Q.2 a. Compare and contrast Procedure-oriented Programming with Object-oriented programming.

Answer:

Features of procedure-oriented programming:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

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 ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Q.3 a. Write a program in C++ for multiplication of two matrices.

Answer:

Program for Multiplication of two matrices:

```
#include<iostream.h>

#include<conio.h>

void main()

{

clrscr();

int a[10][10],b[10][10],c[10][10],m,n,o,p,i,j;

cout<<"Enter number of rows of A: ";

cin>>m;

cout<<"Enter number of columns of A: ";

cin>>n;

cout<<endl<<"Enter elements of matrix A: "<<endl;

for(i=0;i<m;i++)

{

for(j=0;j<n;j++)

{

cout<<"Enter element a"<<i+1<<j+1<<": ";

cin>>a[i][j];

}

}

cout<<endl<<"Enter number of rows of B: ";

cin>>o;
```

```
cout<<"Enter number of columns of B: ";
cin>>p;
cout<<endl<<"Enter elements of matrix B: "<<endl;
for(i=0;i<o;i++)
{
for(j=0;j<p;j++)
{
cout<<"Enter element b"<<i+1<<j+1<<": ";
cin>>b[i][j];
}
}

cout<<endl<<"Displaying Matrix A: "<<endl<<endl;
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
cout<<a[i][j]<<" ";
}
cout<<endl<<endl;
}
cout<<endl<<"Displaying Matrix B: "<<endl<<endl;
for(i=0;i<o;i++)
```

```
{
for(j=0;j<p;j++)
{
cout<<b[i][j]<<" ";
}
cout<<endl<<endl;
}
if(n==o)
{
for(i=0;i<m;i++)
{
for(j=0;j<p;j++)
{
c[i][j]=0;
for(int k=0;k<n;k++)
{
c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
}
}
cout<<endl<<"Matrix A * Matrix B = Matrix C: "<<endl<<endl;
for(i=0;i<m;i++)
{
```

```
for(j=0;j<p;j++)

{

cout<<c[i][j]<<" ";

}

cout<<endl<<endl;

}

}

else

cout<<"Multiplication not possible :(";

getch(); }
```

b. What are structures in C++? How does a structure differ from an array? Explain.

Answer: Page Number 78 of Text Book

Q.4 a. Explain the concept of a class in object-oriented paradigm. How does it accomplish data hiding?

Answer:

CLASS:

A class is a way to bind the data and its associated function together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any others built-in data type. Generally, a class specification has two parts

1. Class declaration
2. Class function definitions

- The class declaration describe the type and scope of its members.
- The class function definations describe how the class functions are implemented.

The general form of a class declaration is:

```
Class class_ name
{
    Private:
            Variable declarations;
            Function declarations;
    Public:
           Variable declarations;
            Function declarations;
}
```

The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies that what follows is an abstract data type class name. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables are collectively called class members. They usually grouped under two sections, namely, private and public to denote which of the members are private and which of them public. The keyword **private** and **public** are known as visibility labels. These keywords are followed by a colon.

The class the class members that have been declared as private can be accessed only from within the class on the other hand, public members can be accessed from outside the class also.

The data hiding (using private declaration) is the key feature of object oriented programming. The use of the keyword private is optional. By default, the members of a class are **private.** If both the labels are missing, then, by default, all the members are **private**. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as data members and the functions are known as member functions. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. This is illustrated in **fig.** The binding of data and functions together into a single class type variable is referred to as encapsulation.

DEFINITION AND DECLARATION OF A CLASS

A class in C++ combines related data and functions together. It makes a data type which

is used for creating objects of this type.

Classes represent real world entities that have both data type properties (characteristics)

and associated operations (behavior).

The syntax of a class definition is shown below :

Class name_of _class

{

private : variable declaration; // data member

Function declaration; // Member Function (Method)

protected: Variable declaration;

Function declaration;

public : variable declaration;

Function declaration;

};

Here, the keyword class specifies that we are using a new data type and is followed by the class name.

