- For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:
- int * p = new int; // dynamic integer, pointed to by p
-
- *p = 10; // assigns 10 to the dynamic integer
- cout << *p; // prints 10
- For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:
- double * numList = new double[size]; // dynamic array
-
- for (int i = 0; i < size; i++)
- numList[i] = 0; // initialize array elements to 0
-
- numList[5] = 20; // bracket notation
- *(numList + 7) = 15; // pointer-offset notation
- // means same as numList[7]

Deallocation of dynamic memory

- To deallocate memory that was created with new, we use the unary operator **delete**. The one operand should be a pointer that stores the address of the space to be deallocated:
- int * ptr = new int; // dynamically created int
- // ...
- delete ptr; // deletes the space that ptr points to

Note that the pointer ptr *still exists* in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
ptr = new int[10];                    // point p to a brand new array
```

- To deallocate a dynamic array, use this form:
- delete [] *name_of_pointer*;

Example:

```
int * list = new int[40];  // dynamic array

delete [] list;              // deallocates the array
list = 0;                              // reset list to null pointer
```

After deallocating space, it's always a good idea to reset the pointer to null unless you are pointing it at another valid target right away.

- **To consider:** So what happens if you fail to deallocate dynamic memory when you are finished with it? (i.e. why is deallocation important?)

Application Example: Dynamically resizing an array

If you have an existing array, and you want to make it bigger (add array cells to it), you cannot simply append new cells to the old ones. Remember that arrays are stored in consecutive memory, and you never know whether or not the memory immediately after the array is already allocated for something else. For that reason, the process takes a few more steps. Here is an example using an integer array. Let's say this is the original array:

```
int * list = new int[size];
```

I want to resize this so that the array called **list** has space for 5 more numbers (presumably because the old one is full). There are four main steps.

1. Create an entirely new array of the appropriate type and of the new size. (You'll need another pointer for this).

```
int * temp = new int[size + 5];
```

3. Copy the data from the old array into the new array (keeping them in the same positions). This is easy with a for-loop.

```
for (int i = 0; i < size; i++)
   temp[i] = list[i];
```

6. Delete the old array -- you don't need it anymore! (Do as your Mom says, and take out the garbage!)

```
delete [] list;  // this deletes the array pointed to by "list"
```

7. Change the pointer. You still want the array to be called "list" (its original name), so change the list pointer to the new address.

```
list = temp;
```

The list array is now 5 larger than the previous one, and it has the same data in it that the original one had. But, now it has room for 5 more items.

Program:

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

Output:

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment: 8

Objective:Programs using Template and Exceptions

Theory:

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how do they work:

Function Template:

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
  // body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,**and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.

- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The**catch** keyword indicates the catching of an exception.

- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  // protected code
}catch( ExceptionName e1 )
{
  // catch block
}catch( ExceptionName e2 )
{
  // catch block
}catch( ExceptionName eN )
{
  // catch block
}
```

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using **throw**statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
  if( b == 0 )
  {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions:

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
  // protected code
}catch( ExceptionName e )
{
 // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try
{
  // protected code
}catch(...)
{
 // code to handle any exception
```

Program:

Function template that returns the maximum of two values:

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)
{
   return a < b ? b:a;
}
```

```
int main ()
{

   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

  return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

The following is the code which throws a division by zero exception and we catch it in catch block.

```
#include <iostream>
using namespace std;

double division(int a, int b)
{
  if( b == 0 )
  {
     throw "Division by zero condition!";
  }
```

```
   return (a/b);
}

int main ()
{
   int x = 50;
   int y = 0;
   double z = 0;

   try {
     z = division(x, y);
     cout << z << endl;
   }catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

```
Division by zero condition!
```

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment: 9

Objective:Programs using Sequential and Random access files

Theory:

The file stream classes support a number of member functions for performing the input and output operations on files. The functions get() and put() are capable of handling a single character at a time. The function getline() lets you handle multiple characters at a time. Another pair of functions i.e., read() and write() are capable of reading and writing blocks of binary data.

The get(), getline() and put() Functions

The functions get() and put() are byte-oriented. That is, get() will read a byte of data and put() will write a byte of data. The get() has many forms, but the most commonly used version is shown here, along with put() :

```
istream & get(char & ch) ;
ostream & put(char ch) ;
```

The get() function reads a single character from the associated stream and puts that value in ch. It returns a reference to the stream. The put() writes the value of ch to the stream and returns a reference to the stream.

