Q.2 a. Distinguish between Procedure-oriented programming and Object-Oriented Programming.

Answer:

Procedure-oriented Programming basically consists of writing a list of instructions for the computer to follow and then organizing them into groups called function. Its characteristics are:

- Emphasis is on doing things (procedure)
- Larger programs are divided into smaller programs known as functions.
- Most Functions have their own local data and also share global data.
- Data moves freely among functions thus susceptible to inadvertent change
- Functions transform data from one form to another
- Applies a top-down approach in program design
- It does not model real world problems

Object-oriented Programming treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it and protects it from accidental modification from outside functions. Some of the striking features of Object-oriented programming:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure
- Data is hidden and cannot be accessed by external functions
- Objects may communicate with each other thru functions.
- New data and functions can be easily added wherever necessary
- Follows bottom-up approach in program design

b. Write a program in C++ which calculates the factorial of a given number.

Answer: Program to calculate factorial of a given number

```
#include <iostream.h>
long factorial (long a)
{
if (a > 1)
return (a * factorial (a-1));
else
return (1);
}
int main ()
{
long ln;
cout << "Type a number: ";
cin >> ln;
```

```
cout << "!" << ln << " = " << factorial (ln);
return 0;
}
```

Q.3 a. What is the main advantage of passing arguments by reference? Explain this with an example.

Answer:
There may arise situations where we would like to change the values of variables in the calling program. Passing arguments by reference is useful in object-oriented programming because it permits the manipulation of objects by reference and eliminates the copying of object parameters back and forth. References can be created not only for built-in data types but also for user-defined data types such as structures and classes. This means that when the function is working with its own arguments, it is actually working on the original data. E.g. In sorting algorithms we compare two adjacent elements in the list and interchange their values if the first element is greater than the second.

b. What does 'this' pointer point to? Explain.

Answer:
C++ contains a special pointer that is called 'this'. 'this' is a pointer that is automatically passed to any member function when it is called and it is a pointer to the object that generates the call.

```
#include<iostream.h>
#include<string.h>
class inventory
{
        class item[20];
        double cost;
        int on_hand;
  public:
        inventory(char *i,double c,int o)
        {
                strcpy(item->item,i);
                this->cost=c;
                this->on_hand=o;
        }
        void show()
};
void inventory::show()
{
        cout<<this->item;
        cout<<this->cost;
        cout<<this->on_hand;
}
```

```
int main()
{
        inventory ob("Wrench",495,4);
        ob.show();
        return 0;
}
```

c. Write a program in C++ to sort an array of positive integers.

Answer:

```
void main()
{ int *num,i,j,temp,n,flag;
num=new int[10];
cout<<"how many number to sort(limit)"<<endl;
cin>>n;
cout<<"enter numbers to sort in nondecreasing order"<<endl;
for(i=0;i<n;i++)
{
cin>>num[i];
}
for(i=0;i<n-1;i++)
{
flag=1;
for(j=0;j<(n-1-i);j++)
{
if(num[j]>num[j+1])
{
flag=0;
temp=num[j];
num[j]=num[j+1];
num[j+1]=temp;
}
}
if(flag)
break;
}
for(i=0;i<n;i++)
{
cout<<num[i]<<endl;
}
}
```

Q.4 a. How do the properties of following two derived classes differ?
(i) class X : public A{//..}
(ii) class Y : private A{//..}

Answer:
The properties of the following two derived class differ:-

(i) class X:public A{//..}
In this class A is publicly derived from class X. the keyword public specifies that objects of derived class are able to access public member function of the base class.

(ii) class Y:private A{//..}
In this class A is privately derived from class Y. the keyword private specifies that objects of derived class cannot access public member function of the base class. Since object can never access private members of a class.

b. Explain the characteristics of static class members.

Answer:
Static class member variables are used commonly by the entire class. It stores values. No different copies of a static variable are made for each object. It is shared by all the objects. It is just like the C static variables.
It has the following characteristics:

- On the creation of the first object of the class a static variable is always initialized by zero.
- All the objects share the single copy of a static variable.
- The scope is only within the class but its lifetime is through-out the program.

c. Is it possible that a function is friend of two different classes? If yes, then how it is implemented in C++?

Answer:
Yes, it is possible that a function can be a friend of two classes. Member functions of one class can be friend with another class. This is done by defining the function using the scope resolution operator.

We can also declare all the member functions of one class as the friend functions of another class. In fact we can also declare all the member functions of a class as friend. Such a class is then known as friend class.

