Q.2a. Discuss the fundamental features of the object oriented programming.

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.What are the rules and significance of declaring identifiers? Give some examples of valid and invalid identifiers.

Answer:
The rules of C++ for valid identifiers state that:
An identifier must:

- start with a letter
- consist only of letters, the digits 0 to 9, or the underscore symbol _
- not be a **reserved word**

Identifiers should be chosen to reflect the significance of the variable in the program being written. Although it may be easier to type a program consisting of single character identifiers, modifying or correcting the program becomes more and more difficult. The minor typing effort of using *meaningful* identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

The following are **valid** identifiers

    Length
    days_in_year
    DataSet1
    Profit95
    Int
    _Pressure
    first_one first_1

although using _Pressure is not recommended.

The following are **invalid**:

    days-in-year
    1data
    int
    first.val throw

c.With the help of an example, describe 'size of ' operator.

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() {
        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
```

Q.3 a. Define array. Give the syntax for defining an array. With the help of syntax and example, explain how single-dimensional array can be initialized at definition time.

Answer:
An array is a group of logically related data items of the same data-type addresses by a common name, and all the items are stored in contiguous memory locations. For example, the statement

```
int marks[10];
```

defines an array by the name marks that can hold a maximum of ten elements. The individual elements of an array are accessed and manipulated using the array name followed by their index.

Syntax
Like any other variables, the array variable must be defined before its use. The syntax for defining an array is as follows

```
datatype arrayname[array_size], ....;
```

In the definition, the array name must be a valid C++ identifier, followed by an integer value enclosed in square braces. The array_size indicates the maximum number of elements the array can hold. Some of the examples are:

```
int marks[10];
char name[50];
int a[10], b[20], c[15];                 // defines three arrays
```

Initialization at Definition
Arrays can be initialized at the time of their definition as follows:

```
datatype arrayname[size]  = { list of values separated by comma};
```

For example, the statement

```
int age[5] = { 22, 30, 18, 16, 35};
```

defines an array of integers of size 5. In this case, the first element of the array age in initialized with 22, second with 30, and so on. A semicolon always follows the closing brace. The array size may be omitted when the array is initialized during array definition as follows:

```
int age[ ] = { 22, 30, 18, 16, 35};
```

in such cases, the compiler assumes the array size to be equal to the number of elements enclosed within the curly braces. Hence, in the above declaration, the size of the array is considered as five.

b. 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.

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. Write the conditions that must be satisfied for function calling.

Answer

The following conditions must be satisfied for a function call:

- The number of arguments in the unction 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.

b. Write a class called “Student” with data members (*char* name, *int* rollnumber, *int* marks). Write appropriate inline member functions to enter and access the student data. Write a member function to calculate the average marks for a student and print it on the console.

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.5 a. What is the use of constructor in C++? List any four properties of constructor.

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. Why is destructor function required in a class?

Answer
Virtual destructor is used in the following situations:

- A virtual destructor is used when one class needs to delete object of a derived class that are addressed by the base-pointer and invoke a base class destructor to release resources allocated to it.
- Destructors of a base class should be declared as virtual functions. When a delete operation is performed on an object by a pointer or reference, the program will first call the object destructor instead of the destructor associated with the pointer or reference type.

c.Explain the syntax for overloading a unary and binary operator using appropriate examples.
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.

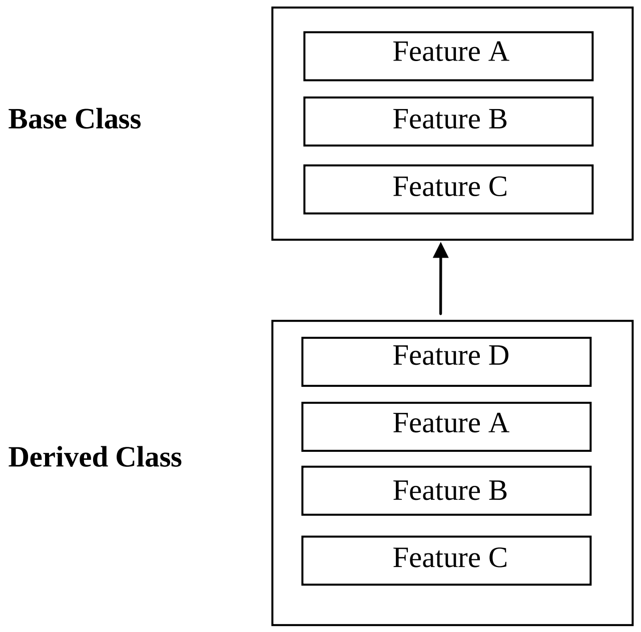
For examples,

```
complex operator + ( complex c1);

int operator – ( int a);
```

Q.6 a. What is Inheritance? What are the rules that must be kept in mind while deciding whether to define members as private, protected, or public?

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. List some of the benefits of Inheritance using appropriate examples/code.

Answer:
Benefits of inheritance are as follows:

- When inherited from another class, the code that provides a behavior required in the derived class need not have to be rewritten. Benefits of reusable code include increased reliability and a decreased maintenance cost of sharing of the code by all its users.
- Code sharing can occur at several levels. For example, at higher level, many users or projects can use the same class. These are referred to as software components. At lower level, code can be shared by two or more classes within a project.
- When multiple classes inherit from the same superclass, it guarantees that the behavior they inherit will be the same in all cases.
- Inheritance permits the construction of reusable software components. Already, several such libraries are commercially available and many more are expected.
- When a software system can be constructed largely out of reusable components, development time can concentrated on understanding the portion of a new system. Thus, software systems can be generated more quickly and easily by rapid prototyping.

c.What would be the output of the following code?

