Improvement Test – May 2017

Sub:	Programming in C++					**Code:**	10EC665/10TE661
Date:	31/05/17	Duration:	90 mins	Max Marks:	50	**Sem:** VI	**Branch:** ECE/TCE

Note: Answer any 5 questions. All questions carry equal marks.
Total marks: 50

			Marks	OBE	
				CO	RBT
1	(a)	Write a C++ program to overload + operator to add two strings.	[05]	CO2	L2
	(b)	Write a C++ program where a member function receives argument as object & returning object.	[05]	CO2	L2
2.		Explain various types of inheritances of C++ with diagrams.	[10]	CO2	L2
3.		Write-up details of access control mechanism with table for type of members & type of derivation	[10]	CO3	L3
4.		Define a class customer with data members accno, name and balance and public member functions for: deposit( ), withdraw( ) (with checking balance) & show( ). Call member function using main function.	[10]	CO3	L3
5.		What is virtual base class? What is problem with diamond inheritance? Write solution with example program.	[10]	CO3	L2
6.		Write one example program in C++ to initialize base class using derived class constructor.	[10]	CO3	L2
7.		What are exceptions & exception handling? Define mechanism of C++ for the same using try, throw & catch blocks with example program as division by zero exception.	[10]	CO3	L3
8.		Explain use of virtual function in run-time polymorphism with example program.	[10]	CO4	L3

Question #	Description	Marks Distribution		Max Marks
1 a	program to overload + operator to add two strings.	5M	5M	5M
1 b	program where a member function receives argument as object & returning object.	2.5M 2.5M	5M	5M
2	types of inheritances of C++ with diagrams	10M	10M	10M
3	access control mechanism with table for type of members & type of derivation	5M 5M	10M	10M
4	class customer with data members accno, name and balance and public member functions for: deposit( ), withdraw( ) (with checking balance) & show( ).	10M	10M	10M
5	virtual base class diamond inheritance Solution with example program.	3M 3M 4M	10M	10M
6	Program in C++ to initialize base class using derived class constructor.	10M	10M	10M
7	exceptions & exception handling Define mechanism of C++ for the same using try, throw & catch blocks with example program as division by zero exception.	4M 6M	10M	10M
8	virtual function in run-time polymorphism with example	5M 5M	10M	10M

QUESTION BANK WITH SOLUTION FOR IMPROVEMENT TEST

Explain exception handling in detail. Why do we require it? Name different types of exception.

Explain try, throw and catch blocks / Explain how to throw exception and how to catch exception.

What are exception specifications?

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.

Types of Exception :

1) Synchronous Exception

2) Asynchronous Exception

Assuming a block will 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 **throw** statements. 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 ()
{
  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;
}
```

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:

Division by zero condition!

Different ways of throwing an exception

throw()

throw no.;

throw

Rethrowing mechanism

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

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

int main () {
  try
  {
    throw 20;                                   An exception occurred.
  }                                             Exception Nr. 20
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr.
" << e << '\n';
  }
  return 0;
}
```