Program:

The following program displays the contents of a file on the screen. It uses the get() function :

```
/* C++ Sequential Input/Output Operations on Files */

#include<iostream.h>
#include<stdlib.h>
#include<fstream.h>
#include<conio.h>
void main()
{
        char fname[20], ch;
        ifstream fin;      // create an input stream
        clrscr();
```

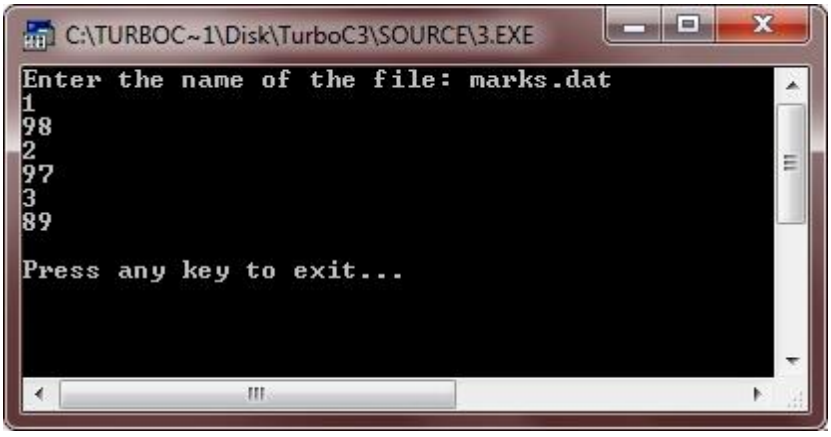
```
        cout<<"Enter the name of the file: ";
        cin.get(fname, 20);
        cin.get(ch);

        fin.open(fname, ios::in);       // open file
        if(!fin)          // if fin stores zero i.e., false value
        {
                cout<<"Error occurred in opening the file..!!\n";
                cout<<"Press any key to exit...\n";
                getch();
                exit(1);
        }

        while(fin)              // fin will be 0 when eof is reached
        {
                fin.get(ch);        // read a character
                cout<<ch;       // display the character
        }

        cout<<"\nPress any key to exit...\n";
        fin.close();
        getch();
}
```

Here is the sample output of the above C++ program. Let's suppose that the file contain the following information:



As stated, when the end-of-file is reached, the stream associated with the file becomes zero. Therefore, when fin reaches the end of the file, it will be zero causing the while loop to stop.

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Content beyond Syllabus

Experiment -1

CONSTRUCTOR AND DESTRUCTOR

AIM:

A program to print student details using constructor and destructor

Constructors

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors iitialize values to object members after storage is allocated to the object.

```
class A
{
 int x;
 public:
 A();  //Constructor
};
```

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
{
 int i;
 public:
 A(); //Constructor declared
};

A::A()   // Constructor definition
{
```

```
 i=1;
}
```

Types of Constructors

Constructors are of three types :

1. Default Constructor
2. Parametrized Constructor
3. Copy COnstructor

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax :

```
class_name ()
{ Constructor Definition }
```

Example :

```
class Cube
{
int side;
public:
Cube()
 {
  side=10;
 }
};

int main()
{
```

```
Cube c;
cout << c.side;
}
```

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
 int side;
};

int main()
{
 Cube c;
 cout << c.side;
}
```

Output : 0

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

Parameterized Constructor

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

Example :

```
class Cube
```

```
{
 int side;
 public:
 Cube(int x)
  {
   side=x;
  }
};

int main()
{
 Cube c1(10);
 Cube c2(20);
 Cube c3(30);
 cout << c1.side;
 cout << c2.side;
 cout << c3.side;
}
```

OUTPUT : 10 20 30

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

Copy Constructor

These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object. We will study copy constructors in detail later.

