BCS-031 Solved assignment july 2017-january 2018 session

Q.1. What is Object Oriented Programming (OOP) approach? Explain how OOP is better than structured programming

A.1.(A)
Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.

The concepts and rules used in object-oriented programming provide these important benefits:

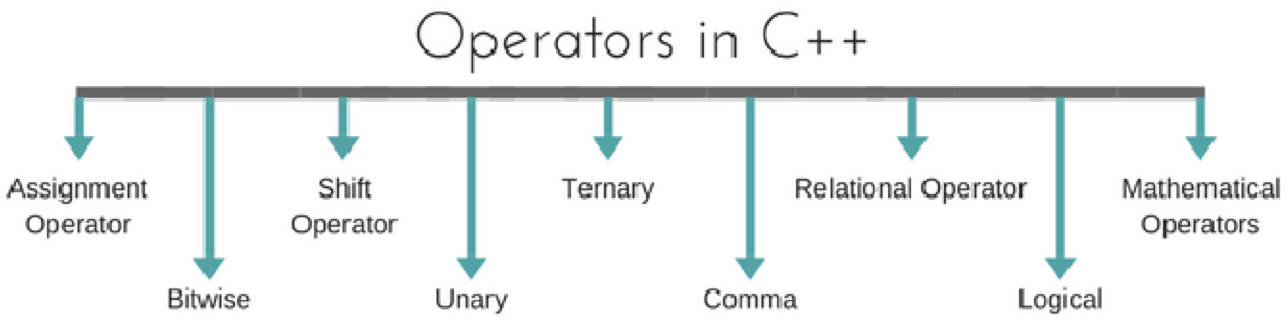
The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Called inheritance, this property of OOP forces a more thorough data analysis, reduces development time, and ensures more accurate coding.
Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of data hiding provides greater system security and avoids unintended data corruption.
The definition of a class is reuseable not only by the program for which it is initially created but also by other object-oriented programs (and, for this reason, can be more easily distributed for use in networks).
The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself.

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Pub Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely	In OOP, data can not move easily from function to function,it can be kept publi

	from function to function in the system.	private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the f of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Q.1.(b) Explain use of different operators of C++ , with the help of examples.

A.(b)
Operators in C++
Operators are special type of functions, that takes one or more arguments and produces a new value. For example : addition (+), substraction (-), multiplication (*) etc, are all operators. Operators are used to perform various operations on variables and constants.



Types of operators
Assignment Operator
Mathematical Operators
Relational Operators
Logical Operators
Bitwise Operators
Shift Operators
Unary Operators

Ternary Operator
Comma Operator

Assignment Operator (=)

Operates '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

Mathematical Operators

There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , diversion (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.
C++ and C also use a shorthand notation to perform an operation and assignment at same type. *Example*,

```
int x=10;
x += 4 // will add 4 to 10, and hence assign 14 to X.
x -= 5 // will subtract 5 from 10 and assign 5 to x.
```

Relational Operators

These operators establish a relationship between operands. The relational operators are : less than (<) , grater thatn (>) , less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=).
You must notice that assignment operator is (=) and there is a relational operator, for equivalent (==). These two are different from each other, the assignment operator assigns the value to any variable, whereas equivalent operator is used to compare values, like in if-else conditions, *Example*

```
int x = 10;  //assignment operator
x=5;       // again assignment operator
if(x == 5)   // here we have used equivalent relational operator, for comparison
{
 cout <<"Successfully compared";
}
```

Logical Operators

The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.
If two statement are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially `while` loop) and in Decision making.

Bitwise Operators

There are used to change individual bits into a number. They work with only integral data types like `char`, `int` and `long` and not with floating point values.
Bitwise AND operators `&`
Bitwise OR operator `|`
And bitwise XOR operator `^`
And, bitwise NOT operator `~`

They can be used as shorthand notation too, `& =` , `|=` , `^=` , `~=` etc.

Shift Operators
Shift Operators are used to shift Bits of any variable. It is of three types,
Left Shift Operator `<<`
Right Shift Operator `>>`
Unsigned Right Shift Operator `>>>`

Unary Operators
These are the operators which work on only one operand. There are many unary operators, but increment `++` and decrement `--` operators are most used.
Other Unary Operators : address of `&`, dereference `*`, **new** and **delete**, bitwise not `~`, logical not `!`, unary minus `-` and unary plus `+`.

Ternary Operator
The ternary if-else `? :` is an operator which has three operands.

```
int a = 10;
a > 5 ? cout << "true" : cout << "false"
```

Comma Operator
This is used to separate variable names and to separate expressions. In case of expressions, the value of last expression is produced and used.
Example :

```
int a,b,c; // variables declaraation using comma operator
a=b++, c++; // a = c++ will be done.
```

Q.1. (C) Explain use of followings in C++ programming, with an example program for each.