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

```
  throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the catch keyword immediately after the closing brace of the try block. The syntax for catch is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught by that handler.

Multiple handlers (i.e., catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the

type of the exception specified in the throw statement is executed.

If an ellipsis (...) is used as the parameter of catch, that handler will catch any exception no matter what the type of the exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers:

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

In this case, the last handler would catch any exception thrown of a type that is neither int nor char.

After an exception has been handled the program, execution resumes after the *try-catch* block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

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

}

Exception specification

Older code may contain *dynamic exception specifications*. They are now deprecated in C++, but still supported. A *dynamic exception specification* follows the declaration of a function, appending a throw specifier to it. For example:

double myfunction (*char* param) *throw* (*int*);

This declares a function called myfunction, which takes one argument of type char and returns a value of type double. If this function throws an exception of some type other than int, the function calls std::unexpected instead of looking for a handler or calling std::terminate.

If this throw specifier is left empty with no type, this means that std::unexpected is called for any exception. Functions with no throw specifier (regular functions) never call std::unexpected, but follow the normal path of looking for their exception handler.

1 *int* myfunction (*int* param) *throw*(); *// all exceptions call unexpected*
2 *int* myfunction (*int* param); *// normal exception handling*

It is better to consider class notes for programming examples.

Explain Inheritance with Types with suitable diagram.

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. T his existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the " **is a"** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. T o define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

class derived-class: access-specifier base-class

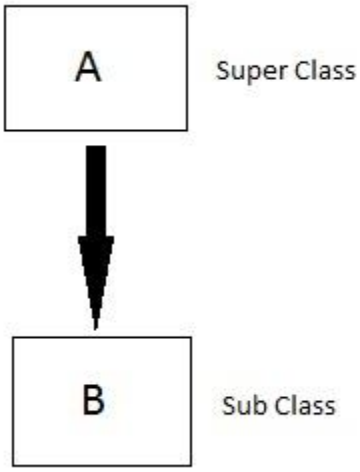
Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

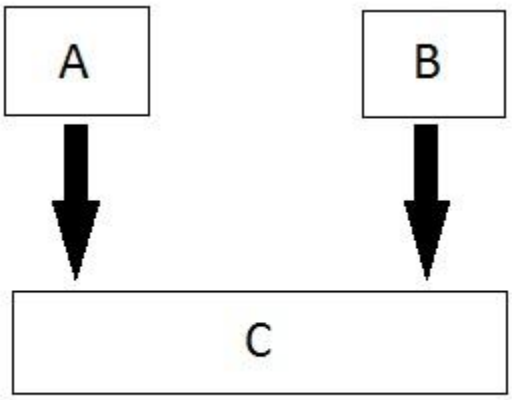
Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



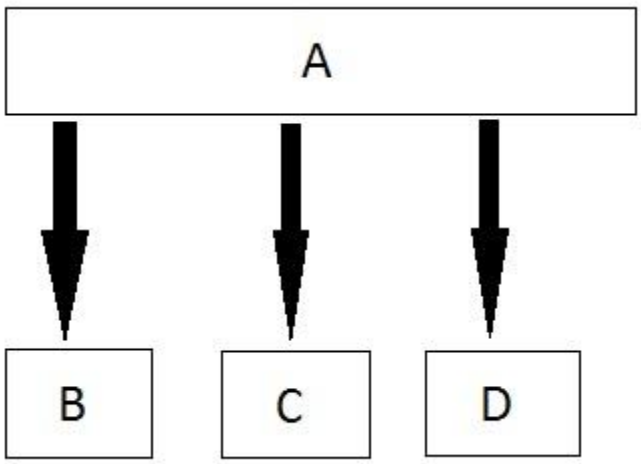
Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



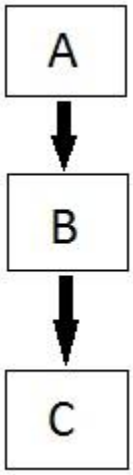
Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



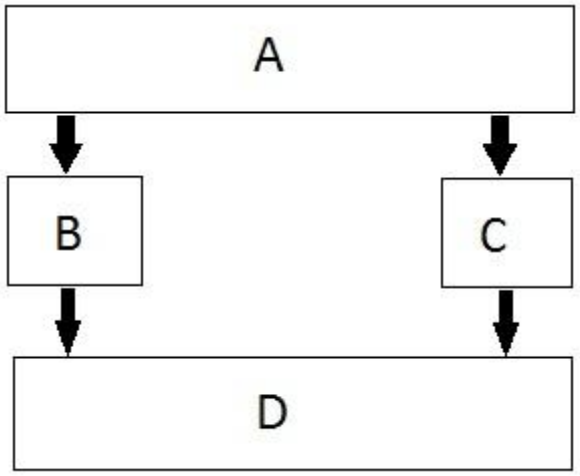
Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



Explain how to Inherit base class in Public, Private and Protected Mode.

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

Consider Table written in class while explaining concept

Access	**public**		**protected**	**private**
Same class	yes	yes	yes	
Derived classes		yes	yes	no
Outside classes		yes	no	no

A derived class inherits all base class methods with the following exceptions:
Constructors, destructors and copy constructors of the base class.
Overloaded operators of the base class.
The friend functions of the base class.

Type of Inheritance:

When deriving a class from a base class, the base class may be inherited through **public , protected** or
private inheritance. T he type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using
different type of inheritance, following rules are applied:

Public Inheritance: When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible
directly from a derived class, but can be accessed through calls to the **public** and **protected** members
of the base class.

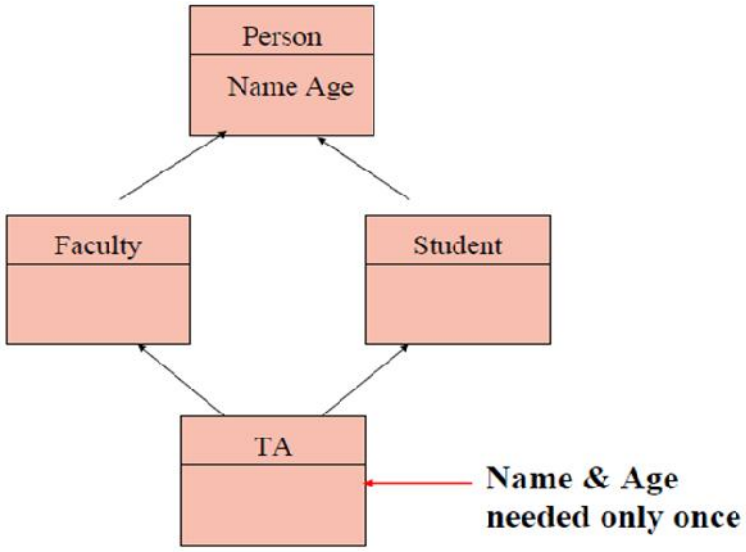
Protected Inheritance: When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

Private Inheritance: When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Explain Virtual Base class.

The diamond problem

The diamond problem occurs when two super classes of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
```

```
  // Data members of person
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
};

class Faculty : public Person {
  // data members of Faculty
public:
   Faculty(int x):Person(x)   {
      cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : public Person {
  // data members of Student
public:
   Student(int x):Person(x) {
       cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x)   {
       cout<<"TA::TA(int ) called"<< endl;
   }
};

int main()  {
   TA ta1(30);
}
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. *The solution to this problem is 'virtual' keyword*. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
   Person()     { cout << "Person::Person() called" << endl;   }
};

class Faculty : virtual public Person {
public:
   Faculty(int x):Person(x)   {
     cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : virtual public Person {
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x)   {
      cout<<"TA::TA(int ) called"<< endl;
   }
};

int main()  {
   TA ta1(30);
}
```

