Q.2 a. Describe the basic characteristics of Object-Oriented Programming.

Answer:
The basic characteristics of Object-oriented Programming are:-

(i) Encapsulation – (Including in an object everything it needs, hiding elements that other objects needn't know about). This keeps data and related routines together and un-clutters the large scale organization of the program. Each object has a 'public' set of routines that can be called, and these routines are all that other objects need to know.

(ii) Inheritance (creating new types of objects from existing ones). Rather than having many seemingly unrelated objects, objects can be organized hierarchically, inheriting behavior. This simplifies the large-scale organization.

(iii) Polymorphism (different objects responding to the same message in different ways). Rather than having a different routine to do the same thing to each of many different types of objects, a single routine does the job. An example of this is how the + operator can be overloaded in C++ so that it can be used with new classes.

b. 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.

c. What is the output of the following program? In view of these outputs, explain the meaning of pre/post increment & pre/post decrement operators.

```
 #include <iostream.h>
 #include<conio.h>
 void main()
{
int a=5,b=0;
clrscr();
b=b+(++a);
cout<< "\n\nb =" <<b;
b=b+(a++);
cout<< "\n\nb =" <<b;
b=b+(--a);
cout<< “\n\nb =” <<b;

b=b+(a--);
cout<<“\n\nb=”<<b;
getch();
}
```

Answer:

```
#include<iostream.h>
#include<conio.h>
void main()
{
int a=5,b=0;
clrscr();
b=b+(++a);           //pre-increament  b=0+(6)    a=6,b=6
cout << "\n\nb = "<<b;

b=b+(a++);          //post-increament  b=6+(6)   a=7,b=12
cout << "\n\nb = "<<b;

b=b+(--a);            //pre -decreament  b=12+(6)  a=6,b=18
cout << "\n\nb = "<<b;

b=b+(a--);            //post -decreament b=18+(6)  a=7,b=24
cout << "\n\nb = "<<b;
getch();
}
6
12
18
24
```

Q.3 a. Write a program that read the student name and marks in three subjects and display the total marks and percentage obtained by the student. The program should declare the student as a structure and read and write the elements accordingly.

Answer:

```
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
struct student
{
   char name[20];
   int mark1,mark2,mark3;
};

void main()
{
  clrscr();
  float total,percent;
  student temp;
  cout<<"enter the student details :\n";
  cout<<"Name : ";
  cin>>temp.name;
  cout<<"Marks (out of 100) in three subjects\n";
  cin>>temp.mark1>>temp.mark2>>temp.mark3;
  total=(temp.mark1+temp.mark2+temp.mark3);
  cout<<"\nTotal marks =  "<<total;
  cout<<"\nTotal Percentage = "<<(total/3);
}
```

b. Write a C++ program to generate the following series:
1 2 3 5 8 13 21 34.......

Answer:

```
#include<iostream.h>
#include<conio.h>
void main()
{
int size,a=0,b=1,c,i;
cout<< "\n\nEnter the size of the series : -";
cin >> size;
cout << "\n\n\t\t\t";

for(i=0;i<size;i++)
{
c=a+b;
cout <<"  "<< c;
a=b;
b=c;
}
getch();
}
```

Q.4 a. Explain inline function in C++ using a suitable example. State various situations where inline expansion may not work.

Answer:
Whenever we write functions, there are certain costs associated with it such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function. To remove this overhead we write an inline function. It is a sort of request to the compiler. If we declare and define a function within the class definition then it is automatically inline. We do not need to sue the term inline.

```
function definition min()
inline void min (int x, int y)
cout<< (x < Y? x : y);
}
Void main()
{
int num1, num2;
cout<<"\Enter the two intergers\n";
cin>>num1>>num2;
min (num1,num2; //function code inserted here
------------------
------------------
}
```

There are certain conditions in which an inline function may not work:

- If a function is returning some value and it contains a loop, a switch or a goto statement.
- If a function is not returning a value and it contains a return statement.
- If a function contains a static variable.
- If the inline function is made recursive.

b. Explain meaning of a reference and call by reference method. Write a C++ program that swap two numbers using call by reference.

Answer:
A reference provides an alias – an alternate name – for the variable, i.e., the same variable's value can be used by two different names : the original name and the alias name.
In call by reference method, a reference to the actual arguments(s) in the calling program is passed (only variables). So the called function does not create its own copy of original value(s) but works with the original value(s) with different name. Any change in the original data in the called function gets reflected back to the calling function.

