Q.2 a. Explain various fundamental features of the object oriented programming? (6)

Answer:

The fundamentals features of the OOPs are the following:

- **Encapsulation:** It is a mechanism that associates the code and data it manipulates into a single unit and keeps them safe from external interference and misuse. In C++, this is supported by a construct called *class*.
- **Data Abstraction:** The technique of creating new data types that are well suited to an application to be programmed is known as data abstraction. It provides the ability to create user-defined data types, for modeling a real world object, having the properties of built-in data types and a set of permitted operators. The *class* is a construct in C++ for creating user-defined data types call *abstract data types (ADTs).*
- **Inheritance:** It allows the extension and reuse of exiting code without having to rewrite the code from scratch. Inheritance involves the creation of new classes (called derived classes) from the existing ones (called base classes), thus enabling the creation of a hierarchy of classes that simulates the class and subclass of the real world.
- **Multiple Inheritance:** The mechanism by which a class is derived from than one base class is known as multiple inheritance.
- **Polymorphism:** It allows a single name / operator to be associated with different operations depending on the type of data passed to it. In C++, it is achieved by function overloading, operator overloading and dynamic binding (virtual functions).
- **Message Passing:** It is the process of invoking an operation on an object. In response to a message, the corresponding method (function) is executed in the object.
- **Extensibility:** It is a feature, which allows the extension of the functionality of the existing software components. In C++, this is achieved through abstract class and inheritance.
- **Genericity:** It is a technique for defining software components that have more than one interpretation depending on the data types of parameters. In C++, genericity is realized through *function templates* and *class templates.*

b. With the help of an example describe size of C++ operator? (5)

Answer:

It looks like a built-in function, but it is called the sizeof operator. The format of sizeof follows:

sizeof data
or
sizeof(data type)

The sizeof operator is unary, because it operates on a single value. This operator produces a result that represents the size, in bytes, of the data or data type specified. Because most data types and variables require different amounts of internal storage on different computers, the sizeof operator enables programs to maintain consistency on different types of computers.

The sizeof operator is sometimes called a *compile-time operator.* At compile time, rather than runtime, the compiler replaces each occurrence of sizeof in your program with an unsigned integer value.

If you use an array as the sizeof argument, C++ returns the number of bytes you originally reserved for that array. Data inside the array have nothing to do with its returned sizeof value—even if it's only a character array containing a short string.

Suppose you want to know the size, in bytes, of floating point variables for your computer. You can determine this by entering the keyword float in parentheses—after sizeof—as shown in the following program.

```
#include <iostream.h>
main() {
```

© IETE

```
        cout << "The size of floating-point variables on \n";
        cout << "this computer is " << sizeof(float) << "\n";
        return 0;
}
```

Expected Output:
The size of floating-point variables on this computer is: 4

c. Explain the difference between 'A' and "A" with suitable example. (3+2)

Answer:

The notations 'A' and "A" have an important difference. The first one ('A') is a character constant while the second ("A") is a string constant. The notation 'A' is a constant occupying a single byte containing the ASCII code of character A. The notation "A" on the other hand, is a constant that occupies two bytes, one for the ASCII code of A and the other for the null character with value 0, that terminates all strings.

Q.3 a. Explain the use of *break* statement in *switch-case* statement? (4)

Answer:

The switch Statement

The **switch** and **case** statements help control complex conditional and branching operations. The **switch** statement transfers control to a statement within its body. The syntax for switch statement is as follows:

selection-statement :
***switch (** expression **)** statement*
labeled-statement :
***case** constant-expression **:** statement*
***default :** statement*

Control passes to the statement whose **case** *constant-expression* matches the value of **switch** (expression). The **switch** statement can include any number of **case** instances, but no two case constants within the same **switch** statement can have the same value. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a **break** statement transfers control out of the body. Use of the **switch** statement usually looks something like this:

```
switch ( expression )
{
  declarations
  .
  case constant-expression :

    statements executed if the expression equals the
   value of this constant-expression
    .
    .
    .    break;
  default :
    statements executed if expression does not equal
    any case constant-expression
}
```