Output:
Person::Person() called
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called

In the above program, constructor of 'Person' is called once. One important thing to note in the above output is, *the default constructor of 'Person' is called*. When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to pass parameters to base class constructor through derived class?

Discuss cases

Case 1: base class and derived class both ha constructor

```
Class Base
{
Int xx;
Public:
Base(int x) { xx=x;}
…
};

Class derived: public base
{
Int yy, zz;
Public:
derived(int y , int z, int x):base(x)
{
yy=y;
zz=z;
}
…
};
Main()
{
derived ob1(2,3,4);  // xx=4  yy=2 zz=3
}
```

Case 2: derived class don’t have constructor but base class has.

```
Class Base
{
Int xx;
Public:
Base(int x) { xx=x;}
…
};

Class derived: public base
{

Public:
derived(int x):base(x)
{

}
…
};
Main()
{
derived ob1(2);  // xx=2
}
```

Case 3: both derived class and base class sharing value

```
Class Base
{
Int xx;
Public:
Base(int x) { xx=x;}
…
};

Class derived: public base
{
Int yy;
```

```
Public:
derived(int y ):base(y)
{
yy=y;
}
…
};
Main()
{
derived ob1(2);  // xx=2  yy=2
}
```

Explanation of the above concept is required in answer.

Explain Virtual Base class/ discuss diamond problem and solution for it in Inheritance.

Virtual base classes, used in virtual inheritance, is a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritance.
Consider the following scenario:

```
class A
{
 public: void Foo() { }
 };
class B : public A
 { };
class C : public A
 { };
class D : public B, public C
 { };
```

The above class hierarchy results in the "dreaded diamond" which looks like this:

```
  A
 / \
```

```
B   C
 \ /
  D
```

An instance of D will be made up of B, which includes A, and C which also includes A. So you have two "instances" (for want of a better expression) of A.
When you have this scenario, you have the possibility of ambiguity. What happens when you do this:

```
main()
{
D d;
d.Foo(); // is this B's Foo() or C's Foo() ??
}
```

Virtual inheritance is there to solve this problem. When you specify virtual when inheriting your classes, you're telling the compiler that you only want a single instance.

```
class A
{
 public: void Foo()
 { }
 };
class B : public virtual A
 { };
class C : public virtual A
 { };
class D : public B, public C
 { };
```

This means that there is only one "instance" of A included in the hierarchy. Hence

```
main()
{
D d;
d.Foo(); // no longer ambiguous
}
```

What is Runtime polymorphism/Late binding/ Virtual function? Explain with example. What is Pure Virtual function/Abstract class?

