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

Answer:

The fundamentals features of the OOPs are the following:

(i) **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*.

(ii) **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).*

(iii) **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.

(iv) **Multiple Inheritance:** The mechanism by which a class is derived from than one base class is known as multiple inheritance.

(v) **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).

(vi) **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.

(vii) **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.

(viii) **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 is the advantage of a sizeof( ) operator?

Answer: **Page Number 16 of Text Book**

c. Explain the difference between:
(i) ‘A’ and “A”
(ii) a = b and a == b
(iii) a & b and a & & b

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.

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 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;
}
```

c. Write some situations where the usage of pointers is required.

Answer:

The usage of pointer is essential in the following situations:

- Accessing array elements.
- Passing arguments to functions by address when modification of formal arguments is to be reflected on actual arguments.
- Passing arrays and strings to functions
- Creating data structures such as linked lists, trees, graphs, etc.
- Obtaining memory from the system dynamically.
-

Q.4 a. Define *Inline function*. What are the guidelines that need to be followed for deciding if the function is to be used as a member function or inline function?

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?

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.

c. What is function overloading? Write overloading functions for swapping two character, two integer and two float parameters.

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. Design a class to represent "account" information of an individual that includes following members:-

Data Members

- Name of account holder -------- String
- Account number ---------- int
- Type of Account --------- char

- Balance Amount ---------- float

Member Functions

- To assign initial values (using constructor)
- To display the name of account holder, account number, account type and balance amount in the account.
- To deposit an amount in the account.
- To withdraw an amount.

Use the above class to write an interactive program.

Answer:

```
# include<iostream.h>
# include<conio.h>
# include<iomanip.h>
class bank {
        char name[20];
         int acno;
        char actype[4];
        float balance;
   public:
        void init();
        void deposit();
        void withdraw();
         void disp_det();
 };
//member functions of bank class
void bank :: init(){
        cout<<" New Account<BR>;
        cout<<"Enter the Name of the depositor : ";
        cin.get(name,19,");
        cout<<"Enter the Account Number : ";
        cin>>acno;
        cout<<"Enter the Account Type : (CURR/SAVG/FD/RD/DMAT) ";
        cin>>actype;
        cout<<"Enter the Amount to Deposit : ";
        cin >>balance;
}
void bank :: deposit(){
        float more;
        cout <<"Depositing<BR>;
        cout<<"Enter the amount to deposit : ";
        cin>>more;
        balance+=more;
}
void bank :: withdraw(){
        float amt;
        cout<<" Withdrwal<BR>;
        cout<<"Enter the amount to withdraw : ";
```

```
        cin>>amt;
        balance-=amt;
}
void bank :: disp_det(){
        cout<<" Account Details <BR>;
        cout<<"Name of the depositor : "<<name<<endl;
        cout<<"Account Number       : "<<acno<<endl;
        cout<<"Account Type         : "<<actype<<endl;
        cout<<"Balance              : $"<<balance<<endl;
}
// main function , exectution starts here
void main(){
        clrscr();
        bank obj;
        int choice  =1;
        while (choice != 0 ){
                cout<<"Enter 0 to exit
                  1. Initialize a new acc.
                  2. Deposit
                  3.Withdraw
                  4.See A/c Status";
                cin>>choice;
                switch(choice){
                        case 0 :obj.disp_det();
                                cout<<"EXITING PROGRAM.";
                                break;
                         case 1 : obj.init();
                                break;
                         case 2: obj.deposit();
                                break;
                        case 3 : obj.withdraw();
                                break;
                        case 4: obj.disp_det();
                                break;
                        default: cout<<"Illegal Option"<<endl;
                }
        }
        getch();
}
```

b. Why is destructor function required in a class? What are the special rules that should be considered while defining a destructor function for a class?

Answer:

Destructor: A destructor is used to destroy the objects that have been created by a constructor. It has the same name as that of the class but is preceded by a tilde. For example,

~class_name () {}

The following rules need to be considered while defining a destructor for a given class:

- The destructor function has the same name as the class but prefixed by a tilde (~). The tilde distinguishes it from a constructor of the same class.
- Unlike the constrictor, the destructor does not take any arguments. This is because there is only one way to destroy an object.
- The destructor has neither arguments, nor a return value.
- The destructor has no return type like constructor, since it is invoked automatically whenever an object goes out of scope.
- There can be only one destructor in each class.

Q.6 a. Write the steps that involves the process of operator overloading.

Answer:

The process of operator overloading generally involves the following steps:

1. Declare a class (that defines the data type) whose objects are to be manipulated using operators.
2. Declare the operator function, in the *public* part of the class. It can be either a normal member function or a friend function.
3. Define the operator function either within the body of a class or outside the body of the class (however, the function prototype must exist inside the class body).

b. Give the syntax for overloading a binary operator. Write a program to overload the binary operator + in order to perform addition of complex numbers.

Answer:

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);
```