(a) Nested if

It is always legal to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a **nested if** statement is as follows:

```
if( boolean_expression 1)
{
  // Executes when the boolean expression 1 is true
if(boolean_expression 2)
 {
    // Executes when the boolean expression 2 is true
}
}
```

You can nest **else if...else** in the similar way as you have nested *if* statement.

Example

```
#include <iostream>
```

```
using namespace std;
int main ()
{ // local variable declaration:
 int a = 100; int b = 200; // check the boolean condition
if( a == 100 ) {
 // if condition is true then check the following if( b == 200 )
{ // if condition is true then print the following
cout << "Value of a is 100 and b is 200" << endl; } }
 cout << "Exact value of a is : " << a << endl; cout << "Exact value of b is : " << b << endl;
 return 0; }
```

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

```
Value of a is 100 and b is 200
 Exact value of a is : 100
Exact value of b is : 200
```

(b) While loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

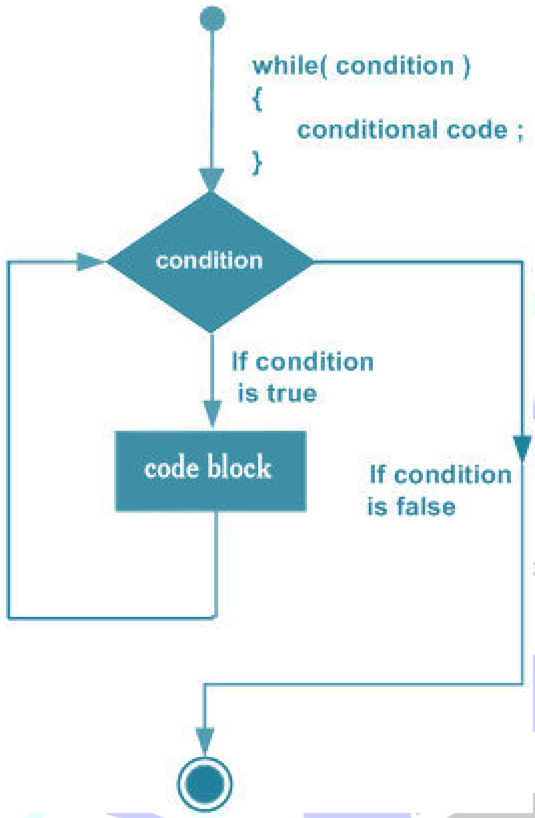
The syntax of a while loop in C++ is:

```
while(condition)
{
  statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#include <iostream>
using namespace std;
 int main ()
{ // Local variable declaration:
 int a = 10; // while loop execution
while( a < 20 )
{ cout << "value of a: " << a << endl; a++; }
 return 0;
}
```

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

```
value of a: 10
value of a: 11
 value of a: 12
value of a: 13
 value of a: 14
 value of a: 15
value of a: 16
 value of a: 17
value of a: 18
value of a: 19
```

Q.2. (a) Define class. Explain how an object is created in C++ ,with the help of an example. Also explain how destructor is defined in C++.

A.2.(a)

C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.
A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows:

```
class Box {



 public:
   double length;   // Length of a box
   double breadth;  // Breadth of a box
   double height;   // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

Box Box1; // Declare Box1 of type Box
Box Box2; // Declare Box2 of type Box
Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear:

```
#include <iostream>
using namespace std;
class Box {
 public:
     double length;   // Length of a box
     double breadth;  // Breadth of a box
     double height;   // Height of a box
};
int main( ) {
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here
 // box 1 specification
   Box1.height = 5.0;
  Box1.length = 6.0;
  Box1.breadth = 7.0;
   // box 2 specification
  Box2.height = 10.0;
   Box2.length = 12.0;
  Box2.breadth = 13.0;
   // volume of box 1
   volume = Box1.height * Box1.length * Box1.breadth;
 cout << "Volume of Box1 : " << volume <<endl;
   // volume of box 2




  volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Destructors

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a tilde ~ sign as prefix to it.

```
class A
{
 public:
 ~A();
};
```

Destructors will never have any arguments.

Example to see how Constructor and Destructor is called

```
class A
{
```

```
A()
 {
 cout << "Constructor called";
 }
~A()
 {
 cout << "Destructor called";
}
};
int main()
{
 A obj1;   // Constructor Called
 int x=1
 if(x)
 {
 A obj2;  // Constructor Called
 }   // Destructor Called for obj2
} //  Destructor called for obj1
```

Q.2. (b) Explain the following in detail, in context of C++ programming.

A.2.(b)

(i) Access Specifier :-

Access specifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,
public
private
protected
These access specifiers are used to set boundaries for availability of members of class be it data members or member functions
Access specifiers in the program, are followed by a colon. You can use either one, two or all 3 specifiers in the same class to set different boundaries for different class members. They change the bounday for all the declarations that follow them.

Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

```
class PublicAccess
{ public: // public access specifier
int x; // Data Member Declaration
 void display(); // Member Function decaration
}
```

Private

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.
class PrivateAccess { private: // private access specifier int x; // Data Member Declaration void display(); // Member Function decaration }

Protected

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A. We will learn this later.)

```
class ProtectedAccess
 {
 protected: // protected access specifier
int x; // Data Member Declaration
void display(); // Member Function decaration
}
```

(ii) Virtual Function :-

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class Employee , the class contains virtual functions like raiseSalary(), transfer(), promote(),.. etc. Different types of employees like Manager, Engineer, ..etc may have their own implementations of the virtual functions present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if raiseSalary() is present for a specific employee type, only that function would be called.

```
class Employee
{
public:
   virtual void raiseSalary()
{  /* common raise salary code */  }
   { /* common promote code */ }
virtual void promote()
};
    virtual void raiseSalary()
class Manager: public Employee {
        increment of manager specific incentives*/  }
```

```
{  /* Manager specific raise salary code, may contain

    virtual void promote()

void globalRaiseSalary(Employee *emp[], int n)

{ /* Manager specific promote */ }
};


{
    for (int i = 0; i < n; i++)

                              // according to the actual object, not

emp[i]->raiseSalary(); // Polymorphic Call: Calls raiseSalary()
```

(iii) Friend Function :-

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

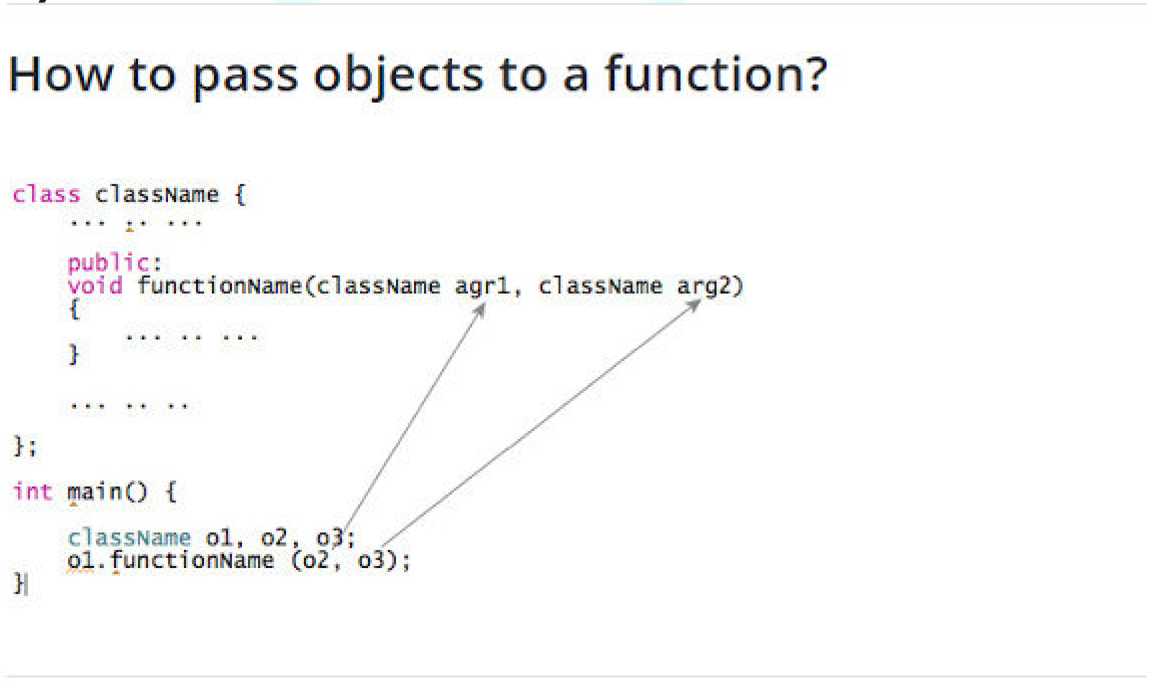
 }

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

```
class Box
{
 double width;
public: double length;
friend void printWidth( Box box );

void setWidth( double wid );
```

Q.2.(c) Explain how an object is passed as a parameter to a function, with the help of a program.

A.2.(c)



How to pass objects to a function?

```
class className {
    ... .. ...

    public:
    void functionName(className agr1, className arg2)
    {
        ... .. ...
    }

    ... .. ..
};

int main() {

    className o1, o2, o3;
    o1.functionName (o2, o3);
}
```

```
#include <iostream>
using namespace std;
class Complex
{
      int real;
private:
      int imag;
      Complex(): real(0), imag(0) { }
public:
      void readData()

         cout << "Enter real and imaginary number respectively:"<<endl;

{
         cin >> real >> imag;

       void addComplexNumbers(Complex comp1, Complex comp2)
}
      {

        // real represents the real data of object c3 because this function is called
using code c3.add(c1,c2);

real=comp1.real+comp2.real;


         imag=comp1.imag+comp2.imag;

// imag represents the imag data of object c3 because this function is called using
code c3.add(c1,c2);
      }

      void displaySum()

   c3.addComplexNumbers(c1, c2);

{
         cout << "Sum = " << real<< "+" << imag << "i";
      }
};
int main()
{
   Complex c1,c2,c3;

   c1.readData();
   c2.readData();
}
c3.displaySum();
   return 0;
```

Q.3. (a) What is constructor? Explain how it is overloaded, with the help of a C++ program.