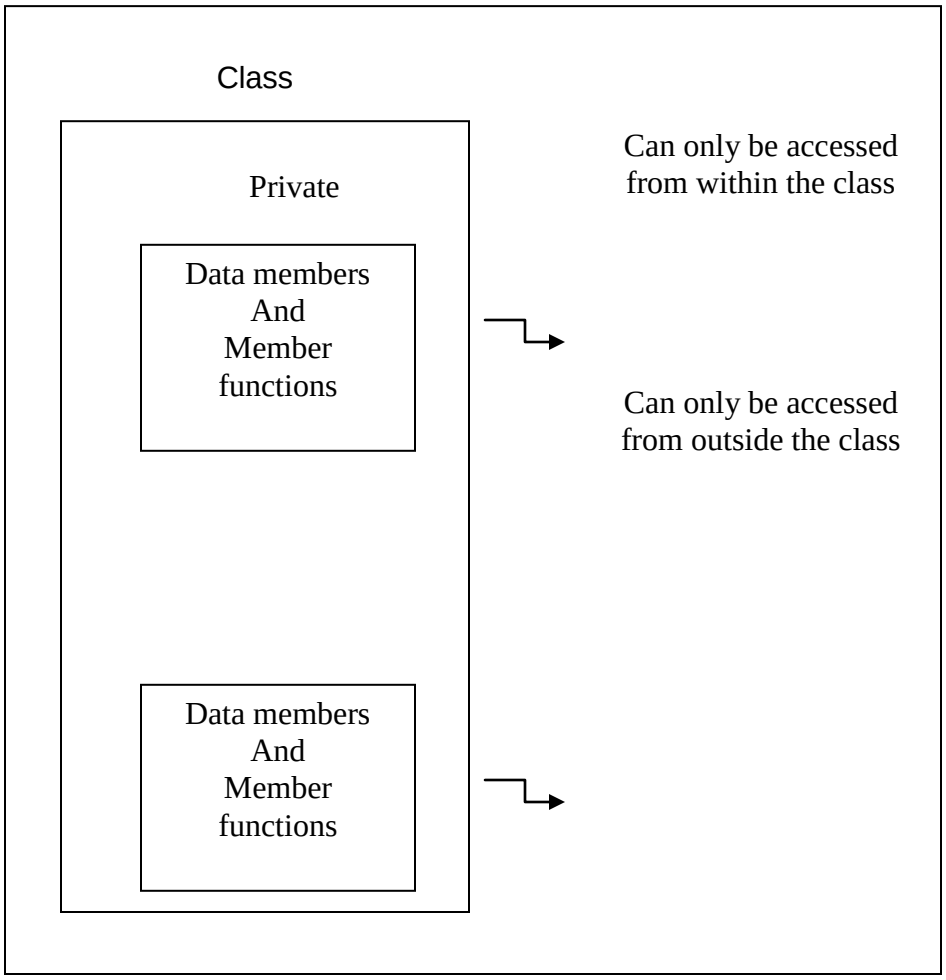
The body of the class has two keywords namely :

(i) *private (ii) public*

In C++, the keywords **private** and **public** are called access specifiers. The data hiding concept in C++ is achieved by using the keyword **private.** Private data and functions can only be accessed from within the class itself. Public data and functions are accessible outside the class also. This is shown below :

Data hiding not mean the security technique used for protecting computer databases. The security measure is used to protect unauthorized users from performing any operation (read/write or modify) on the data.

The data declared under **Private** section are hidden and safe from accidental manipulation. Though the user can use the private data but not by accident.



b. What is friend function? What are merits and demerits of using friend functions? Show by an example how friend function is used in C++.

Answer:

A non member function cannot access the private data member of the class. But this is made possible using a friend function. A friend function is that which does not belong to the class still has complete access to the private data members of the class. We simply need to declare the function inside the class using the keyword "friend". The syntax for declaring the friend function is as follows:

```
class demo
{
public:
```

```
friend void func (void);
}
```

The characteristics of a friend function are as follows.