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

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

class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};

class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }
    int area () {
```

```
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};

class Triangle: public Shape{
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }
    int area () {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};

// Main function for the program
int main( ) {
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);

  // store the address of Rectangle
  shape = &rec;

  // call rectangle area.
  shape->area();

  // store the address of Triangle
  shape = &tri;
```

```
   // call triangle area.
   shape->area();

   return 0;
}
```

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

```
Parent class area
Parent class area
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword virtual so that it looks like this:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
```

```
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function

A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.

Pure Virtual Functions

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

```
class Shape {
  protected:
    int width, height;
  public:
```

```
    Shape( int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // pure virtual function
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

C++ Program to create a class EMPLOYEE (Calculate DA,Tax,Net Sal etc) Given that an EMPLOYEE class contains the following members:

Date Members: Employee_Number, Employee_Name, Basic, DA, IT, Net_Sal;
Member Function: to read data, to calculate Net_Sal and to print data members;

Write a C++ program to read data on N employees and compute the Net_Sal of each employee (DA = 52% of Basic and Income Tax = 30% of the gross salary).

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


class employee
{
        int   emp_num;
        char  emp_name[20];
        float emp_basic;
        float sal;
        float emp_da;
```

```
        float net_sal;
        float emp_it;

        public:

                void get_details();
                void find_net_sal();
                void show_emp_details();
};

void employee :: get_details()
{
        cout<<"\nEnter employee number:\n";
        cin>>emp_num;
        cout<<"\nEnter employee name:\n";
        cin>>emp_name;
        cout<<"\nEnter employee basic:\n";
        cin>>emp_basic;
}

void employee :: find_net_sal()
{
        emp_da=0.52*emp_basic;
        emp_it=0.30*(emp_basic+emp_da);
        net_sal=(emp_basic+emp_da)-emp_it;
```

```
}

void employee :: show_emp_details()
{
        cout<<"\n\n\nDetails of   :  "<<emp_name;
        cout<<"\n\nEmployee number:      "<<emp_num;
        cout<<"\nBasic salary     :  "<<emp_basic;
        cout<<"\nEmployee DA      :  "<<emp_da;
        cout<<"\nIncome Tax       :  "<<emp_it;
        cout<<"\nNet Salary       :  "<<net_sal;
}

int main()
{
        employee emp[10];
        int i,num;
        clrscr();

        cout<<"\nEnter number of employee details\n";
        cin>>num;

        for(i=0;i<num;i++)
                emp[i].get_details();

        for(i=0;i<num;i++)
```

```
                emp[i].find_net_sal();


        for(i=0;i<num;i++)
                emp[i].show_emp_details();


        getch();
        return 0;
}
```

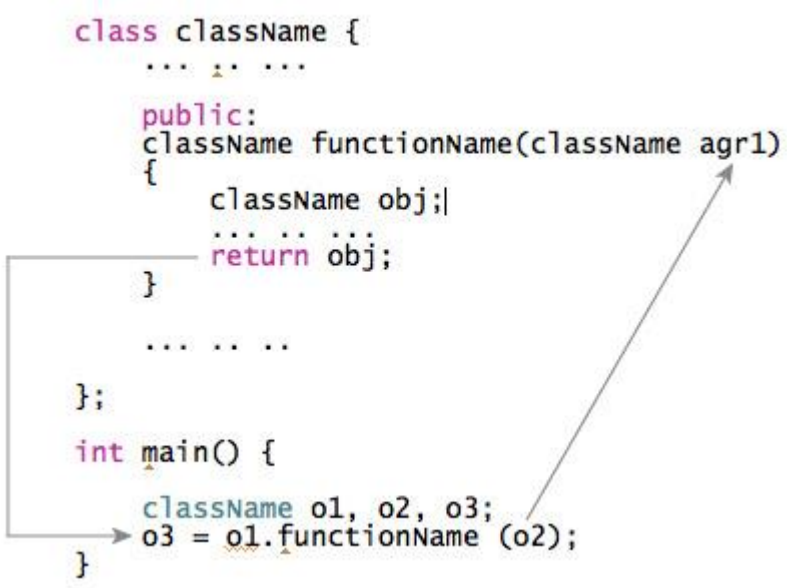
Write a program to count number of objects in C++.

```
class ObjectCount {
   static int count;
 protected:
   ObjectCount() {
      count++;
   }
 public:
   void static showCount() {
      cout << count;
   }
};

int ObjectCount::count = 0;

main()
{
ObjectCount o1,o2;
ShowCount();
}
```

Write a function returning object and taking object as argument. Also write suitable main function for same.