A.3.(a) Constructors are special type of member functions that initialises an object automatically when it is created.
Compiler identifies a given member function is a constructor or not by its name and the return type.
Constructor has the same name as that of the class and it does not have any return type. Also, the constructor is always be public.

```
class temporary

{
```

```
private: int x;
 float y;
public: // Constructor
temporary(): x(5), y(5.5)
{
// Body of constructor
} ... .. ... };
 int main()
{
Temporary t1;
 }

#include <iostream>


using namespace std;

class Area
{

       int length;

private:

    public:

int breadth;


       Area(): length(5), breadth(2){ }

// Constructor


           cout << "Enter length and breadth respectively: ";

void GetLength()
       {

       int AreaCalculation() {  return (length * breadth);  }

cin >> length >> breadth;
       }
           cout << "Area: " << temp;

void DisplayArea(int temp)
       {
       }
};
```

Example Of Constructors :-

```
int main()

{

    Area A1, A2;

int temp;


    A1.GetLength();

temp = A1.AreaCalculation();

    cout << endl << "Default Area when value is not taken from
user" << endl;

A1.DisplayArea(temp);
temp = A2.AreaCalculation();
    A2.DisplayArea(temp);

}
    return 0;
```

Constructor Overloading

Constructor can be overloaded in a similar way as function overloading.
Overloaded constructors have the same name (name of the class) but different number of arguments.
Depending upon the number and type of arguments passed, specific constructor is called.
Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Example:-

```
// Source Code to demonstrate the working of overloaded constructors
#include <iostream>
using namespace std;

class Area
{
   private:

      // Constructor with no arguments

int length;
      int breadth;
```

```
    public:

       Area(int l, int b): length(l), breadth(b){ }

Area(): length(5), breadth(2) { }

       // Constructor with two arguments


          cout << "Enter length and breadth respectively: ";

void GetLength()
       {
          cin >> length >> breadth;
       }
          cout << "Area: " << temp << endl;

int AreaCalculation() {  return length * breadth;  }

       void DisplayArea(int temp)
       {
       }
};

int main()
{

    A1.DisplayArea(temp);

Area A1, A2(2, 1);
    int temp;

    cout << "Default Area when no argument is passed." << endl;
    temp = A1.AreaCalculation();
}

cout << "Area when (2,1) is passed as argument." << endl;
    temp = A2.AreaCalculation();
    A2.DisplayArea(temp);

    return 0;
```

Q.3. (b) What is inheritance? What are different types of inheritance? Explain how multiple inheritance is implemented in C++, with the help of a program.

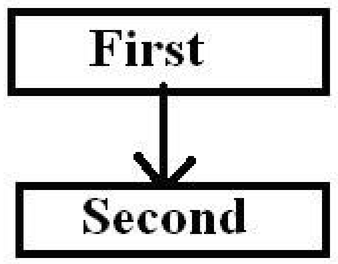
A.3.(b)

Inheritance:-Inheritance means using the Pre-defined Code This is very Main Feature of OOP With the advantage of Inheritance we can use any code that is previously created. With the help of inheritance we uses the code that is previously defined but always Remember, We are only using that code but not changing that code.

With the Advent of inheritance we are able to use pre-defined code and also able to add new code. All the pre-defined code is reside into the form of classes if we want to use that code then we have to inherit or extend that class.
The Class that is Pre-defined is called as Base or super Class and the class which uses the Existing Code is known as derived or sub class The Various Types of Inheritance those are provided by C++ are as followings:

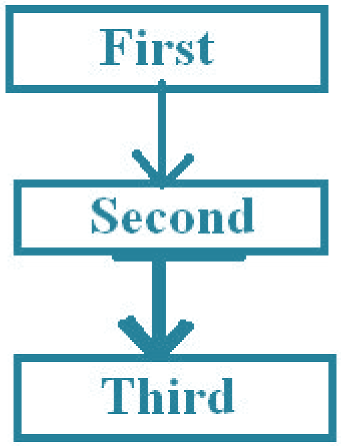
1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

In Inheritance Upper Class whose code we are actually inheriting is known as the Base or Super Class and Class which uses the Code are known as Derived or Sub Class.
1) In Single Inheritance there is only one Super Class and Only one Sub Class Means they have one to one Communication between them
Single Inheritance



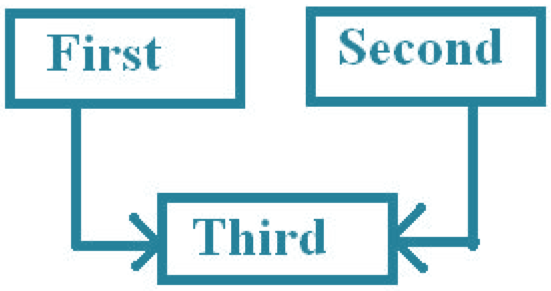
2) In Multilevel Inheritance a Derived class can also inherited by another class Means in this Whena Derived Class again will be inherited by another Class then it creates a Multiple Levels.