1. It can be declared both in the private and the public part of the class without altering the meaning.
2. It is not called using the object of the class.
3. Unlike member functions , it cannot access the member names directly. Therefore to access the data members of the class, it needs to use the object name, the dot operator and the data member's name. example obj.xyz
4. It is invoked like a normal function.
5. ☐It does not belong to the class

Example program:

```
#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;
}
```

Merits of friend function:

- We can access the other class members in our class with the help of friend functions.
- We CAN access the members without inheriting the class.
- Friend function is used to increase versatility of overload operator.

Demerits of friend function:

- Maximum size of the memory will occupied by objects according to the size of friend Members.
- We can't do any run time polymorphism concepts in those members.

Q.5 a. What are destructors? List atleast five special characteristics of the destructors.

Answer:

Destructor: A destructor , as the name implies, is used to destroy the objects that have been created by the constructor.

Declaration of a Destructor :

The syntax for declaring a destructor is :

```
-name_of_the_class()
{
}
```

So the name of the class and destructor is same but it is prefixed with a ~ (tilde). It does not take any parameter nor does it return any value. Overloading a destructor is not possible and can be explicitly invoked. In other words, a class can have only one destructor. A destructor can be defined outside the class. The following program illustrates this concept :

```
//Illustration of the working of Destructor function
#include<iostream.h>
#include<conio.h>
class add
{
private :
int num1,num2,num3;
public :
add(int=0, int=0); //default argument constructor
//to reduce the number of constructors
void sum();
void display();
~ add(void); //Destructor
};
//Destructor definition ~add()
Add:: ~add(void) //destructor called automatically at end of program
{
Num1=num2=num3=0;
cout<<"\nAfter the final execution, me, the object has entered in the"
<<"\ndestructor to destroy myself\n";
}
```

```
//Constructor definition add()
Add::add(int n1,int n2)
{
num1=n1;
num2=n2;
num3=0;
}
//function definition sum ()
void add::sum()
{
num3=num1+num2;
}
//function definition display ()
Void add::display ()
{
cout<<"\nThe sum of two numbers is "<<num3<<end1;
}
void main()
{
Add obj1,obj2(5),obj3(10,20): //objects created and initialized clrscr();
Obj1.sum(); //function call
Obj2.sum();
Obj3.sum(); cout<<"\nUsing obj1 \n";
obj1.display(); //function call
cout<<"\nUsing obj2 \n";
obj2.display();
cout<<"\nUsing obj3 \n";
obj3.display(); }
```

Special Characteristics of Destructors :

(i) These are called automatically when the objects are destroyed.

(ii) Destructor functions follow the usual access rules as other member functions.

(iii) These **de-initialize** each object before the object goes out of scope.

(iv) No argument and return type (even void) permitted with destructors.

(v) These cannot be inherited.

(vi) **Static** destructors are not allowed.

(vii) Address of a destructor cannot be taken.

(viii) A destructor can call member functions of its class.

(ix) An object of a class having a destructor cannot be a member of a union

b. What is operator overloading? Why is it necessary to overload an operator? List atleast four rules for operator overloading.

Answer:

Overloading: The process of making a function or an operator behave in different manners is known as overloading. Function overloading means the use of same function name to perform a variety of different tasks. The correct function to be invoked is selected by seeing the number and type of the arguments but not the function type.

Operator overloading and its need:

Operator overloading refers to adding a special meaning to an operator. The operator doesn't lose its original meaning but the new meaning is added for that particular class.

Operator overloading is one of the most exciting features of C++. It is helpful in enhancement of the power of extensibility of C++ language. Operator overloading redefines the C++ language. User defined data types are made to behave like built-in data types in C++. Operators +, *. <=, += etc. can be given additional meanings when applied on user defined data types using operator overloading. The mechanism of providing such an additional meaning to an operator is known as operator overloading in C++.