The following program illustrates the overloading of the binary operator + in order to perform addition of complex numbers.

```
#include (iostream.h>
class complex {
        private :
                float real;
                float imag;
        public :
                complex() {
                        real = imag = 0.0;
                }

        void getdata() {
                cout << "Enter Real Part :";
                cin >> real;
                cout << "Enter Imaginary Part :";
                cin >> imag;
        }

        complex operator + ( complex c) ;

        void outdata (char *msg) {
                cout << endl << msg;
                cout << "(" << real;
                cout << ", " << imag << ")";
        }
}

complex complex :: operator + ( complex c) {
        complex temp;
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        retrun (temp);
}

void main () {

        complex c1, c2, c3;
        cout << " Enter Complex Number c1 -----" << endl;
        c1.getdata();
        cout << " Enter Complex Number c2 -----" << endl;
        c2.getdata();

        c3 = c1 + c2;
```

```
    c3.outdata(" c3 = c1 + c2: ");
}
```

c. Is it possible to overload the ternary (? :) operator? Support your answer with proper reason.

Answer:

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.

Q.7 a. Explain the term Polymorphism. What are the different forms of polymorphism?

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.

b. Explain the difference between inheriting a class with public and private visibility mode.

Answer:

The syntax of declaring a derived class from base class is as follows:

```
class DerivedClass : [VisibilityMode} BaseClass {
        // members of derived class
        // and they can access members of the base class
};
```

The derivation of *Derivedclass* from the *BaseClass* is indicated by colon (:). The *VisibilityMode* enclosed within the square brackets implies that is optional. The default visibility mode is **private.** If the visibility mode is specified, it must be either *public* or *private.*

Inheritance of a base class with visibility mode *public*, by a derived class, causes *public* members of the base class to become *public* members of the derived class and the *protected* members of the base class become *protected* members of the derived class. Member functions and objects of the derived class can treat these derived members as though they are defined in the derived class itself. It is known that the public members of

a class can be accessed by the objects of the class. Hence, the objects of a derived class can access public members of the base class that are inherited as public using dot operator. However, *protected* members cannot be accessed with dot operator.

Inheritance of a base class with visibility mode *private*, by a derived class, causes *public* members of the base class to become *private* members of the derived class and the *protected* members of the base class become *private* members of the derived class. Member functions and objects of the derived class can treat these derived members as though they are defined in the derived class with the *private* modifier. Thus, objects of a derived class cannot access these members.

The private members of the base class remain private to the derived class, whether the base class in inherited publicly or privately, they add to the data items of the derived class but are not accessible to the member of a derived class. Derived classes can access them through the inherited member functions of the base class.

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

Answer:

```
//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. 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.9 a. 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;
}
```

b. Explain the following:

(i) ifstream
(ii) ofstream
(iii) fstream

Answer:

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.

Text Book

C++ and Object-Oriented Programming Paradigm, Debasish Jana, Second Edition, PHI, 2005