Multilevel Inheritance



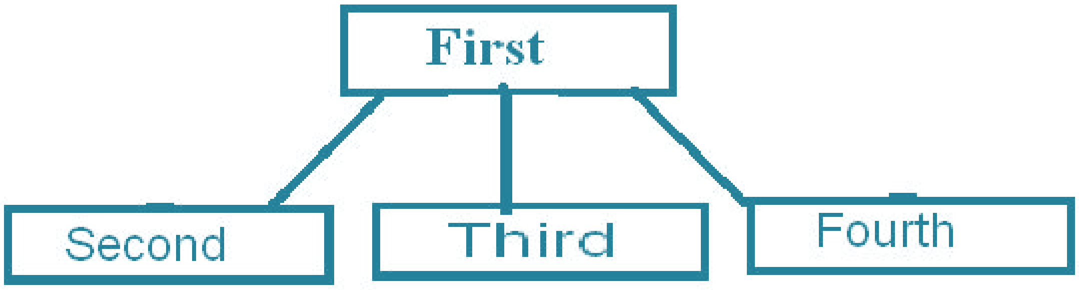
3) Multiple Inheritances is that in which a Class inherits the features from two Base Classes When a Derived Class takes Features from two Base Classes.

Multiple Inheritances



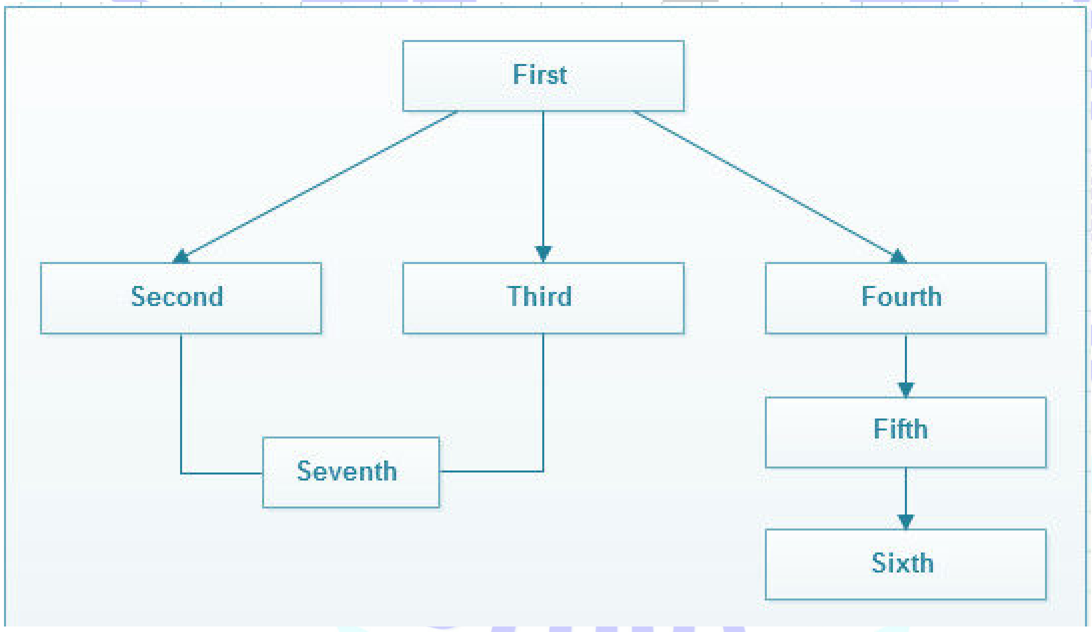
4) Hierarchical Inheritance is that in which a Base Class has Many Sub Classes or When a Base Class is used or inherited by many Sub Classes.

Hierarchical Inheritance



5) Hybrid Inheritance: - This is a Mixture of two or More Inheritance and in this Inheritance a Code May Contains two or Three types of inheritance in Single Code.

Hybrid Inheritance



Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.