For example, we can add two strings as:

String1+String2=string3;

"good"+ "girl"= "goodgirl";

An example program of function overloading:

```
//Function volume is overloaded three times.
#include<iostream.h>
using namespace std;
int vol(int);
double vol(double,int);
long vol(int,int,int);
int main()
{
cout<<volume(2)<<"\n";
cout<<volume(2.5,8)<<"\n";
cout<<volume(2,3,5)<<"\n";
return 0;
}
int volume(int s)
{
return(s*s*s);
}
double volume(double r,int h)
{
return(3.14519*r*r*h);
}
long volume(int l,int b,int h)
{
return(l*b*h);
}
```

The output would be:

8

157.26

30

Need of Operator overloading:

It provides a flexible option for creation of new definitions for most of the c++ operator.

Rules for operator overloading:

(i) Only existing operators can be overloaded.

(ii) The overloaded operator must have at least one operand that is of user defined type.

(iii) We cannot used friend functions to overload certain operators.

(iv) We cannot change the basic meaning of an operator. i.e. we can't redefine the plus(+) operator to subtract one value from other.(v) Overloaded operators follow the syntax rules of the original operators.

Steps that are involved in the process of overloading:

• Creates a class that defines the data type that is to be used in the overloading operation.

• Declare the operator function operator op() in the public part of a class.

• Define the operator function to implement the required operation.

Restriction and limitations of overloading operators:

- Operator function must be member functions are friend functions.
- The overloading operator must have atleast one operand that is of user defined datatype.

Q.6 a. What is multiple inheritance? Discuss the syntax and rules of multiple inheritance in C++. How can you pass parameters to the constructors of base classes in multiple inheritance? Explain with suitable example.

Answer:

C++ provides 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 they appear 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 th order in which they are declared.

Example Program:

```
#include <iostream>
 using namespace std;
// Base class Shape
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
    int width;
    int height;
};
// Base class PaintCost
class PaintCost
{
  public:
    int getCost(int area)
    {
      return area * 70;
    }};
```

```
// Derived class
class Rectangle: public Shape, public PaintCost
{
  public:
    int getArea()
    {
      return (width * height);
    }
};
int main(void)
{
  Rectangle Rect;
  int area;
  Rect.setWidth(5);
  Rect.setHeight(7);
  area = Rect.getArea();
  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;
  // Print the total cost of painting
  cout << "Total paint cost: $" << Rect.getCost(area) << endl;
  return 0;
}
```

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

```
Total area: 35
Total paint cost: $2450
```

Example Program:

```
#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

b. How does inheritance influence the working of constructor and destructor? Given the following set of definitions.

```
class x
        {
        };
        class y: public x
        {
        };
        class z: public y
        {
        };
        z obj;
```

What order will the constructor and destructor be invoked?

Answer:

Inheritance has a great deal of influence over the working of constructors and destructors. In case of inheritance the base class constructor is invoked first and then the derived class instructor. In case of multiple inheritance, the constructors of base classes are executed in the order in which they occur in the declaration in the derived class. Similarly in case of multilevel inheritance, the base class constructors are executed in the order of their inheritance. Destructors in case of inheritance are executed exactly in the opposite order in which constructors are executed. Likewise in case of multilevel inheritance destructors are executed in the reverse of that is, from derived to base.

```
class x
{
};
class y: public x
{
```

```
};
class z: public y
{
};
z obj;
```

In the above definitions, the contructor will be fired in the following sequence class

x _ class y _ class z

similarly, destructors will be fired exactly in the reverse sequence class

z _ class y _ class x

Q.7 a. Explain the meaning of polymorphism. Describe how polymorphism is accomplished in C++ taking a suitable example.

Answer:

Polymorphism is the property of representing one operation into many different forms. One operation may express different in different situations. The process of making a function or an operator behave in different manners is known as overloading the function or the operator. Polymorphism is one of the most useful features of object oriented programming. It can be achieved both at run time and at compile time.

At **compile time polymorphism** is achieved using function overloading and operator overloading. At the time of compilation the compiler knows about the exact matching as to which function to call or invoke. This is known as compile time polymorphism or early binding.

Polymorphism can also be achieved at **run time**. This is done using the concept of virtual functions. Which class function is to be invoked is decided at run time and then the corresponding object of that class is created accordingly. This is also called as late binding. The following example program explains how polymorphism can be accomplished using function overloading.

