Q.2 a. Define Object-oriented programming. List and explain various features of Object-oriented programming paradigm. (8)

Answer:

1. **Inheritance:** Inheritance as the name suggests is the concept of inheriting or deriving properties of an exiting class to get new class or classes. In other words we may have common features or characteristics that may be needed by number of classes. So those features can be placed in a common tree class called base class and the other classes which have these characteristics can take the tree class and define only the new things that they have on their own in their classes. These classes are called derived class. The main advantage of using this concept of inheritance in Object oriented programming is it helps in reducing the code size since the common characteristic is placed separately called as base class and it is just referred in the derived class. This provide the users the important usage of terminology called as reusability.

2. Polymorphism and overloading: Poly refers many. So Polymorphism as the name suggests is a certain item appearing in different forms or ways. That is making a function or operator to act in different forms depending on the place they are present is called Polymorphism. Overloading is a kind of polymorphism. In other words say for instance we know that +, - operate on integer data type and is used to perform arithmetic additions and subtractions. But operator overloading is one in which we define new operations to these operators and make them operate on different data types in other words overloading the existing functionality with new one. This is a very important feature of object oriented programming methodology which extended the handling of data type and operations.

3. Data Hiding: This concept is the main heart of an Object oriented programming. The data is hidden inside the class by declaring it as private inside the class. When data or functions are defined as private it can be accessed only by the class in which it is defined. When data or functions are defined as public then it can be accessed anywhere outside the class. Object Oriented programming gives importance to protecting data which in any system. This is done by declaring data as private and making it accessible only to the class in which it is defined. This concept is called data hiding. But one can keep member functions as public.

4. Encapsulation: The technical term for combining data and functions together as a bundle is encapsulation.

5. Reusability: Reusability is nothing but re- usage of structure without changing the existing one but adding new features or characteristics to it. It is very much needed for any programmers in different situations. Reusability gives the following advantages to users. It helps in reducing the code size since classes can be just derived from existing one and one need to add only the new features and it helps users to save their time.

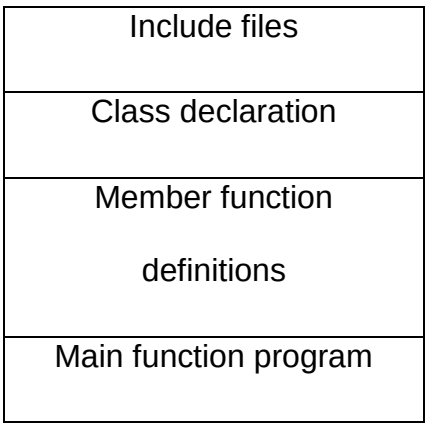
b. Explain in brief the structure of a C++ program. (4)

Answer:

Structure of C++ Program:

A typical C++ program contains four sections as shown in fig. These sections may be placed in separate code files and then compiled independently or jointly.

Generally a C++ program is organized into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface from the implementation details (member function definition). Finally, the main program that uses the class is places in a third file which “includes” the previous two files as well as any other file required.



c. Write a program in C++ which generates the following pattern (4)

5 5 5 5 5
4 4 4 4

3 3 3
2 2
1

Answer:

```
# include <  iostream . h>
        # include < conio . h>
         void main  ( )
                { int   i  , j  ;
                   clrscr   ( );
        for   (i= 5;  i >  = 1; i --)
        {
        for   (j=1  ;  j<=I ; j++)
                cout   <<  i;
          cout   << endl;
                }
           getch    (  );
                }
```

Q.3 a. Write a program in C++ for addition of two matrices. (8)