```
#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
```

Q.7 a. Explain the term Polymorphism. In what situation Virtual destructors are used?

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.

Virtual destructor is used in the following situations:

- A virtual destructor is used when one class needs to delete object of a derived class that are addressed by the base-pointer and invoke a base class destructor to release resources allocated to it.
- Destructors of a base class should be declared as virtual functions. When a delete operation is performed on an object by a pointer or reference, the program will first call the object destructor instead of the destructor associated with the pointer or reference type.

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.

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.8 a. Explain template. Write a program using function template to find the cube of a given integer, float and a double number.

Answer:
A template is one of the features which enable us to define generic classes and functions and thus provides support for generic programming. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.
A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int, array and float array.
A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

```
        //Using a function template

        #include <iostream.h>

        template < class T >

        T cube( T value1) {
                return value1*value1*value1;

        }

        int main() {
                 int int1;

                 cout <<"Input integer value: ";
                 cin  >> int1;
                 cout << "The cube of integer value is: "<< cube( int1);

                 double double1;
                cout << "\nInput double value; ";
```

```
                 cin  >> double1;
                 cout << "The cube of double value is: "<< cube( double1);

                float float1;
                cout << "\nInput float value";
                cin >> float1;
                 cout << "The cube of float value is: "<< cube(float1);

                cout<< endl;
                return 0;
        }
```

b.What are the rules adopted by compiler for selecting a suitable template when the program has overloaded function templates?

Answer:
The compiler adopts the following rules for selecting a suitable template when the program has overloaded function templates:

- Look for an exact match on functions; if found, call it.
- Look for a function template from which a function that can be called with an exact match be generated; if found, call it.
- Try ordinary overloading resolution for the functions; if found, call it.

If no match is found in all the three alternatives, then that call is treated as an error. In each case if there is more than one alternative in the first step that finds a match, the call is ambiguous and is an error.

c.What is Class Template? Give the syntax for declaring class template.

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.

Q.9 a. Explain the following giving syntax /examples: (4×2)
(i) put() and get() functions
(ii) getline() and write() functions

Answer

(i) put() and get() functions

The stream classes of C++ supports two member functions, get() and put(). The function get() is a member function of the input stream class istream and is used to read a single character from the input device. The function put() is a member function of the output stream class ostream and is used to write a single character to the output device. The function get() has two versions with the following prototypes:

```
void get(char &);
void get(void);
```

Both the functions can fetch a white-space character including the blank space, tab, and newline character. It is well known that, the member functions are invoked by their objects using dot operators. Hence, these two functions can be used to perform input operation either by using the predefined object, cin or an user defined object of the istream class. The following program illustrates the use of get() function to read a line (until a carriage return key is pressed).

```
#include <iostream.h>
void main {
        char c;
        cin.get( c );
        while( c != '\n' ) {
                cout << c;
                cin.get( c );
        }
}
```

The function put(), which is a member of the output stream class ostream prints a character representation of the input parameter. For example, the statement

```
cout.put( 'R' );
```

prints the character R and the statement

```
cout.put( c );
```

prints the contents of the character variable c.

(ii) getline() and write() functions

The C++ stream classes support line-oriented functions, getline() and write() to perform input and output operations. The getline() function reads a whole line of text that ends with the new line or until the maximum limit is reached.

The istream::getline member function has the following versions:

```
istream& getline(signed char*, int len, char = '\n');
istream& getline(unsigned char*, int len, char = '\n');
```

The prototype of write() function is:

```
ostream::write( char * buffer, int size);
```

It displays size number of characters from the input buffer. The display does not even stop when the NULL character is encountered. If the length of the buffer is less than the indicated size, it displays beyond the bounds of buffer. Therefore it is responsibility of the user to make sure that the size does not exceed the length of the string.

b. Write a C++ program to display the contents of a file on the console, where filename is entered interactively.

Answer:

```
#include <fstream.h>
#include <iomanip.h>

int main() {
        char ch;
        char filename[25];
        cout << "Enter Name of the File:";
        cin >> filename;

        // create a file object in read mode

        ifstream ifile( filename);

        if ( !ifile) {                  // file open status
                cerr << "Error opening " << filename << endl;
                return 1;
        }

        ifile >> resetiosflags (ios::skipws);

        while( ifile ) {
                ifile >> ch;
                cout << ch;
        }
        return 0;
}
```

TEXTBOOK

1. Object-oriented Programmeming with C++, Poornachandra Sarang, PHI, 2004