We can use the **break** statement to end processing of a particular case within the **switch** statement and to branch to the end of the **switch** statement. Without **break**, the program continues to the next case,

executing the statements until a **break** or the end of the statement is reached. In some situations, this continuation may be desirable.
The **default** statement is executed if no **case** *constant-expression* is equal to the value of **switch (** *expression* **)**. If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body are executed. There can be at most one **default** statement. The **default** statement need not come at the end; it can appear anywhere in the body of the **switch** statement. In fact it is often more efficient if it appears at the beginning of the **switch** statement. A **case** or **default** label can only appear inside a **switch** statement.
The type of **switch** *expression* and **case** *constant-expression* must be integral. The value of each **case** *constant-expression* must be unique within the statement body.
The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines where execution starts in the statement body. Switch statements can be nested. Any static variables are initialized before executing into any **switch** statements.

The following examples illustrate **switch** statements:

```
switch( c )
{
   case 'A':
      capa++;
   case 'a':
      lettera++;
default :
      total++;
}
```

All three statements of the **switch** body in this example are executed if c is equal to 'A' since a **break** statement does not appear before the following case. Execution control is transferred to the first statement (capa++;) and continues in order through the rest of the body. If c is equal to 'a', lettera and total are incremented. Only total is incremented if c is not equal to 'A' or 'a'.

```
switch( i ) {
   case -1:
      n++;
break;
   case 0 :
      z++;
      break;
   case 1 :
      p++;
      break;
}
```

In this example, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the statement body after one statement is executed. If i is equal to 1, only n is incremented. The **break** following the statement n++; causes execution control to pass out of the statement body, bypassing the remaining statements. Similarly, if i is equal to 0, only z is incremented; if i is equal to 1, only p is incremented. The final **break** statement is not strictly necessary, since control passes out of the body at the end of the compound statement.

b. Write the syntax for *initialization at definition* of two-dimensional array. Give one example also. (4)

Answer:
A two-dimensional array can be initialized during its definition as follows:

```
data-type array-name[row-size][col-size] = {
        { elements of first row},
        {elements of second row},
        ……………
        ……………
        {elements of n-1 row}
};
```

For example, the statement

```
int arr[3][3] = {
        {1, 2, 3},
        {6, 7, 8},
        (5, 9, 3}
};
```

defines two dimensional array of order 3 X 3 and initializes all its elements.

c. Write the syntax for accessing structure members in C++. Also construct a structure called *"Student"* whose members are roll_no, name, branch and marks. Use this structure in your program that will read student information and then display that information? (8)

Answer:
C++ provides the period or dot(.) operator to access the members of a structure . The dot operator connects a structure variable and its member. The syntax for accessing members of a structure variable is as follows:

structvar.membername

Here, structvar is a structure variable and membername is one of the member of structure. Thus, the dot operator must have a structure variable on its left and a member name on its right.

```
#include <iostream.h>

struct Student {
        int roll_no;
        char name[25];
        char branch[10];
        int marks;
};

void main() {
        Student s1;
        cout << "Enter data for student" << endl;
        cout << "Roll Number" ;
        cin >> s1.roll_no;
        cout << "Name" ;
        cin >> s1.name;
        cout << "Branch" ;
```

```
        cin >> s1.branch;
        cout << “Marks Obtained” ;
        cin >> s1marks;

        cout << “ Student Report” << endl;
        cout << “Roll Number :” << s1.roll_no << endl;
        cout << “Name :”  << s1.name << endl;
        cout << “Branch :”  << s1.branch << endl;
        cout << “Marks Obtained :”  << s1.marks << endl;
}
```

Q.4 a. Define *Inline function*? What are the guidelines that need to be followed for deciding whether to the member function inline or not?. (6)

Answer:

Inline function: *If a member function is define as well declared, in the definition of the class itself, the member function is said to be defined inline.*

Following are the certain guidelines need to be followed while declaring a member function as inline function:

(i) Defining inline functions can be considered once a fully developed and tested program too slowly and shows bottlenecks in certain functions.