Answer:

```
# include < iostream . h>
        # include < conio . h>
void main ( )
        {
        int  a [2] [2] , b[2,2] , s [2,2];
        clrscr  (  ) ;
        cout << "Enter  first matrix : " << endl;
        for (int i = 0;  i<=1; j + +)
         for (int j = 0;  i<=1; j + +)
        {
        cout <<  " Enter" <<i+1             << j+1    << " element :";
        cin >>   a[i]  [j] ;
        }
        cout << " Enter second matrix:           " <<end l;
        for (i   =0;    i<   =1;   i++)
        for (j   =0;    j<   =1;   j++)
        {
         cout  << " Enter         " << i+1  << j+1   << "element:"
        cin >> b [i]  [j]    ;
          }
        for (i  =0 ;  i < = 1; i ++)
        for (j  =0 ;  j < = 1; j ++)
        s [i]  [j]   = a [i] [j]  + b   [i] [j];
        cout << "The addition matrix is :" << endl;
        for (i   =0, i   < 1;    i + +)
         for (j   =0, j   <= 1;    j + +)
        cout << s [i]  [j]  <<  '\t';
        cout << endl ;   }
```

```
getch ( ) ;
  }
```

b. How do structures in C and C++ differ? Explain with the help of an example. (4)

Answer:

Following are the differences between Structures in C and Structures in C++ :

- Structures in C cannot have Direct functions/methods inside a structure definition. But it still can have methods in the form of function pointers. While structures in C++ can have functions/methods inside a structure definition.
- In C, an object(variable) of a structure is created using the keyword struct(otherwise syntax error). For example struct student sid; Whereas in C++ the struct keyword can be omitted while creating structure variables(objects). For example: student sid;
- C structures does not permit Data hiding concept whereas C++ structures allow Data hiding.

c. Write a program code in C++ for generating a fibonacci series for more than one number. (4)

Answer:

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

```
# include < conio . h>
 void main  ( )
        {       int   n  ;
                int   F1,   F2,   F3  ;
                  clrscr   (   );
cout  << " Enter how many no. series you need.") ;
                cin >> n;
                F1  = 1;          F2  = 1;
cout   <<  F1   << endl;
cout   <<  F2   << endl;
int c =2;
while (c  <  n)
  {     F3   =  F1+F2 ;
        F1   =  F2;
        F2  =  F3;
        cout   << F3  <<  endl ;
        c++;
        }
        getch    ( ) ;
        }
```

Q.4 a. Differentiate between *Call by value* and *Call by reference* by taking suitable example. (8)

Answer:

Call by Value: - In this method the values of the actual parameters (appearing in the function call) are copied into the formal parameters (appearing in the function definition), i.e., the function creates its own copy of argument values and operates on them. The following program illustrates this concept :

```
//calculation of compound interest using a function
#include<iostream.h>
#include<conio.h>
#include<math.h> //for pow()function
void main()
{
float principal, rate, time; //local variables
void calculate (float, float, float); //function prototype clrscr();
cout<<"\nEnter the following values:\n";
cout<<"\nPrincipal:";
cin>>principal;
cout<<"\nRate of interest:";
cin>>rate;
cout<<"\nTime period (in yeaers) :";
cin>>time;
calculate (principal, rate, time); //function call
getch ();
}
//function definition calculate()
void calculate (float p, float r, float t)
{
float interest; //local variable
interest = p* (pow((1+r/100.0),t))-p;
cout<<"\nCompound interest is : "<<interest;
}
```

Call by Reference: - 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 definition swap()
void swap (int & a, int & b)
{
int temp=a;
a=b;
b=temp;
}
```

b. What is Inline function? What are advantages and disadvantages of using Inline function? Write an Inline function in C++ that returns maximum of two numbers. (8)

Answer:

- C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time. i.e. Inline function is the optimization technique used by the compilers. One can simply prepend inline keyword to function prototype to make a function inline. Inline function instruct compiler to insert complete body of the function wherever that function got used in code.
- Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.
- To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.
- A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

Advantages :-

1)It does not require function calling overhead.

2)It also save overhead of variables push/pop on the stack, while function calling.

3)It also save overhead of return call from a function.

4) It increases locality of reference by utilizing instruction cache.

5) After in-lining compiler can also apply intraprocedural optmization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..

Disadvantages:

1) May increase function size so that it may not fit on the cache, causing lots of cache miss.

2) After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.

3) It may cause compilation overhead as if some body changes code inside inline function
than all calling location will also be compiled.
4) If used in header file, it will make your header file size large and may also make it unreadable.
5) If somebody used too many inline function resultant in a larger code size than it may cause thrashing in memory. More and more number of page fault bringing down your program performance.
6) Its not useful for embeded system where large binary size is not preferred at all due to memory size constraints.
Following is an example, which makes use of inline function to return max of two numbers:

```
#include <iostream>
using namespace std;
inline int Max(int x, int y)
{
   return (x > y)? x : y;
}

// Main function for the program
int main( )
{

   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0; #
}
```

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

```
Max (20,10): 20
Max (0,200): 200
```

```
Max (100,1010): 1010
```

Q.5 a. What is constructor? Is it mandatory to use constructor in a class? List five special characteristics of the constructor functions. (8)

Answer:

Declaration and Definition of a Constructor:-

A constructor (having the same name as that of the class) is a member function which is automatically used to initialize the objects of the class type with legal initial values.

It is defined like other member functions of the class, i.e., either inside the class definition or outside the class definition.

For example, the following program illustrates the concept of a constructor :

```
#include <iostram.h>
#include <conio.h>
Class rectangle
{
private :
float length, breadth;
public:
rectangle ()//constructor definition
{
//displayed whenever an object is created
cout<<”I am in the constructor”;
length =10.0;
breadth =20.5;
}
float area()
{
return (length*breadth);
}
};
void main()
{
clrscr();
rectangle rect; //object declared
       cout<<”\nThe area of the rectangle with default
       parameters is:”<<rect.area()<<”sq.units\n”;
getch();}
```

SPECIAL CHARACTERISTICS OF CONSTRUCTORS:

These have some special characteristics. These are given below:

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

(ii) All objects of the class having a constructor are initialized before some use.

(iii) These should be declared in the public section for availability to all the functions.

(iv) Return type (not even void) cannot be specified for constructors.

(v) These cannot be inherited, but a derived class can call the base class constructor.

(vi) These cannot be static.

(vii) Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.

(viii) These can have default arguments as other C++ functions.

(ix) A constructor can call member functions of its class.

(x) An object of a class with a constructor cannot be used as a member of a union.

(xi) A constructor can call member functions of its class.

(xii) We can use a constructor to create new objects of its class type by using the syntax.

```
Name_of_the_class (expresson_list)
For example,
Employee obj3 = obj2; // see program 10.5
Or even
          Employee obj3 = employee (1002, 35000); //explicit
          call
```

(xiii) The make implicit calls to the memory allocation and deallocation operators new and delete.

(xiv) These cannot be virtual.

b. What is operator overloading? Explain with the help of suitable example. List various exceptions to operator overloading. (8)

Answer:

Operator overloading is one of the advanced concepts of C++. It is a feature through which most of the standard operators can be used with class objects.

When we use an expression like ’2 +3′, we know that the answer will be the sum of two integers. This is because the compiler knows how to interpret the + operator when used with integers. But, what if we want to do something like ‘obj1 = obj2 + obj3′ (where all these are objects of same class) ? How + operator should work in this case? The answer is through Operator Overloading.

Operator Overloading in C++

Consider the following class :

```
class example
{
public:
```

```
    int a;
    int b;
};
```

For creation of three objects of this class and assigning the sum of two of these objects to the third one is coded as:

```
example obj1, obj2, obj3;

    obj1.a = 1;
    obj1.b = 1;

    obj2.a = 2;
    obj2.b = 2;

    obj3.a = 0;
    obj3.b = 0;

    obj3 = obj1 + obj2;
```

If addition of two objects and the addition of corresponding integer members is required For example, something like this :

```
obj3.a = obj1.a + obj2.a;
obj3.b = obj1.b + obj2.b
```

This is what exactly can be done through operator overloading. So we need to overload these operators in order to achieve exactly what is represented in the above two lines.

Now, the question arises, how to overload the operators?

Declaration of an overloaded operator +:

```
 example operator+(const example& obj);
```

This declaration should be made part of class example. Similarly, we can overload = operator.

Operator Overloading Working Example

```
#include <iostream>
class example
{
public:
    int a;
    int b;
    example operator+(const example& obj);
    void operator=(const example& obj);
};

void example::operator=(const example& obj)
{
    (*this).a = obj.a;
    (*this).b = obj.b;

    return;
}
```

```
example example::operator+(const example& obj2)
{
    example tmp_obj = *this;
    tmp_obj.a = tmp_obj.a + obj2.a;
    tmp_obj.b = tmp_obj.b + obj2.b;
    return tmp_obj;
}

int main(void)
{
    example obj1, obj2, obj3;

    obj1.a = 1;
    obj1.b = 1;

    obj2.a = 2;
    obj2.b = 2;

    obj3.a = 0;
    obj3.b = 0;

    obj3 = obj1 + obj2;

    std::cout<<obj3.a<<"  "<<obj3.b<<"\n";

    return 0;
}
```

In the example above :

When ‘obj1 + obj2′ is encountered, function corresponding to overloaded operator + is called. We can think of ‘obj1 + obj2′ as something like ‘obj1.add(obj2)’. The function corresponding to overloaded operator + is called in context of obj1 and hence only obj2 is needed to be passed as argument. obj1 can be accessed through ‘this’ pointer in that function. Here in this function, individual integer member is added and the resultant object is returned.

Similarly, every thing happens the same way when the resultant object of the sum of obj1 and obj2 is assigned to obj3 through overloaded = operator. Each integer member of class is assigned to corresponding member of obj3.

The output of this program is :

3 3

Thus the + and = operators worked exactly in the similar way as they work for standard types. Through operator overloading the code becomes relatively neat and easy to maintain.

Exception to Operator Overloading

Though most of the operators can be overloaded, there are certain operators that can not be overloaded. These are :

1. dot (.) operator
2. sizeof operator
3. Scope resolution (::) operator
4. Arithmetic if (?:) operator
5. (.*) operator

Q.6 a. What does inheritance mean in C++? Explain the ambiguity problem that occurs in single inheritance. Write a program to avoid that ambiguity. (8)

Answer:

AMBIGUITY PROBLEM IN SINGLE INHERITANCE

The major problem that occurs with the single inheritance is ambiguity. So whenever data members and member functions are defined with the same name in both the base and the derived classes, then the name creates an ambiguity in defining the single class. So to remove this problem, these names must be without ambiguity. The scope resolution operator :: may be used to refer to any base member explicitly. This allows access to a name that has been redefined in derived class. For example, following program segment illustrates how ambiguity occurs when the getdata () member function is accessed from the main () program:

```
class base A
{
        public :
                int a, b, c ;
                getdata ( );
        } ;
class base B
        {
        public :
                int a,b,c;
                getdata ();
        };

class derived C: public base A, public base B
```

```
{
public :
        int a, b, c ;
        getdata ();
};
void main ()
{
derived C obj ;
obj. getdata ()
}
```

Members are ambiguous without scope operator. When the member function getdata() is accessed by the class object, the compiler cannot distinguish between the member function of the base class base A and the base class bas B. So it is essential to declare the scope resolution operator explicitly to call a base class member as represented below:

```
obj. base A: : getdata ( ) ;
obj. base B: : getdata ( ) ;
```

Program to avoid ambiguity in single inheritance.

```
#include <iostream.h>
class A
        {
        private :
                int i ;
        public  :
                void getdata (int x) ;
                void display ();
        };
        class B
        {
                private:
                        int i;
```

```
                public :
                void getdata (int y);
                void display ();
                };
class C : : public A, public B
        {
        };
void A : : getdata (int x)
        {
                i   = x ;
        }
void A : : display ()
        {
        cout <<"\n value of i is ="<<i;
        }
void B : : getdata (int y)
        {
                j=y;
        }
void B : : display ()
        {
        cout<<"\n value of j is ="<<j;
        }
void main ()
        {
        class c obj;
        int x,y;
        count <<"\n Enter the of x";
        cin>>x;
        obj. A : : getdata (x);
        //Here member is ambiguous without scope
        cout<<"\n Enter the value of Y";
```