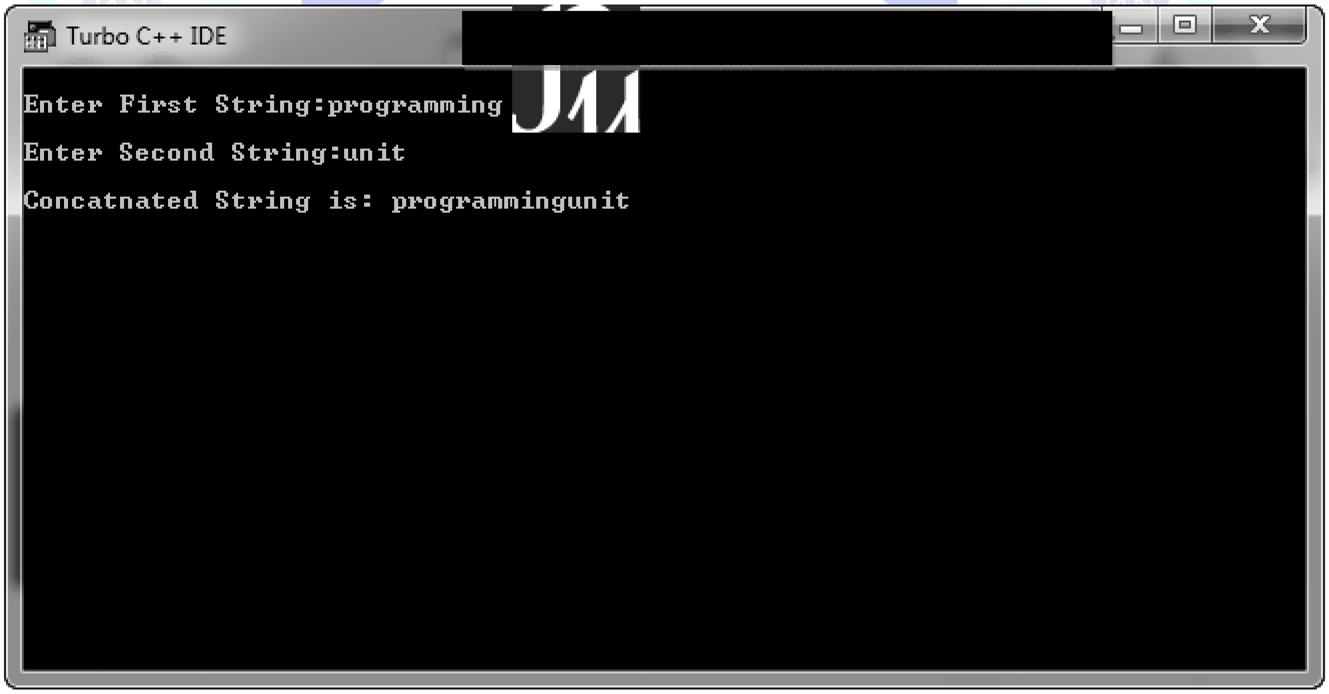
The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B’s constructor is called before A’s constructor.

#include<iostream>using namespace std; class A{public: A() { cout << "A's constructor called" << endl; }}; class B{public: B() { cout << "B's constructor called" << endl; }}; class C: public B, public A // Note the order{public: C() { cout << "C's constructor called" << endl; }}; int main(){   C c;   return 0;}

Q.3. (c) Write a C++ program to overload ‘+’ operator in such a way that it return the sum of lengths of two strings (Note: if S1 and S2 are two strings then S1+S2 or S2 + S1 should give the sum of lengths of S1 and S2).

A.3.(c)

```
#include<conio.h>
#include<string.h>
#include<iostream.h>
class string {

    char *s;

public:
    int size;

    void getstring(char *str)

{
        size = strlen(str);

    }

s = new char[size];
        strcpy(s,str);


void string::operator+(string ob)

void operator+(string);
};

{
    size = size+ob.size;

    cout<<"\nConcatnated String is: "<<s;
s = new char[size];
    strcat(s,ob.s);
}

void main()
{
    string ob1, ob2;

    cin>>string1;

char *string1, *string2;
    clrscr();

    cout<<"\nEnter First String:";

    ob1.getstring(string1);


    //Calling + operator to Join/Concatenate strings

cout<<"\nEnter Second String:";
    cin>>string2;

    ob2.getstring(string2);

    ob1+ob2;
    getch();
}
```



[redacted]

```
Enter First String:programming

Enter Second String:unit

Concatnated String is: programmingunit
```

Q.4.What is stream manipulator? Explain use of setw( ) and setprecision( ) as stream manipulator.

A.4.

C++ offers the several input/output manipulators for
formatting, commonly used manipulators are given below..

Manipulator	Declaration in
endl	iostream.h
setw	iomanip.h
setprecision	iomanip.h
setf	iomanip.h

setw() and setfill() manipulators
setw manipulator sets the width of the filed assigned for
the output.to be written in some output representations. If the standard width of the
The field width determines the minimum number of characters
representation is shorter than the field width, the representation is padded

with fill characters (using setfill).

setfill character is used in output insertion operations to

fill spaces when results have to be padded to the field width.