The following program shows how a friend function can be a friend of two classes:

```
#include<iostream.h>
using namespace std;
class one;
class two
{
int a;
public:
```

```
void setvalue(int n){a=n;}
friend void max(two,one);
};
class one
{
int b;
public:
void setvalue(int n){b=n;}
friend void max(two,one);
};
void max(two s,one t)
{
if(s.a>=t.b)
cout<<s.a;
else
cout<<t.b;
}
int main()
{
one obj1;
obj1.setvalue(5);
two obj2;
obj2.setvalue(10);
max(obj2,obj1);
        return 0;}
```

Q.5 a. Why do we need constructors?

Answer:

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked automatically whenever an object of its associated class is created. It is called constructor because it construct the values of data members of the class.

The constructors enable us to initialize the objects when they are created. They can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of the memory. Class objects can be initialized dynamically too. The advantage is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

b. Differentiate between Default constructor and copy constructor using suitable example.

Answer:
Default constructor: A constructor that accepts no argument is called default constructor. This default constructor takes no parameters, or has default values for all parameters. The default constructor for the class A is A::A(). If no such constructor is defined, then the compiler supplies a default constructor. A default parameter is one where the parameter value is supplied in the definition. For example in the class A defined below 5 is the default value parameter.

```
Class A
{ int value;
Public:
A(int param=5)
{
value = param;
}
};
```

A copy constructor is a special constructor in the C++ programming language used to create anew object as a copy of an existing object. This constructor takes a single argument: a reference
to the object to be copied. Normally the compiler automatically creates a copy constructor for each class (known as an implicit copy constructor) but for special cases the programmer creates the copy constructor, known as an explicit copy constructor. In such cases, the compiler doesn't create one.

For example the following will be valid copy constructors for class A.

```
A(A const&);
A(A&);
A(A const volatile&);
    A(A volatile&);
```

c. Explain the concept of operator overloading. Write a program to overload the operator '+' for complex numbers.

Answer:
To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied.

e.g. The mathematical operator "+" can add two numbers eg 4+5=9

To use it for strings "foot"+"ball'="football" will require describing (coding)) the task in the form of function and attaching it to the operator. The given program applies mathematical '+' operator to concatenate (add) two strings.

To perform complex number addition using operator overloading.

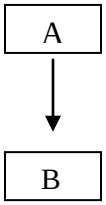
```
# include <iostream.h>
# include <conio.h>
class complex {
float r, i;
public:
complex()
{
r=i=0;
}
void getdata()
{
cout<<"R.P";
cin>>r;
cout<<"I.P";
cin>>i;
}
void outdata (char*msg)
{
cout<<endl<,msg;
cout<<"("<<r;
cout<<"+j" <<i<<")";
}
Complex operator+(Complex);
};
Complex complex::Operator+(Complex(2))
{
complex temp;
temp.r=r+c2.r;
temp.i=I=c2.i;
return(temp);
}
void main()
{
clrscr();
complex c1, c2, c3;
cout<<"Enter 2 complex no: "<<endl;
cl.getdta();
c2.getdata();
c3=c1+c2;
c3.outdata ("The result is :");
getch();
        }
```

Q.6 a. What does inheritance mean in C++? What are the different forms of Inheritance?

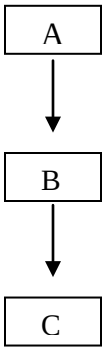
Answer:
C++ strongly supports the concept of reusability. The C++ classers can be reused in several ways. Once a class has been written and tested, it can be adapted by other programs to suit their requirements. This can be done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from the old one is called inheritance (or derivation). The old class is base class and new class is called derived class.

The derived class inherits some or all traits from the base class. A class can also inherit properties from more than one class.

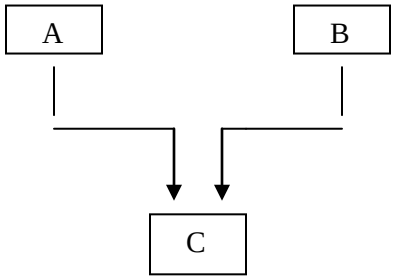
(i) A derived class with only one base class is called single inheritance.



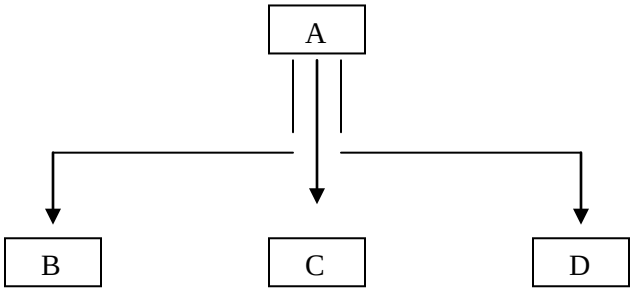
(ii) The mechanism of deriving a class from another ‘derived class’ is known s multilevel inheritance.