```
//Function volume is overloaded three times.
#include<iostream.h>
int volume(int);
double volume(double,int);
long volume(long,int,int);
```

```
int main()
{
cout<<volume(10)<<"\n";
cout<<volume(2.5,8)<<"\n";
cout<<volume(100L,75,15)<<"\n";
return 0;
}
int volume(int s)
{
return(s*s*s);
}
double volume(double r,int h)
{
return(3.14519*r*r*h);
}
long volume(long l,int b,int h)
{
return(l*b*h);
}
```

The output would be:

1000

157.26

112500

b. What is an exception? How is it handled in C++? What are the advantages of using exception handling mechanism in a program?

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' blocks 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 below illustrates the working of the three blocks.

```
#include<iostream.h>
using namespace std;
int main()
{
cout<<"\n Start "<<endl;
try
{
cout<<"\n Inside try block "<<endl;
throw 10;
cout<<"\n this will not execute ";
}
catch(int i)
{
cout<<"\n catch number "<<endl;
cout<<i<<endl;
}
cout<<"End";
return 0;
}
```

Q.8 a. What do you mean by template in C++? Briefly explain its various types. List various limitations of using a template.

Answer:

Templates are a feature of the C++ programming language that allow functions and classes to operate with generic types. This allows a function or class to work on many

different data types without being rewritten for each one. This is effectively a Turing complete language.

Templates are of great utility to programmers in C++, especially when combined with multiple inheritance and operator overloading. The C++ Standard Library provides many useful functions within a framework of connected templates.

There are two kinds of templates: *function templates* and *class templates*.

Function templates

A *function template* behaves like a function except that the template can have arguments of many different types (see example). In other words, a function template represents a family of functions. The format for declaring function templates with type parameters is

template <class identifier> function_declaration;

template <typename identifier> function_declaration;

Both expressions have exactly the same meaning and behave exactly the same way. The latter form was introduced to avoid confusion because a type parameter does not need to be a *class*, it may also be a basic type like *int* or *double*.

For example, the C++ Standard Library contains the function template max(x, y) which returns either *x* or *y*, whichever is larger. max() could be defined like this, using the following template:

```
template <typename Type>
Type max(Type a, Type b)
{
   return a > b ? a : b;
}
```

This single function definition works with different kinds of data types. A function template does not occupy space in memory. The actual definitions of a function template are generated at compile-time, when the compiler has determined what types the function will be called for. The function template does not save memory.

```
#include <iostream>
int main()
{
  // This will call max <int> (by argument deduction)
  std::cout << max(3, 7) << std::endl;
  // This will call max<double> (by argument deduction)
  std::cout << max(3.0, 7.0) << std::endl;
  // This type is ambiguous, so explicitly instantiate max<double>
  std::cout << max<double>(3, 7.0) << std::endl;
  return 0;
}
```

In the first two cases, the template argument T is automatically deduced by the compiler to be int and double, respectively. In the third case deduction fails because the type of the parameters must in general match the template arguments exactly. This function template can be instantiated with any copy-constructible type for which the expression (y < x) is valid. For user-defined types, this implies that the less-than operator must be overloaded.

Class templates

A class template provides a specification for generating classes based on parameters. Class templates are commonly used to implement containers. A class template is instantiated by passing a given set of types to it as template arguments. The C++ Standard Library contains many class templates, in particular the containers adapted from the Standard Template Library, such as vector.

Explicit template specialization

When a function or class is instantiated from a template, a specialization of that template is created by the compiler for the set of arguments used (and the specialization is referred to as being a generated specialization). However, the programmer may decide to implement a special version of a function (or class) for a given set of template arguments which is called an explicit specialization. If a class template is specialized by a subset of its parameters it is called partial template specialization. If all of the parameters are specialized it is a *full specialization*. Function templates cannot be partially specialized.

Explicit specialization is used when the behavior of a function or class for particul choices of the template parameters must deviate from the generic behavior: that is, from the code generated by the main template, or templates. For example:

```
template <>
bool max(bool a, bool b) {
   return a||b;
}
```

Templates are considered type-safe; that is, they require type-checking at compile time. Hence, the compiler can determine at compile time whether the type associated with a template definition can perform all of the functions required by that template definition.

By design, templates can be utilized in very complex problem spaces, whereas macros are substantially more limited.

Fundamental drawbacks to the use of templates:

1. Some compilers exhibited poor support for templates. So, the use of templates could decrease code portability.
2. Many compilers lack clear instructions when they detect a template definition error. This can increase the effort of developing templates, and has prompted the development of Concepts for possible inclusion in a future C++ standard.
3. Since the compiler generates additional code for each template type, indiscriminate use of templates can lead to code bloat, resulting in larger executables.
4. Because a template by its nature exposes its implementation, injudicious use in large systems can lead to longer build times.
5. It can be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.
6. Templates of Templates (nesting) are not supported by all compilers, or might have a max nesting level.
7. Templates are in the headers, which require a complete rebuild of all project pieces when changes are made.

8. No information hiding. All code is exposed in the header file. No one library ca
   solely contain the code.

b. Write a function template for sorting a list of arrays.

Answer:

```
#include<iostream.h>
using namespace std;
template<class T>
void sort(T a[],int n)
{
for(int i=0;i<n-1;i++)
for(int j=i+1;j<n;j++)
{ if(a[i]>a[j])
swap(a[i],a[j]);
}
}
template<class X>
void swap(X &a,X &b)
{
X temp=a;
a=b;
b=temp;
}
int main()
{
int v[5]={32,78,12,98,56};
sort(v,5);
cout<<"\n the list of sorted array is :- ";
for(int i=0;i<5;i++)
cout<<"\n"<<v[i];
```

```
return 0;
}
```

Q.9 a. Describe the concept of error handling during file operations. Explain various error handling functions in detail with the help of an example.

Answer:

Error Handling during File Operations:

There are many problems encountered while dealing with files like

- a file which we are attempting to open for reading does not exist.
- The file name used for a new file may already exist.
- We are attempting an invalid operation such as reading past the end of file.
- There may not be any space in the disk for storing more data.
- We may use invalid file name.
- We may attempt to perform an operation when the file is not opened for that purpose.The C++ file stream inherits a 'stream-state 'member from the class ios.This member records information on the status of a file that is being currently used.The stream state member uses bit fields to store the status of error conditions stated above.The class ios support several member functions that can be used to read the status recorded in a file stream.

Error Handling Functions :

Function	Return value and meaning
eof()	Returns true(non zero value) if end of file is encountered while reading otherwise returns false(zero).
fail()	Returns true when an input or output operation has failed .
bad()	Returns true if an invalid operation is attempted or any unrecoverable error has occurred.However,if it is false,it may be possible to recover from any other error reported and continues operation.
good()	Returns true if no error has occurred.This means all the above functions are false.For instance,if file.good() is true.all is well with the stream file and we can proceed to perform I/O operations.When it returns false,no further operations is carried out.

Example:

```
……
……..
ifstream infile;
infile.open("ABC");
while(!infile.fail())
{
……
…….. (process the file)
…….
}
if (infile.eof())
{
……(terminate the program normally)
}
else
if (infile.bad())
{
…….(report fatal error)
}
else
{
infile.clear(); //clear error state
…….
…….
}
……..
……..
```

The function **clear()** resets the error state so that further operations can be attempted.

Text Book

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