Syntax
setw([number_of_characters]);

setfill([character]);

Consider the example

```
#include <iostream.h>

#include

   int main()

<iomanip.h>


   {

                 cout<<"USING setw()

.............\n";
```

```
                    cout<< setw(10)

                    cout<< setw(10)

<<11<<"\n";

   cout<< setw(10)
<<2222<<"\n";


<<33333<<"\n";
    cout<< setw(10)

<<4<<"\n";

                    cout<<"USING setw() &

setfill() [type- I]...\n";

    cout<< setfill('0');

<<11<<"\n";


     cout<< setw(10)

    cout<< setw(10)

<<2222<<"\n";

       cout<< setw(10)

<<33333<<"\n";

                    cout<<"USING setw() &

cout<< setw(10)
<<4<<"\n";

                    cout<< setfill('-')<< setw(10)

setfill() [type-II]...\n";

<<11<<"\n";


  cout<< setfill('@')<< setw(10)

cout<< setfill('*')<< setw(10)
```

<<2222<<"\n";

<<33333<<"\n";

Output

```
cout<< setfill('#')<< setw(10)
<<4<<"\n";
   return 0;

    }
     USING setw()

..............
          2222

11
          33333

             4
```

USING setw() &

setfill() [type- I]...

```
  0000000011

 0000002222

0000033333



0000000004
```

USING setw() &

setfill() [type-II]...

```
    --------11
```

```
******2222
   #########4
@@@@@33333
```

setf() and setprecision() manipulator

setprecision manipulator sets the total number of digits to

be displayed, when floating point numbers are printed.

Syntax

setprecision([number_of_digits]);

: 1234.5

cout<<setprecision(5)<<1234.537;

// output will be

specifies the maximum number of meaningful digits to display in total countingboth those before and those after the decimal point. Notice that it is not a

On the default floating-point notation, the precision field

minimum and therefore it does not pad the displayed number with trailing zeros

if the number can be displayed with less digits than the precision.

In both the fixed and scientific notations, the precision

even if this includes trailing decimal zeros. The number of digits before the

field specifies exactly how many digits to display after the decimal point, decimal point does not matter in this case.

Syntax

setf([flag_value],[field bitmask]);

field bitmask   flag
values

right or internal

adjustfield   left,

basefield   dec,

Consider the example

oct or hex

floatfield   scientific
or fixed
#include <iostream.h>
#include <iomanip.h>
        cout<<"USING

int main()

{

fixed ....\n";

            cout.setf(ios::floatfield,ios::fixed);

cout<<setprecision(5)<<1234.537<< endl;

scientific ....\n";

cout<<"USING

            cout<<setprecision(5)<<1234.537<< endl;

cout.setf(ios::floatfield,ios::scientific);

```
                return 0;

    USING scientific

}

Output:-

    USING fixed
.......................

    1234.53700

#include <iostream.h>

..................

    1234.5
```

Consider the example to illustrate base fields

```
                cout<<"Decimal

#include <iomanip.h>

int main()

{
                int
num=10;

value is :"<< num << endl;

                cout<<"Octal value is :"<< num << endl;

cout.setf(ios::basefield,ios::oct);

                cout.setf(ios::basefield,ios::hex);

}
```

cout<<"Hex value is :"<< num << endl;

return 0;

Q.4.(b) What is template? Write appropriate statements to create a template class for Queue data structure in C++.

A.4.(b)

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>**.

You can use templates to define functions as well as classes, let us see how do they work:

Function Template

The general form of a template function definition is shown here:

```
emplate <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> elements;

  public:
    void push(T const &);
    void pop();
    T top();
    bool empty();
};
```

```
template <class T>
void Stack<T>::push(T const &elem) {
    elements.push_back(elem);
}

template <class T>
void Stack<T>::pop() {
    if (elements.empty()) {
        throw out_of_range("Stack<>::pop(): empty stack");
    } else {
        elements.pop_back();
    }
}

template <class T>
T Stack<T>::top() {
    if (empty()) {
        throw out_of_range("Stack<>::top(): empty stack");
    }
    return elements.back();
}

template <class T>
bool Stack<T>::empty() {
    return elements.empty();
}


int main() {
    try {
        Stack<int> intStack;       // stack of ints
        Stack<string> stringStack; // stack of strings

        // manipulate integer 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
```

Q.4.(c)What are containers? Explain use of List container class with an example.

A.4.(c)

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.
The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).
Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...
Many containers have several member functions in common, and share functionalities.
The decision of which type of container to use for a specific need does not on the efficiency of some of its members (complexity). This is especially true generally depend only on the functionality offered by the container, but also inserting/removing elements and accessing them.
for sequence containers, which offer different trade-offs in complexity between stack,queue and priority_queue are implemented as container adaptors. Container adaptors are not full container classes, but classes that provide a specific or list) to handle the elements. The underlying container is encapsulated in interface relying on an object of one of the container classes (such as deque such a way that its elements are accessed by the members of the container adaptor independently of the underlying container class used.