(ii) Inline functions can be used when member functions consist of one very simple statement such as the return statement. For example,

```
inline int date :: getday() {
     return day;
}
```

(iii) If a function is too large to be expanded, it will not be treated be treated as inline. Thus, declaring a function will not guarantee that the compiler will consider it as an inline function.

(iv) Functions consisting of loops will not be considered as inline functions.

b. What are the conditions that must be satisfied for function calling? (4)

Answer:

The following conditions must be satisfied for a function call:

- The number of arguments in the function call and the function declaratory must be same.
- The data type of each of the arguments in the function call should be the same as the corresponding parameter in the function declaratory statement. However, the names of the arguments in the function call and the parameters in the function definition can be different.

c. What is function overloading? Write overloading functions for swapping two characters, two integers and two float parameters. (2+4)

Answer:

Function overloading *is a concept that allows multiple functions to share the same name with different argument types. Function overloading implies that the function definition can have multiple forms. Assigning one or more function body to the same name is known as function overloading or function name overloading.*

```
#include <iostream.h>
void swap(char &x, char &y) {
        char t;
```

```
        t = x;
        x = y;
        y = t;
}

void swap(int &x, int &y) {
        int t;
        t = x;
        x = y;
        y = t;
}

void swap(float &x, float &y) {
        float t;
        t = x;
        x = y;
        y = t;
}

void main() {
        char ch1, ch2,
        cout << "Enter two characters :";
        cin >> ch1 >> ch2;
        swap(ch1, ch2);
        cout << "After Swapping characters :" << ch1 << " " <<ch2 << endl;

        int in1, in2,
        cout << "Enter two integers :";
        cin >> in1 >> in2;
        swap(in1, in2);
        cout << "After Swapping integers :" << in1 << " " <<in2 << endl;

        float fl1, fl2,
        cout << "Enter two floats :";
        cin >> fl1 >> fl2;
        swap(fl1, fl2);
        cout << "After Swapping floats :" << fl1 << " " <<fl2 << endl;
}
```

Q.5 a. What is the use of constructor in C++? List any four properties of constructor? (2+4)

Answer:

A constructor is a special member function whose main use is to allocate the required resources such as memory and initialize the objects of its class. It is generally used to initialize the object member parameters and allocate the necessary resources to the object members.

Properties

1. It has same name as that of the class to which it belongs.
2. It is executed automatically whenever the class is instantiated.
3. It does not have any return type.
4. It can't be invoked explicitly.

5. It can access any data member like other member functions.
6. Constructor must be declared in public section of the class.

b. Write a complete C++ program to do the following :

(i) 'Student' is a base class, having two data members: entryno and name; entryno is integer and name of 20 characters long. The value of entryno is 1 for Science student and 2 for Arts student, otherwise it is an error.

(ii) 'Science' and 'Arts' are two derived classes, having respectively data items marks for Physics, Chemistry, Mathematics and marks for English, History, Economics.

(iii) Read appropriate data from the screen for 3 science and 2 arts students.

(iv) Display entryno, name, marks for science students first and then for arts students. (10)