It is useful when you want to change the original variables in the calling function by the called function.

```
//Swapping of two numbers using function call by reference
#include<iostream.h>
#include<conio.h>
void main()
```

```
{
clrscr();
int num1,num2;
void swap (int &, int &); //function prototype
cin>>num1>>num2;
cout<<"\nBefore swapping:\nNum1: "<<num1;
cout<<endl<<"num2: "<<num2;
swap(num1,num2); //function call
cout<<"\n\nAfter swapping : \Num1: "<<num1;
cout<<endl<<"num2: "<<num2;
getch();
}
//function fefinition swap()
void swap (int & a, int & b)
{
Int temp=a;
a=b;
b=temp;
}
```

Q.5 a. Write a C++ program that implement the following specifications:
A class that represent bank account number, type of accounts (s for savings and c for current), balance amount.

The class contains member functions to do the following:
(i) To initialize the data member
(ii) To deposit money
(iii) To withdraw money after checking balance (minimum balance is Rs. 1000/-)
(iv) To display the data members

Answer:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
class bank
{
        char name[25],type;
        long ac_no;
        float bal, amt;
        public:
        class bank *next;
        bank()
        {
```

```
        strcpy(name," ");
        type=' ';
        ac_no=0;
        bal=0;
        next=NULL;
}
void deposit()
{
        cout<<endl<<"Amount to be deposited:-> ";
        cin>>amt;
        bal+=amt;
        cout<<endl<<"Amount deposited successfully";

}
void withdraw()
{
        cout<<endl<<"How much u want to withdraw:-> ";
        cin>>amt;
        if(bal-amt >= 1000)
        {
                bal=amt;
                cout<<endl<<"Withdraw Done";
        }
        else
        {
                cout<<endl<<"amount exceeds the available balance ";
                getch();
        }
}
void showdata()
{
        cout<<endl<<name<<'\t'<<ac_no<<'\t';
        if (type == 'S') cout<<"Savings";
        else
        if (type=='C') cout<<"Current";
        cout<<'\t'<<bal;

}
int search(long x)
{
        if (ac_no==x)
           return 1;
        else
           return 0;

}
```

```
        void getdata();
};
void bank::getdata()
{
        cout<<endl<<"Enter name of Account Holder:-> ";
        cin>>name;
        cout<<endl<<"Enter Account Number:-> ";
        cin>>ac_no;
  rep:
        cout<<endl<<"Enter type of Account S=Saving, C=Current):-> ";
        cin>>type;
        if (type!='S' && type !='C')
        {
                cout<<endl<<"Wrong Input";
                getch();
                goto rep;
        }
        cout<<endl<<"Enter Amount:-> ";
        cin>>bal;
        next=NULL;

}

void main()
{
        class bank *bptr=NULL, *tmp=NULL, *counter=NULL;
        int i=0,found=0,ch=0;

        long acno=0;

        while(1)
        {
                clrscr();
                cout<<endl<<"1. Add a new customer";
                cout<<endl<<"2. View Customer Details";
                cout<<endl<<"3. Deposit Amount";
                cout<<endl<<"4. Withdraw Amount";
                cout<<endl<<"5. Exit.";
                cout<<endl<<"Enter your choice:-> ";
                cin>>ch;
                switch(ch)
                {
                  case 1: if (bptr==NULL)
                          {
                            bptr=new bank;
                            if (bptr==NULL)
                            {
                            cout<<endl<<"Memory Allocation Problem";
                            exit(1);
                            }
                          }
                          else
                          {
                            tmp=new bank;
                            if (tmp==NULL)
                            {
                                 cout<<endl<<"Memory Allocation problem";
                                 exit(1);
                            }
                            tmp->getdata();
                            for        (counter=bptr;        counter->next!=NULL;
counter=counter->next);
                            counter->next=tmp;
                          }
                          break;
                  case 2:
                          cout<<endl<<"Name \t Ac. No. \t Type \tBalance \n";
                          for  (counter=bptr;  counter!=NULL;  counter=counter-
>next)
                         counter->showdata();
                         getch();
                         break;

                  case 3:
                          cout<<endl<<"Enter the account no. to deposit ";
                          long acno;
                          cin>>acno;
                          for        (counter=bptr;counter!=NULL;counter=counter-
>next)
                          {
                             found=counter->search(acno);
                             if(found==1)
                             {
                                  counter->deposit();
                                  getch();
                                  break;
                             }
                          }
                          if (found==0)
                          {
                            cout<<endl<<"Account does not exist ";
                            getch();
                          }
                          break;
                  case 4:

                          cout<<endl<<"Enter the account no. to Withdraw ";
                          cin>>acno;
                          for        (counter=bptr;counter!=NULL;counter=counter-
>next)
                          {
                             found=counter->search(acno);
                             if(found==1)
                             {
                                  counter->withdraw();
                                  break;
                             }
                          }
                          if (found==0)
                          {
                            cout<<endl<<"Account does not exist ";
                            getch();