ALGORITHAM:

1. Start the process
2. Invoke the classes
3. Call the read() function
a. Get the inputs name ,roll number and address
4. Call the display() function
a. Display the name,roll number,and address of the student
5. Stop the process

```
#include<iostream.h>
#include<conio.h>
class stu
{
                private: char name[20],add[20];
                         int roll,zip;
                public: stu ( );//Constructor
                         ~stu( );//Destructor
                         void read( );
                         void disp( );
};
stu :: stu( )
{
        cout<<"This is Student Details"<<endl;
}
void stu :: read( )
{
        cout<<"Enter the student Name";
        cin>>name;
        cout<<"Enter the student roll no ";
        cin>>roll;
        cout<<"Enter the student address";
        cin>>add;
        cout<<"Enter the Zipcode";
        cin>>zip;
}
void stu :: disp( )
{
        cout<<"Student Name :"<<name<<endl;
        cout<<"Roll no   is      :"<<roll<<endl;
        cout<<"Address is      :"<<add<<endl;
```

```
        cout<<"Zipcode is       :"<<zip;
}
stu : : ~stu( )
{
        cout<<"Student Detail is Closed";
}

void main( )
{
        stu s;
        clrscr( );
s.read ( );
s.disp ( );
getch( );
}
```

Output:

```
Enter the student Name
James
Enter the student roll no
01
Enter the student address
Newyork
Enter the Zipcode
919108

Student Name : James
Roll no is : 01
Address is : Newyork
Zipcode is :919108
```

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment 2

Aim: A program that demonstrate the functionality of Friend function.

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

One of the important concept of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data. But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member function which is friend function and friend class.

friend Function in C++

If a function is defined as a friend function then, the private and protected data of class can be accessed from that function. The complier knows a given function is a friend function by its keyword friend. The declaration of friend function should be made inside the body of class (can be anywhere inside class either in private or public section) starting with keyword friend.

```
#include <iostream>
using namespace std;
class Distance
{
   private:
      int meter;
   public:
```

```
    Distance(): meter(0){ }
    friend int func(Distance);  //friend function
};
int func(Distance d)          //function definition
{
  d.meter=5;        //accessing private data from non-member function
  return d.meter;
}
int main()
{
  Distance D;
  cout<<"Distace: "<<func(D);
  return 0;
}
```

Output

```
Distance: 5
```

Here, friend function **func()** is declared inside Distance class. So, the private data can be accessed from this function.

Though this example gives you what idea about the concept of friend function, this program doesn't give you idea about when friend function is helpful.

Suppose, you need to operate on objects of two different class then,friend function can be very helpful. You can operate on two objects of different class without using friend function but, you program will be long, complex and hard to understand.

Example to operate on Objects of two Different class using friend Function

```
#include <iostream>
```

```
using namespace std;
class B;     // forward declaration
class A {
   private:
     int data;
   public:
     A(): data(12){ }
     friend int func(A , B);   //friend function Declaration
};
class B {
   private:
      int data;
   public:
      B(): data(1){ }
      friend int func(A , B);  //friend function Declaration
};
int func(A d1,B d2)
/*Function func() is the friend function of both classes A and B. So, the private data of both class can be accessed from this function.*/
{
  return (d1.data+d2.data);
}
int main()
   {
      A a;
      B b;
      cout<<"Data: "<<func(a,b);
      return 0;
   }
```

In this program, classes A and B has declared **func()** as a friend function. Thus, this function can access private data of both class. In this program, two objects of two different class A and B are passed as an argument to friend function. Thus, this function can access private and protected data of both class. Here, **func()** function adds private data of two objects and returns it to main function.

To work this program properly, a forward declaration of a class should be made as in above example(forward declaration of class B is made). It is because class B is referenced from class A using code: friend int func(A , B);. So, class A should be declared before class B to work properly.

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2

Experiment 3

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create virtual function, precede the function's declaration in the base class with the keyword virtual. When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.

- Base class pointer can point to derived class object. In this case, using base class pointer if we call some function which is in both classes, then base class function is invoked. But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual functions support runtime polymorphism.
- Consider the following program code : Class A

```
{
  int a;
  public:
     A()
     {
       a = 1;
     }
     virtual void show()
     {
       cout <<a;
     }
};

Class B: public A
{
  int b;
  public:
     B()
     {
       b = 2;
     }
     virtual void show()
     {
       cout <<b;
     }
};
```

```
int main()
{
  A *pA;
  B oB;
  pA = &oB;
  pA→show();
  return 0;
}
```

- Output is 2 since pA points to object of B and show() is virtual in base class A.

turn calls *_vptr which is automatically set when an instance of the class is created and it points to the virtual table for that class.

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
Co-relation	1	2	3	2	3	2	-	-	-	-	1	

PSO	PSO1	PSO2	PSO3	PSO4
Co-relation	2	3	2	2