Answer:

```
#include<iostream.h>
class student {
  protected:
    int entryno;
    char name[20];
  public:
    void getdata(){
      cout<<"enter name of the student"<<endl;
      cin>>name;
    }
    void display(){
      cout<<"Name of the student is"<<name<<endl;
    }
};

class science:public student {
  int pcm[3];
  public:
    void getdata(){
      student::getdata();
      cout<<"Enter marks for Physics,Chemistry and Mathematics"<<endl;
      for(int j=0;j<3;j++){
        cin>>pcm[j];
      }
    }

    void display(){
      entryno=1;
      cout<<"entry no for Science student is"<<entryno<<endl;
      student::display();
      cout<<"Marks in Physics,Chemistry and Mathematics are"<<endl;
      for(int j=0;j<3;j++){
```

```
        cout<<pcm[j]<<endl;;
      }
    }
};

class arts:public student {
  int ehe[3];
  public:
    void getdata(){
      student::getdata();
      cout<<"Enter marks for English,History and Economics"<<endl;
      for(int j=0;j<3;j++){
        cin>>ehe[j];
      }
    }

    void display(){
      entryno=2;
      cout<<"entry no for Arts student is"<<entryno<<endl;;
      student::display();
      cout<<"Marks in English,History and Economics are"<<endl;
      for(int j=0;j<3;j++){
        cout<<ehe[j]<<endl;;
      }
    }
};

void main(){
  science s1[3];
  arts a1[3];
  int i,j,k,l;
  cout<<"Entry for Science students"<<endl;
  for(i=0;i<3;i++){
    s1[i].getdata();
  }
  cout<<"Details of three Science students are"<<endl;
  for(j=0;j<3;j++){
    s1[j].display();
  }
  cout<<"Entry for Arts students"<<endl;
  for(k=0;k<3;k++){
    a1[k].getdata();
  }
  cout<<"Details of three Arts students are"<<endl;
  for(l=0;l<3;l++){
    a1[l].display();
  }
}
```

Q.6 a. Give the syntax for overloading a unary and binary operator. Is it possible to overload the ternary (? :) operator? Support your answer with proper reason. (2+2+3)

Answer:

The **syntax for overloading a unary operator** is as follows:

```
returntype operator OperatorSymbol () {

        // body of Operator function

}
```

The keyword *operator* facilitates overloading of the C++ operators. The keyword *operator* indicates that the *OperatorSymbol* following it, is the C++ operator to be overloaded to operate on members of its class. The following examples illustrate the overloading of unary operaters:

```
int operator +();
void operator –();
```

The **syntax for overloading a binary operator** is as follows:

```
returntype operator OperatorSymbol (arg) {

        // body of Operator function

}
```

The keyword *operator* facilitates overloading of the C++ operators. The keyword *operator* indicates that the *OperatorSymbol* following it, is the C++ operator to be overloaded to operate on members of its class. The operator overloaded in a class is know as overloaded operator function.

For examples,

```
complex operator + ( complex c1);

int operator – ( int a);
```

No, it is not possible to overload the ternary (? :) operator.
The ternary (? :) operator has an inherent meaning and it requires three arguments. C++ does not support the overloading of an operator, which operates on three operands. Hence, the conditional operator, which is the only ternary operator in C++ language, cannot be overloaded.

b. Write a program to illustrate the overloading of new and delete operators. (5)

Answer:

```
#include <iostream.h>
const int ARRAY_SIZE = 10;

class vector {
        private:
                int *array;
        public :
                // overloading of new operator

                void * operator new(size_t size) {
```

```
                    vector *my_vector;
                    my_vector = ::new vector;

                    my_vector ->array = new int[ARRAY_SIZE];
                    return my_vector;
          }

          // overloading of delete operator

          void operator delete(void* vec) {
                    vector *my_vect;
                    my_vect = (vector *) vec;
                    delete (int *) my_vect -> array;
                    ::delete vec;
          }

          void read();
          int sum();
}

void vector ::read() {
          for (int i=0; i<ARRAY_SIZE; i++) {
                    cout << "Vector[" << i << "] = ";
                    cin >> array[i];
          }
}

int vector:: sum() {
          int sum = 0;
          for (int i=0; i<ARRAY_SIZE; i++)
                    sum = sum + array[i];
          return sum;
}

void main() {
          vector *my_vector = new vector;
          cout << "Enter Vector Data ........." << endl;
          my_vector -> read();
          cout << "Sum of Vector = " << my_vector ->sum();
          delete my_vector;
}
```

c. What are the restrictions for overloading operators? (4)