```
cin>>y;
obj. B : : getdata (y);
obj. A : : display ();
obj. B : : display ();
getch ();
}
```

b. List various limitations of Inheritance. (8)

Answer:

Disadvantages of Inheritance

Inspite of the fact that Inheritance is the key concept of object oriented programming and provide many advantages, it suffers from many limitations also. The following are the limitations of inheritance.

1. **Increase Program complexity**: Although inheritance is generally recommended as a solution for complicated projects but overuse or improper use of inheritance can simply increase the complexity and reduce the understandability.
2. **Improper utilization of memory**: In inheritance, there are certain members in the base class that are not used at all, however memory is allocated to them. Thus, there is improper utilization of memory for certain members of the base class.
3. **Extra time needed for understating class libraries**: In some situations, reusing a complex library in your program may require extra time to understand its interfaces and correct usage of the classes contained in the libraries which slows down your initial design and coding. So it is better to write all the necessary code by yourself in order to understand exactly how it works.
4. **Compiler overheads**: Invoking member functions of the classes involved in inheritance cre4ates more compiler overheads and increases execution time compared to the invocation of simple functions. However, it can be compromised against the large benefits of inheritance
5. **Proper understanding of hierarchy**: Deriving new classes from existing classes is not always as straight forward as it might initially appear. Mistakes in an inheritance hierarchy can cripple an object model, so deep understanding of inheritance and its effects is crucial.

Q.7 a. What is Polymorphism? Explain in details various types of Polymorphism. (8)

Answer:

Polymorphism:

Polymorphism is the phenomenon where the same message sent to two different objects produces two different set of actions. Polymorphism is broadly divided into two parts:

Static polymorphism – exhibited by overloaded functions.

Dynamic polymorphism – exhibited by using late binding.

Static Polymorphism

Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called early binding. The term early binding stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.

The resolution of a function call is based on number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration:

```
void add(int , int);
void add(float, float);
```

When the add() function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Dynamic Polymorphism:

Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed. The term late binding refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Static Vs Dynamic Polymorphism:

- Static polymorphism is considered more efficient, and dynamic polymorphism more flexible.
- Statically bound methods are those methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run-time.

This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statically bound functions.

- As applications are becoming larger and more complicated, the need for flexibility is increasing rapidly. Most users have to periodically upgrade their software, and this could become a very tedious task if static polymorphism is applied. This is because any change in requirements requires a major modification in the code. In the case of dynamic binding, the function calls are resolved at run-time, thereby giving the user the flexibility to alter the call without having to modify the code.

To the programmer, efficiency and performance would probably be a primary concern, but to the user, flexibility or maintainability may be much more important. The decision is thus a trade-off between efficiency and flexibility.

b. What is an exception? Explain the mechanism of throwing and rethrowing exceptions. (8)

Answer:

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.
To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

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

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
```

```
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

Output is:

```
An exception occurred. Exception Nr. 20
```

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.
The exception handler is declared with the catch keyword immediately after the closing brace of the try block. The syntax for catch is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught by that handler.
Multiple handlers (i.e., catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the throw statement is executed.
If an ellipsis (...) is used as the parameter of catch, that handler will catch any exception no matter what the type of the exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case, the last handler would catch any exception thrown of a type that is neither int nor char. After an exception has been handled the program, execution resumes after the try-catch block, not after the throw statement. It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments.

For example:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Throwing Mechanism

When an exception that is desired to be handled is detected, it is thrown using the throw statement in one of the following forms:

Throw (exception);

Throw exception;

Throw; // used for rethrowing an exception

- The operand object exception may be of any type, including constants. It is also possible to throw objects not intended for error handling.
- When an exception is throw, it will be caught by the catch statement associated with the try block. That is the control exists the current try block, and is transferred to the catch block after that try block.
- Throw point can be in deeply nested scope within a try block or in a deeply nested function call. In any case, control is transferred to the catch statement.

Catching Mechanism:

Code for handling exceptions is included in catch blocks. A catch block looks like a function definition and is of the form

Catch (type arg)

{

\\ statements for

\\ managing exceptions

}

The type indicated the type of exception that catch block handles. The parameter arg in an optional parameter name. Note that the exception-handling code is placed

between two braces. The catch statement catches an exception whose type matches with the type catch argument. When it is caught, the code in the catch block in executed. If the parameter in the catch statement is named, then the parameter can be used in the exception-handling code. After excuting the handler, the control goes to the statement immediately following the catch block.

Due to mismatch, if an exception is not caught, abnormal program termination will accure. It is important to note that the catch block is simply skipped if the catch statement does not catch an exception.

Rethrowing An Exception:

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any argument as shown below:

Throw;

This causes the current exception to be thrown th the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block. Program demonatrates how an exception is rethrown and caught.

```
#include <isotream>
using namespace std;
void divide (double  x,  double  y)
{
        cout  << " inside function\n";
try
        }
                If (y ==0.0)
                        throw y;            // Throwing double
                else
                        cout  << "Division = " << x/y << "\n";
        }
        catch (double)                                  //Catch a double
        {
                        cout <<"Caught double inside function \n";
                        throw;                                  // Rethrowing double
        }
        cout <<"End of function \n\n";
}
int main()
{
        cout <<"Inside main \n";
        try
        {
        divide (10.5.2.0);
        divide(20.0.0.0);
```

```
}

catch (double)
{
        cout << “Caught double inside main\n”;
}
cout << “ End of main\n”;
        return0;
}
```

Q.8 a. Differentiate between Function Template and Class Template. (8)

Answer:

- 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>.
- Templates can be used to define functions as well as classes.

Function Template:

The general form of a template function definition is:

```
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.

The following is the example of a 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
```

Class Template:

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template <class type> class class-name
{
.
.
.
}
```

Here, type is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
  private:
    vector<T> elems;     // elements

  public:
    void push(T const&);  // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const{       // return true if empty.
        return elems.empty();
    }
};

template <class T>
void Stack<T>::push (T const& elem)
{
    // append copy of passed element
    elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
```

```
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::pop(): empty stack");
    }
         // remove last element
    elems.pop_back();
}

template <class T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::top(): empty stack");
    }
         // return copy of last element
    return elems.back();
}
int main()
{
    try {
        Stack<int>         intStack;  // stack of ints
        Stack<string> stringStack;    // stack of strings
        // manipulate int stack
        intStack.push(7);
        cout << intStack.top() <<endl;
        // manipulate string stack
        stringStack.push("hello");
        cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (exception const& ex) {
        cerr << "Exception: " << ex.what() <<endl;
        return -1;
    }
}
```

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

```
7
```

```
hello
Exception: Stack<>::pop(): empty stack
```

b. Explain the concept of Template Specialization. Give example. (8)

Answer:

Template specialization : A template specialization allows a template to make specific implementations when the pattern is of a determined type. For example, suppose that our class template pair included a function to return the result of the module operation between the objects contained in it, but we only want it to work when the contained type is int. For the rest of the types we want this function to return 0. This can be done the following way:

```
// Template specialization
 #include <iostream.h>
template <class T>
class pair {
T value1, value2;
 public:
pair (T first, T second)
{
value1=first; value2=second;
}
T module ()
{return 0;}
};
template <> class pair <int>
{
int value1, value2;
 public: pair (int first, int second)
{
value1=first; value2=second;
}
 int module ();
};
 template <> int pair<int>::module()
 {
return value1%value2;
 }
int main ()
 {
pair <int> myints (100,75);
pair <float> myfloats (100.0,75.0);
 cout << myints.module() << '\n';
cout << myfloats.module() << '\n';
return 0;
```

}

As shown in the code the specialization is defined this way:

template <> class class_name <type>

The specialization is part of a template, for that reason we must begin the declaration with template <>. And indeed because it is a specialization for a concrete type, the generic type cannot be used in it and the first angle-brackets <> must appear empty. After the class name we must include the type that is being specialized enclosed between angle-brackets <>. When we specialize a type of a template we must also define all the members equating them to the specialization (if one pays attention, in the example above we have had to include its own constructor, although it is identical to the one in the generic template). The reason is that no member is "inherited" from the generic template to the specialized one.

Q.9 a. Explain in detail: (8)
(i) Standard I/O in C++
(ii) File I/O in C++

Answere:

Standard input and output :

We can write to the standard output using code like this:

cout << x << y << endl;

The code is evaluated from left to right, so the value of x is written first, then the value of y, then a newline. Therefore, the code given above is equivalent to:

cout << x;

cout << y;

cout << endl;

Similarly, you can read from the standard input using code like this:

cin >> x >> y; which is equivalent to:

cin >> x;

cin >> y; (Again, the code is evaluated from left to right, so variable x will be set to the first input value and variable y will be set to the second input value.) Note:

1. Remember that you must #include <iostream> in order to use cin, cout, or endl.

2. To remember which is the input operator and which is the output operator, think of the angle brackets as pointing in the direction the values are flowing; for example, when we write the value of x using "cout << x", the brackets point to cout (so the

value is going from x to cout). When we read a value into x you use "cin >> x"; this time, the brackets point to x (so the value is going from cin to x).

3. Both the output operator << and the input operator >> are overloaded; they can be used to write/read any of the primitive types. However, the input operator >> skips all whitespace in the input, so we cannot use it to read a whitespace character (e.g., a space, tab, or newline).

4. It is up to the programmer to know what kind of data he/she is going to read. If the data in the input does not match the type of the variable we are reading into, we will either get a runtime error, or the value we read will be garbage. For example, if we run the following program:

```
#include <iostream>
int main()
{
 int x;
 cout << "enter a number: ";
cin >> x; return(0);
 }
```

and we type a letter instead of a number, we will either get a runtime error, or the value of x will be garbage.

File I/O : To read from a file one must use a variable of type ifstream. To write to a file we must use a variable of type ofstream. In both cases, we must open the file before we can read or write.

For example, here's how to open the file named "input.dat" for reading:

```
 #include <fstream>
 ifstream inFile;
 inFile.open("input.dat");
if (inFile.fail())
 {
cerr << "unable to open file input.dat for reading" << endl;
 exit(1);
```

```
}
```

Note that to use files you must #include <fstream> (including iostream is not good enough). Also note that this code writes its error message to cerr; that is the standard error, and should generally be used for error messages instead of the standard output. Once inFile has successfully been opened for reading, we can use the usual input operator to read values:

```
int n, sum = 0;
while (inFile >> n)
 {
 sum += n;
}
```

In this example, each time the while loop condition is evaluated; the next integer in the input file is read into variable x. The while loop condition will evaluate to false when all of the values in the input file have been read. If you prefer to read one character at a time (including whitespace characters), you can use the get operation:

```
 char ch;
 while (inFile.get(c))
 {
...
}
```

In this example, each time the while loop condition is evaluated, the next character in the input file is read into variable ch. As in the previous example, the condition will evaluate to false when there are no more characters in the input.

b. Explain the concept of streams. List and briefly explain various stream classes. (8)

Answer:

Class name	Contents
ios(General input/output stream class)	Contains basic facilities that are used by all other input and output classes Also contains a pointer to buffer object(streambuf object) Declares constants and functions that are necessary for handling formatted input and output operations

istream(input stream)	Inherits the properties of ios Declares input functions such as get(),getline() and read() Contains overloaded extraction operator>>
ostream(output stream)	Inherits the property of ios Declares output functions put() and write() Contains overloaded insertion operator <<
iostream (input/output stream)	Inherits the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions
streambuf	Provides an interface to physical devices through buffer Acts as a base for filebuf class used ios files

Text Book

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