(iii) A derived class with several base classes is called multiple inheritance



(iv) The traits of one class may be inherited by more than one class. This process is called hierarchical inheritance.



b. What is multiple inheritance? Write a program that explains how to pass parameters to the constructors of base classes in multiple inheritance.

Answer:

C++ provides us with a very advantageous feature of multiple inheritance in which a derived class can inherit the properties of more than one base class. The syntax for a derived class having multiple base classes is as follows:

```
Class D: public visibility base1, public visibility base2
{
Body of D;
}
```

Visibility may be either 'public' or 'private'.

In case of multiple inheritance, the base classes are constructed in the order in which theyappear in the declaration of the derived class. However the constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. An example program illustrates the statement.

```
#include<iostream.h>
using namespace std;
class M
{
protected:
int m;
public:
M(int x)
{
m=x;
cout<< "M initialized\n";
}
};
class N
{
protected:
int n;
```

```
public:
N(int y)
{
n=y;
cout<< "N initialized\n";
}
};
class P: public N, public M
{ int p,r;
public:
P(int a,int b,int c, int d):M(a),N(b)
{
p=c;
r=d;
cout<< "P initialized\n";
}
};
void main()
{
P p(10,20,30,40);
}
```

Output of the program will be:
N initialized
M initialized
P initialized

Q.7 a. Differentiate between late and early binding.

Answer:
The concept of polymorphism is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at compile time and therefore compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding or compile time polymorphism.

This means that an object is bound to its function call at compile time.
If the member function could be selected while the program is running. This is known as run time polymorphism. At run time when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding. It is also called dynamic binding because the selection of the appropriate function is done dynamically at run time.

b. How is polymorphism achieved at run time? Explain with appropriate example.

Answer:

Polymorphism means one name having multiple forms. It is implemented using the overloaded functions and operators. The overloaded functions could be 'selected" for invoking by matching arguments both type and number. This can be done at both compile time and run time. If appropriate member function can be selected while the program is running, this is known as run-time polymorphism. This can be achieved using virtual functions. At run time when it is known what class objects are under consideration, the appropriate version of the function is invoked.

```
//Program to demonstrate Run-time polymorphism

        #include <iostream.h>
        class Base
        {
                public:
                void display() { cout<<"\n display Base";}
                virtual void show() {cout<<"\nshow Base";}
        };
        Class Derived: public Base
        {
                public:
                void display(){cout<<"\ndisplay Derived";}
                void show(){cout<<"\n show Derived";}
        };
        int main()
        {
                Base B;
                Derived D;
Base *bptr;

cout<<"\n bptr points to Base \n";
bptr=&B;
```

```
    bptr->display();   //calls Base version
    bptr->show();  //calls Base version

    cout<<"\n bptr points to Derived\n";
    bptr= &D;
    bptr->display();
    bptr->show();

    return 0;

}//main over
```

c. How is exception handling implemented in C++?

Answer:
Exception Handling: Using exception handling we can more easily manage and respond to run time errors. C++ exception handling is built upon 3 keywords: try, catch and throw.

- The 'try' block contains program statements that we want to monitor for exceptions.
- The 'throw' block throws an exception to the 'catch' block if it occurs within the try block.
- The 'catch' block proceeds on the exception thrown by the 'throw' block.

When an exception is thrown, it is caught by its corresponding catch statement which processes the exception. There can be more that one catch statement associated with a try. The catch statement that is used is determined by the type of the exception.

Q.8 a. Differentiate between function overloading and function templates. Explain using examples for both.

Answer:
Function names can be overloaded in C++. We can assign the same name to two or more distinct functions. Example:

```
int mul(int a, int b);  //prototype
float mul(float a, float b); //prototype
```

The function name **mul()** is overloaded, meaning it has more than one implementation. The execution of the correct implementation is decided by the compiler by matching the type of the arguments in the function call with the types in the function prototypes.

We define function templates to create a family of functions with different argument types. E.g. given is **swap()** function template that will swap two values of a given type of data.

```
Template<class T>
Void swap(T&x, T&y)
{
        T temp=x;
        x=y;
        y=temp;
}
```

We can invoke the **swap()** function like any ordinary function. E.g. we can apply the **swap()** function as follows:

```
void f(int m, int n, float a, float b)
{
        swap(m,n); //swap two integer values
        swap(a,b); //swap two float values
}
```

This will generate a swap( ) function from the function template for each set of argument types. Thus we do not have to define separate swap( ) functions for different argument types.

b. Write a function template to find a maximum value from an array.