Answer:

Q6 (c) - There are some restrictions for overloading ~~operators~~ operators in C++, such as the following:

1. One cannot extend the language by inventing new operator, e.g. one cannot create own "exponentiation" operator using the character **. One must limit oneself to existing language provided operators.
2. An operator's arity cannot be changed, e.g. negation (~) operator cannot be used as a binary operator. a = ~b is ok, however, a = b~c is not correct.
3. Operator precedence cannot be changed.
4. Operator's associativity cannot be changed. Addition and subtraction operators are both left-associative (expression is evaluated left-to-right) and one cannot change the rule.
5. One cannot change operators for built-in data types like int, char, float. At least one operand of overloaded operators must be user-defined type.
6. Following operators cannot be overloaded:

a.) Class member operator .
b.) Pointer to member operator .*
c.) Scope resolution operator ::
d.) Conditional expression operator ?:

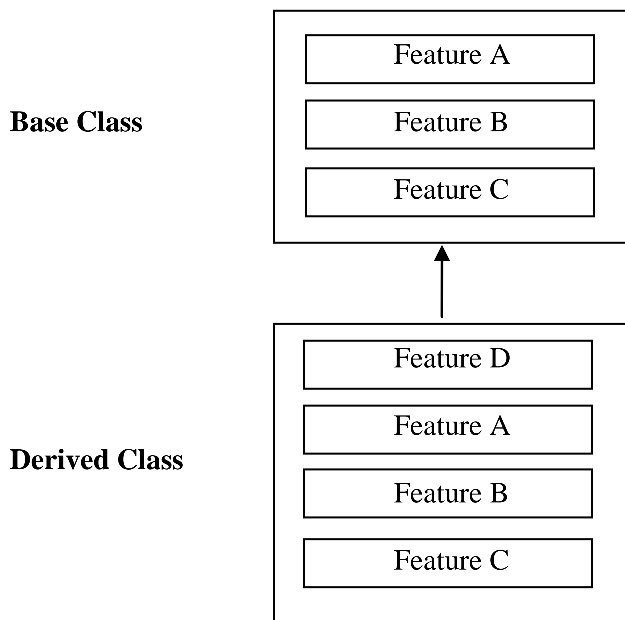
Signature 28.10.15

Q.7 a. What is Inheritance? What are the rules must be kept in mind while deciding whether to define members as private, protected, or public? (5)

Answer:

The technique that allows the extension and reuse of exiting code without having to rewrite the code from scratch is known as Inheritance. Inheritance involves the creation of new classes (called derived classes) from the existing ones (called base classes), thus enabling the creation of a hierarchy of classes that simulates the class and subclass of the real world.

Inheritance is a technique of organizing information in a hierarchical form. It allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. Classes are created by first inheriting all the variables and behavior defined by some primitive class and then adding specialized variables and behaviors.



Thus, inheritance is a prime feature of OOPs used as a process of creating new classes (called derived classes, from the existing classes (called base classes). The derived class inherits all the capabilities of the base class and can add refinements of its own. The base class remains unchanged. The derivation of new class from the existing class is shown in the above figure. The derived class inherits all features (A, B and C) of the base class and adds its own feature D. The arrow in the figure symbolizes derived from. Its direction from the derived class towards the base class represents that the derived class accesses features of the base class and not vice versa.

The following rules are to be kept in mind while deciding whether to define members as private, protected, or public are as follows:

- A private member is accessible only to members of the class in which the private member is declared. They cannot be inherited.
- A private member of the base class can be accessed in the derived class through the member functions of the base class.
- A protected member is accessible to members of its own class and to any of the members in a derived class.
- If a class is expected to be used as a base class in future, then members which might be needed in the derived class should be declared protected rather private.