                          }
                          break;

                  case 5:exit(1);

                  default: cout<<endl<<"Wrong Choice ";

                 }//switch
          }
          getch();
}
```

b. Why is a destructor function required in class? Can a destructor accept argument?

Answer:

Just as a constructor is used to initialize an object when it is created, a destructor is used to clean up the object just before it is destroyed. A destructor has the same name as the class itself, but is preceded with a ~ symbol. Unlike constructors a class may have at most one destructor. A destructor does not take any arguments and has no explicit return type.

e.g. the destructor for the class integer can be defined as shown below:

```
~ integer() { };
```

The destructor will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Destructors are generally useful for classes which have pointer data members which point to memory blocks allocated by the class itself. In such cases it is important to release member-allocated memory before the object is destroyed. A destructor can do just that. Whenever new is used to allocate memory in the constructors, we should use delete to free that memory. This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

Q.6 a. What do you mean by operator overloading? What is the difference between implementation of overloading unary and binary operators using member and friend functions?

Answer:
When an operator is overloaded it doesn't lose its original meaning but it gains an additional meaning relative to the class for which it is defined. We can overload unary and binary operators using member and friend functions.
When a member operator function overloads a binary operator the function will have only one parameter. This parameter will receive the object that is on right of the operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by "this".
Overloading a unary operator is similar to overloading a binary operator except that there is only one operand to deal with. When you overload a unary operator using a member function the function has no parameter. Since there is only one operand, it is this operand that generates the call to the operator function.
Overloading operator using a friend function: as we know that a friend function does not have a 'this' pointer. Thus in cases of overloading a binary operator two arguments are passed to a friend operator function. For unary operator, single operand is passed explicitly.

b. Illustrate with an example the overloading of ++ operator for incrementing.

Answer:

```
include<iostream>
using namespace std;

//Increment and decrement overloading
class Inc {
        private:
                int count ;
        public:
                Inc() {
                        //Default constructor
                        count = 0 ;
                }

                Inc(int C) {
                        // Constructor with Argument
                        count = C ;
                }

                Inc operator ++ () {
                        // Operator Function Definition
                        return Inc(++count);
                }



                void display(void) {
                        cout << count << endl ;
                }
};

void main(void) {
        Inc a, b(4), c, d, e(1), f(4);

        cout << "Before using the operator ++()\n";
        cout << "a = ";
        a.display();
        cout << "b = ";
        b.display();
        ++a;
        b++;
        cout << "After using the operator ++()\n";
        cout << "a = ";
        a.display();
        cout << "b = ";
        b.display();
        c = ++a;
        d = b++;
        cout << "Result prefix (on a) and postfix (on b)\n";
        cout << "c = ";
        c.display();
        cout << "d = ";
        d.display();
```

}

Q7 a. What is polymorphism? Differentiate between compile-time and run-time polymorphism using suitable examples.

Answer:
The information is known to the compiler at the 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 static binding.
The linking of function with a class much later after the compilation, this process is termed as late binding or dynamic binding because the selection of the appropriate function is done dynamically at run time. This requires the use of pointers to objects.
(write small C++ programs for illustration)

b. Is it possible for a derived class to inherit two or more base classes? Explain how it is implemented in C++ using an example program and its output.

Answer:
It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.
// An example of multiple base classes.

```
#include <iostream>
using namespace std;
class base1 {
protected:
int x;
public:
void showx() { cout << x << "\n"; }
};
class base2 {
protected:
int y;
public:
void showy() {cout << y << "\n";}
};
// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
void set(int i, int j) { x=i; y=j; }
};
int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
```

```
        ob.showy(); // from base2
        return 0;
        }
```

Q8. a. What is a class template? Write a template-based complete program for adding two objects of the vector class.

Answer:
The class template definition is very similar to an ordinary class definition except the prefix **template<class T>** and the use of type **T**. This prefix tells the complier that we are going to declare a template and use **T** as a type name in the Declaration. Thus, vector has become a parameterized class with the type **T** as its parameters. **T** may be substituted by any data type including the user defined types. Now we can create vectors for holding different data types.

Example;

```
        vector<int> v1(10); //10 element int vector
        vector<float> v2(30); //30 element float vector
```

The type **T** may represent a class name as well.

Example:

```
        Vector<complex> v3 (5); // vector of 5 complex numbers
```

A class created from a class template is called a template class. The syntax for defining an object of a template class is:
Classname<type> objectname (arglist);
This process of creating a specific class from a class template is called instantiation. The complier will perform the error analysis only when an instantiating take place. It is, therefore, advisable to create and debug an ordinary class before converting it in to template.

b. How is an exception handled in C++? Write a program that illustrates the application of multiple catch statements.

Answer:
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. An example illustrates the working of the three blocks.

It is possible that arguments of several catch statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

```
#include<iostream.h>
void test(int x)
{
        try
        {
                if (x == 1) throw x;                    //int
                else
                        if(x == 0) throw ‘x’;           //char
                else
                        if(x == -1) throw 1.0;          //double
                cout<<”End of try-block \n”;
        }
        catch(char c)   //Catch 1
        {
                cout<<” Caught a Character \n”;
        }
        catch(int c)    //Catch 2
        {
                cout<<” Caught an integer \n”;
        }
        catch(double d)         //Catch 3
        {
                cout<<” Caught a double   \n”;
        }
        cout<<” End of try-catch system \n\n”;

}
void main()
{
        cout<<” Testing Multiple catches \n”;
        cout<<” x == 1 \n”;
        test(1);
        cout<<” x == 0 \n”;
        test(0);
        cout<<” x == -1 \n”;
        test(-1);
        cout<<” x == 2 \n”;
        test(2);

}
```

Q9. a. Using a suitable C++ program, illustrate the meaning of following ios function: width(), fill() and precision().

Answer:
By default, when a value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the **width()** function. Its prototype is shown here:

streamsize width(streamsize *w*);

Here, *w* becomes the field width, and the previous field width is returned. In some implementations, the field width must be set before each output. If it isn't, the default field width is used. The **streamsize** type is defined as some form of integer by the compiler.
After you set a minimum field width, when a value uses less than the specified width, the field will be padded with the current fill character (space, by default) to reach the field width. If the size of the value exceeds the minimum field width, the field will be overrun. No values are truncated.
When outputting floating-point values, you can determine the number of digits to be displayed after the decimal point by using the **precision()** function. Its prototype is shown here:

streamsize precision(streamsize *p*);

Here, the precision is set to *p*, and the old value is returned. The default precision is 6.
In some implementations, the precision must be set before each floating-point output.
If it is not, then the default precision will be used.
By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the **fill()** function. Its prototype is

char fill(char *ch*);

After a call to **fill()** , *ch* becomes the new fill character, and the old one is returned.
Here is a program that illustrates these functions:

```
#include <iostream>
using namespace std;
int main()
{
cout.precision(4) ;
cout.width(10);
cout << 10.12345 << "\n"; // displays 10.12
cout.fill('*');
cout.width(10);
cout << 10.12345 << "\n"; // displays *****10.12
// field width applies to strings, too
cout.width(10);
cout << "Hi!" << "\n"; // displays *******Hi!
cout.width(10);
cout.setf(ios::left); // left justify
cout << 10.12345; // displays 10.12*****
return 0;
}
```

This program's output is shown here:

```
10.12
*****10.12
*******Hi!
10.12*****
```

b. Briefly explain the three foundational items of the standard template library(STL):
containers, algorithms and iterators

Answer:
Containers: *Containers* are objects that hold other objects, and there are several different types. For example, the **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called *sequence containers* because in STL terminology, a sequence is a linear list. In addition to the basic containers, the STL also defines *associative containers,* which allow efficient retrieval of values based on keys. For example, a **map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key.
Each container class defines a set of functions that may be applied to the container.
For example, a list container includes functions that insert, delete, and merge elements.
A stack includes functions that push and pop values.
Algorithms: *Algorithms* act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of containers. Many algorithms operate on a *range* of elements within a container.
Iterators: *Iterators* are objects that are, more or less, pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. There are five types of iterators:

Iterator	Access Allowed
Random Access	Store and retrieve values. Elements may be accessed randomly.
Bidirectional	Store and retrieve values. Forward and backward moving.
Forward	Store and retrieve values. Forward moving only.
Input	Retrieve, but not store values. Forward moving only.
Output	Store, but not retrieve values. Forward moving only.

In general, an iterator that has greater access capabilities can be used in place of one that has lesser capabilities. For example, a forward iterator can be used in place of an input iterator.
Iterators are handled just like pointers. You can increment and decrement them.
You can apply the ***** operator to them. Iterators are declared using the **iterator** type defined by the various containers.
The STL also supports *reverse iterators*. Reverse iterators are either bidirectional or random-access iterators that move through a sequence in the reverse direction. Thus, if a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point to one element before the end.

Text Book

C++ & Object-Oriented Programming Paradigm, Debasish Jana, 2nd Edition, PHI, 2005.