Container class templates
Sequence
containers:
array
Array class (class template)
Vector
Vector(class template)
deque
Double
ended queue (class template)
forward_list

Forward
list (class template)
list
List(class template)
Container adaptors:
Stack
LIFO
stack (class template)
queue
FIFO
queue (class template)
priority_queue
Priority queue (class template)
Associative containers:
set
Set(class template)
multiset
Multiple-key
set (class template)
map
Map(class template)
multimap
Multiple-key
map (class template)
Unordered associative containers:
unordered_set
Unordered Set (class template)
unordered_multiset
Unordered Multiset (class template)
unordered_map
Unordered Map (class template)
unordered_multimap
Unordered Multimap (class template)

Q.5.(a) What is exception? How exceptions are handled in C++? Write program to handle stack overflow as exception.

A.5.(a)

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch,and throw.

· **throw**: A program throws an exception when a problem shows up. This is done using a throw keyword.

· **catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

· **try**: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block wil raise an exception,a method catches an exception using a combination of the try and catch keywords.
A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
 // protected code
}catch( ExceptionName e1 )
{

 // catch block
{

}catch( ExceptionName e2 )


 // catch block


}

}catch( ExceptionName eN )

{

 // catch block
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Throwing Exceptions
Exceptions can be thrown anywhere within a code block using throwstatements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.
Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b) {

 if( b == 0 ) {

 throw "Division by zero condition!";

 }

 return (a/b);

}
```

Catching Exceptions

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {

 // protected code

}catch( ExceptionName e ) {

 // code to handle ExceptionName exception

}
```

Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows

```
try {

 // protected code

}catch(...) {
 // code to handle any exception

}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

```
#include <iostream>

using namespace std;

double division(int a, int b) {

  if( b == 0 ) {

    throw "Division by zero condition!";

  }

  return (a/b);

}
int main () {
```

Because we are raising an exception of type const char*, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

```
int x = 50;

  int y = 0;

  double z = 0;

  try {

    z = division(x, y);

    cout << z << endl;

  }catch (const char* msg) {

    cerr << msg << endl;

  }
  return 0;
}
```

Q.5.(b)What is function overloading? Explain with an example

A.5.(b)

Function refers to a segment that groups code to perform a specific task.
In C++ programming, two functions can have same name if number and/or type of arguments passed are different.
These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }

int test(int a) { }

float test(double a) { }

int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.
Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code
int test(int a) { }
double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example 1: Function Overloading

```
#include <iostream>
```

```
using namespace std;
void display(int);
void display(float);
void display(int, float);
int main() {
   int a = 5;
   float b = 5.5;
   display(a);
   display(b);
   display(a, b);
   return 0;
}
void display(int var) {
   cout << "Integer number: " << var << endl;
}
void display(float var) {
   cout << "Float number: " << var << endl;
}
void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Q.5.(c) Write C++ program to create a file and store your contact details in it.

A.5.(c)
When a program runs, the data is in the memory but when it ends or the computer shuts down, it gets lost. To keep data permanently, we need to write it in a file.
File is used to store data. In this topic, you will learn about reading data from a file and writing data to the file.
fstream is another C++ standard library like iostream and is used to read and write on files.
These are the data types used for file handling from the **fstream** library:

Data type	Description
ofstream	It is used to create files and write on files.
ifstream	It is used to read from files.
fstream	It can perform the function of both of stream and If stream which means it can create files, write on files, and read from files.

Opening a file

We need to tell the computer the purpose of opening our file. For e.g.- to write on the file, to read from the file, etc. These are the different modes in which we can open a file.

Mode	Description
ios::app	opens a text file for appending. (appending means to add text at the end).
ios::ate	opens a file for output and move the read/write control to the end of the file.
ios::in	opens a text file for reading.
ios::out	opens a text file for writing.
ios::trunc	truncates the content before opening a file, if file exists.

Let's look at the syntax of opening a file.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
ofstream file;
file.open ("example.txt");
return 0;
}
```

We have opened the file 'example.txt' to write on it. 'example.txt' file must be created in your working directory. We can also open the file for both reading and writing purposes. Let's see how to do this:

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
fstream file;
file.open ("example.txt", ios::out | ios::in );
return 0;
}
```

Closing a file

C++ automatically close and release all the allocated memory. But a programmer should always close all the opened files. Let's see how to close it.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
ofstream file;
file.open ("example.txt");
file.close();
return 0;
}
```

Reading and writing on a file

We use **<<** and **>>** to write and read from a file respectively. Let's see an example.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
char text[200];
fstream file;
file.open ("example.txt", ios::out | ios::in );
cout << "Write text to be written on file." << endl;
cin.getline(text, sizeof(text));
// Writing on file
file << text << endl;
// Reding from file
file >> text;
cout << text << endl;
//closing the file
file.close();
return 0;
}
```