- A public member is accessible to members of its own class, members of the derived class, and outside users of the class.
- The private, protected, and public sections may appear as many times as needed in a class and in any order. In case an inline member function refers to another member (data or function), that member must be declared before the inline member function is defined. Nevertheless, it is normal practice to place the private section first, followed by the protected section and finally the public section.
- The visibility mode in the derivation of a new class can be either public or private.
- Constructors of the base class and the derived class are automatically invoked when the derived class is instantiated. If a base class has constructors with arguments, ten their invocation must be explicitly specified in the derived class's initialization section. However, no-argument constructor need not be invoked explicitly constructors must be defined in the public section of a class (base and derived) otherwise, the compiler generates the error message: *unable to access constructor.*

b. What would be the output of the following code: (4)

```
#include <iostream.h>

class BC {

    public:

        BC(int a){
            cout<<"\nOne-argument constructor in base class\n";
        }
    };

class DC : public BC {

    public:
        DC(int d) : BC(d){
            cout<<"\nOne-argument constructor exists in derived Class\n";
    }

};

void main(){
    DC objD(3);
}
```

Answer:

The expected out is:

One-argument constructor in base class

One-argument constructor exists in derived Class

c. Explain the term Polymorphism? What are the different forms of polymorphism? What are the rules that need to be kept in mind while deciding virtual functions? (2+2+3)

Answer:

The technique to allow a single name / operator to be associated with different operations depending on the type of data passed to it is known as Polymorphism. In C++, it is achieved through function overloading, operator overloading and dynamic binding (virtual functions).

Polymorphism is a very powerful concept that allows the design of flexible applications. The word Polymorphism is derived from two Greek words, Poly means many and morphos means forms. So, Polymorphism means ability to take many forms.
Polymorphism can be defined as one interface multiple methods which means that one interface can be used to perform different but related activities.

The different form of Polymorphism is

- Compile time (or static) polymorphism.
- Runtime (or Dynamic) polymorphism.

Compile Time Polymorphism
In compile time polymorphism, static binding is performed. In static binding, the compiler makes decision regarding selection of appropriate function to be called in response to function call at compile time. This is because all the address information requires to call a function is known at compile time. It is also known as early binding as decision of binding is made by the compiler at the earliest possible moment. The compile time polymorphism is implemented in C++ using function overloading and operator overloading. In both cases, the compiler has all the information about the data type and number of arguments needed, so it can select the appropriate function at compile time.
The advantage of static binding is its efficiency, as it often requires less memory and function calls are faster. Its disadvantage is the lack of flexibility.
Runtime Polymorphism
In runtime polymorphism, dynamic binding is performed. In dynamic binding, the decision regarding the selection of appropriate function to be called is made by the compiler at run time and not at compile time. This is because the information pertaining to the selection of the appropriate function definition corresponding to a function a call is only known at run time. It is also known as late binding as the compiler delays the binding decision until run time. In C++, the runtime polymorphism is implemented using virtual functions.
The advantage of dynamic binding is that it allows greater flexibility by enabling user to create class libraries that can be reused and extended as per requirements. It also provides a common interface in the base class for performing multiple tasks whose implementation is present in the derived classes. The main disadvantage of dynamic binding is that there is little loss of execution speed, as compiler will have to perform certain overheads at run time.

When virtual functions are used for implementing run time polymorphism, there are certain rules to be followed:

- When a virtual function in a base class is created, there must be a definition of the virtual function in the base class even base class version of the function is never actually called.
- They cannot be static members
- They can be a friend function to another class
- They are accessed using object pointers.
- A base pointer can server as a pointer to a derived object since it is type-compatible whereas a derived object pointer variable cannot serve as a pointer to base objects.
- Its prototype in a base class and derived class must be identical for the virtual function to work properly.
- The class cannot have virtual constructors, but can have virtual destructor.

- To realize the potential benefits of virtual functions supporting runtime polymorphism, they should be declared in the public section of a class.