Answer:

```
#include<iostream.h>
using namespace std;
template<class T>
void max(T a[],T &m, int n)
{
        for(int i=0;i<n;i++)
            if(a[i]>m)
```

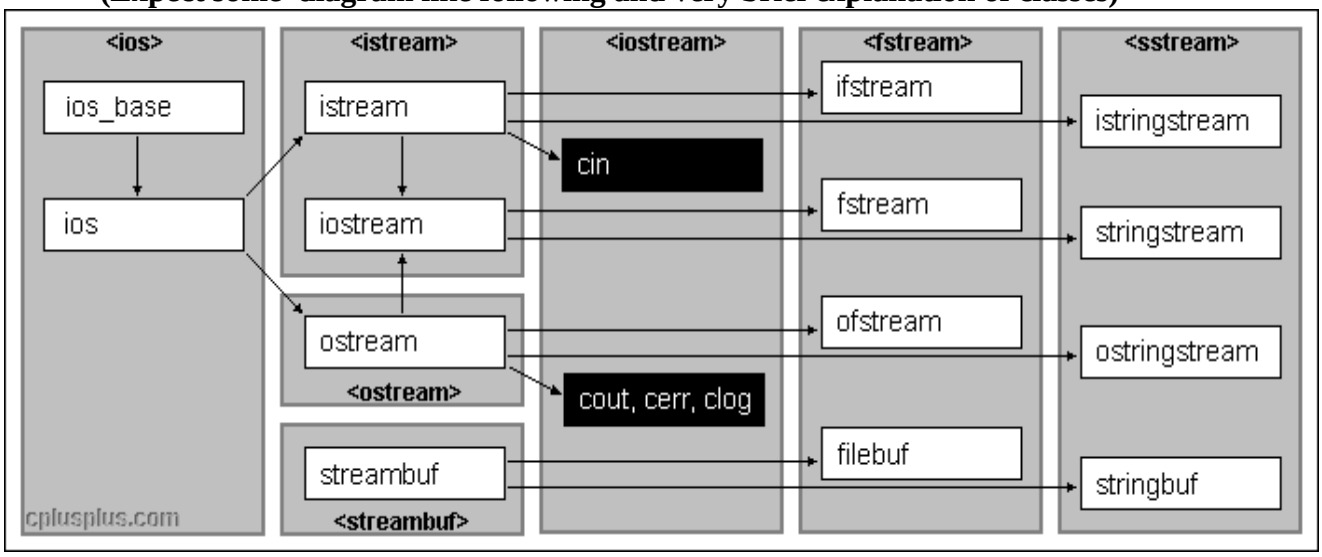
```
            m=a[i];
    }
    int main()
    {
        int x[5]={10,50,30,40,20};
        int m=x[0];
        max(x,m,5);
        cout<<"\n The maximum value is : "<<m;
        return 0;
    }
```

Q.9 a. Explain the I/O stream hierarchy in C++.

Answer:

The following hierarchies of classes are used to define various streams which deal with both the console and disc files. These classes are declared in the header file iostream. ios is the base for istream, ostream and iostream base classes. ios is declared as a virtual base class so that only one copy of its members are inherited by the iostream.

(Expect some diagram like following and very brief explanation of classes)



b. Explain the following functions(with example) for manipulating file pointers:
seekg(), seekp(), tellg(), tellp()

Answer:
A file pointer is used to navigate through a file. We can control this movement of the file pointer by ourselves. The file stream class supports the following functions to manage such situations.

- **seekg ():** moves get pointer (input) to a specified location.
- **seekp ():** moves put pointer (output) to a specified location.
- **tellg ():** gives the current position of the get pointer.
- **tellp ():** gives the current position of the put pointer.

For example:
infile.seekg (10); will move the file pointer to the byte number 10.
ofstream fileout;
fileout.open("morning", ios::app);
int p = fileout.tellp();
the above statements will move the output pointer to the end of the file morning and the value of p will represent the number of bytes in the file.
seekg and seekp can also be used with two arguments as follows.
seekg (offset, ref position);
seekp (offset, ref position);
where offset referes to the number of bytes the file pointer is to be moved from the location specified by the parameter ref position.

TEXT BOOK

I. Object-oriented Programming with C++, Poornachandra Sarang, PHI, 2004