Q.8 a. What is Class Template? Explain the syntax of a class template with suitable examples. (8)

Answer:

It is possible to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
   T values [2];
  public:
   mypair (T first, T second)
   {
     values[0]=first;
     values[1]=second;
   }
};
```

This class defination serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

This same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the above class template has been defined inline within the class declaration itself.

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
     T a, b;
  public:
      mypair (T first, T second) {
          a=first;
          b=second;
     }
     T getmax ();
};

template <class T>
T mypair<T>::getmax (){
     T retval;
```

```
    retval = a>b? a : b;
    return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

b. Create a class number to store an integer number and the member function read() to read a number from console and the member function div() to perform division operations. It raises exception if an attempt is made to perform *divide-by-zero* operation. It has an empty class name DIVIDE used as the throw's expression-id.
Write a C++ program to use these classes to illustrate the mechanism for detecting errors, raising exceptions, and handling such exceptions. (8)

Answer:

```
#include <iostream.h>
class number {
        private :
                int num;
        public :
                void read() {           // read number from keyboard
                        cin >> num;
                }
        class DIVIDE {};                // abstract class used in exceptions

        int div( number num2 ) {
                if (num2.num == 0)              // check for zero division if yes raise exception
                        throw DIVIDE();
                else
                        return num / num2.num;     // compute and return the result
        }
};

int main() {
        number num1, num2;
        int result;
        cout << "Enter First Number : ";
        num1.read;
        cout << "Enter Second Number: ";
        num2.read();

        try {
                cout << "Trying division operation";
                result = num1.div(num2);
```

```
        cout << result << endl;
    } catch (number::DIVIDE)  {                    // exception handler block
        cout << "Exception : Divide-By-Zero";
        return 1;
    }
    cout << "No Exception generated:"
    return 0;
}
```

Q.9 a. Explain the following: (3×3)
(i) ifstream
(ii) ofstream
(iii) fstream

Answer:

(i) ifstream

The header file ifstream.h is a derived class from the base class of istream and is used to read a stream of objects from a file.

For example, the following program segment shows how a file is opened to read a class of stream objects from a specified file.

```
#include <fstream.h>
void main() {
    ifstream infile;
    infile.open("file_name");

    ……………
    ……………
}
```

(ii) ofstream

The header file ofstream.h is derived from the base class of ostream and is used to write a stream of objects in a file.

For example, the following program segment shows how a file is opened to write a class of stream objects on a specified file.

```
#include <fstream.h>
void main() {
    ofstream outfile;
    outfile.open("file_name");

    ……………
    ……………
}
```

(iii) fstream

The header file fstream.h is a derived class from the base class of iostream and is used for both reading and writing a stream of objects on a file. The statement #include<fstream.h> automatically includes the header file iostream.h

For example, the following program segment shows how a file is opened for both reading and writing a class of stream objects from a specified file.

```
#include <fstream.h>
void main() {
    fstream infile;
    infile.open("file_name" , ios::in || ios::out);

    ……………
    ……………
}
```

When a file is opened for both reading and writing, the I/O streams keep track of two file pointers, one for input operation and other for output operation.

b. Write a program to open a file whose name is passed as command line argument. (7)

Answer:

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

void main(int argc,char *argv[]){
        if(argc < 2){
           cerr<<"Illegal Usage Correct Usage: size <file-name>";
           exit(1);
        }
        ifstream in(argv[1],ios::in|ios::binary);

        if(!in){
                cerr<<"Error opening the input file";
                exit(1);
        }
        long int size=0;
        char ch;
        while(!in.eof()){
                in>>ch;
                size++;
        }
        cout<<"The size of file "<<argv[1]<<" is "<<size<<" bytes.";

}
```

TEXT BOOK

I. C++ and Object-Oriented Programming Paradigm, Debasish Jana, Second Edition, PHI, 2005 (TB-I:)

© IETE