How to return an object from the function?

In C++ programming, object can be returned from a function in a similar way as structures.



Example 2: Pass and Return Object from the Function

In this program, the sum of complex numbers (object) is returned to the main() function and displayed.

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

```
    {
      cout << "Enter real and imaginary number respectively:"<<endl;
      cin >> real >> imag;
    }
    Complex addComplexNumbers(Complex comp2)
    {
      Complex temp;

      // real represents the real data of object c3 because this function is called using code
c3.add(c1,c2);
      temp.real = real+comp2.real;

      // imag represents the imag data of object c3 because this function is called using code
c3.add(c1,c2);
      temp.imag = imag+comp2.imag;
      return temp;
    }
    void displayData()
    {
      cout << "Sum = " << real << "+" << imag << "i";
    }
};

int main()
{
  Complex c1, c2, c3;

  c1.readData();
  c2.readData();
```

```
    c3 = c1.addComplexNumbers(c2);

    c3.displayData();

    return 0;
}
```

Explain use of scope resolution operator.

Scope resolution operator in C++
In C++, scope resolution operator is **::**. It is used for following purposes.
1) To access a global variable when there is a local variable with same name:

```
// C++ program to show that we can access a global variable
// using scope resolution operator :: when there is a local
// variable with same name
#include&lt;iostream&gt;
using namespace std;

int x;  // Global x

int main()
{
  int x = 10; // Local x
  cout<<::x;//0
  cout << x;  //10
  return 0;
}
```

Output:
Value of global x is 0
Value of local x is 10

2) To define a function outside a class.

```
// C++ program to show that scope resolution operator :: is used
```

```
// to define a function outside a class
#include&lt;iostream&gt;
using namespace std;

class A
{
public:

  // Only declaration
  void fun();
};

// Definition outside class using ::
void A::fun()
{
  }
```

3) To access a class's static variables.

```
// C++ program to show that :: can be used to access static
// members when there is a local variable with same name

class Test
{
   static int x;
public:
   static int y;

   // Local parameter 'a' hides class member
   // 'a', but we can access it using ::
   void func(int x)
   {       // We can access class's static variable       // even if there is a local variable
         }
};

// In C++, static members must be explicitly defined
// like this
```

```
int Test::x = 1;
int Test::y = 2;

int main()
{
   Test obj;
   int x = 3 ;
   obj.func(x);


   return 0;
}
```

4) In case of multiple Inheritance:

If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

```
// Use of scope resolution operator in multiple inheritance.
#include&lt;iostream&gt;
using namespace std;

class A
{
protected:
   int x;
public:
   A() { x = 10; }
};

class B
{
protected:
   int x;
public:
   B() { x = 20; }
};

class C: public A, public B
```

```
{
public:
  void fun()
  {
    cout <<"A's x is"<< A::x;
    cout<<"B's x is" <<B::x;
  }
};

int main()
{
   C c;
   c.fun();
   return 0;
}
```

Run on IDE

Output:

A's x is 10

B's x is 20

Write note on

1) **Dynamic memory allocation vs Static allocation of object in memory**

Static Allocation means, that the memory for your variables is allocated when the program starts. The size is fixed when the program is created. It applies to global variables, file scope variables, and variables qualified with static defined inside functions.

Eg.

```
Class sample
{ };
main()
{
Sample ob;
}
```

Dynamic memory allocation is a bit different. You now control the exact size and the lifetime of these memory locations. If you don't free it, you'll run into memory leaks, which may cause your application to crash, since it, at some point cannot allocation more memory.

Eg.

```
Class sample
{
Int  func() {return 10;}
   };

main()
{
Sample *o;
O=new sample;
cout<<O->func();
delete o  ;

}
```

When you are done with the memory, you have to free it:

```
delete o  ;
```

2) Pointer to class

A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator -> operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

Let us try the following example to understand the concept of pointer to a class:

```
#include <iostream>


using namespace std;


class Box {
  public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
```

```
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }

    double Volume() {
      return length * breadth * height;
    }

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

int main(void) {
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2
  Box *ptrBox;                // Declare pointer to a class.

  // Save the address of first object
  ptrBox = &Box1;

  // Now try to access a member using member access operator
  cout << "Volume of Box1: " << ptrBox->Volume() << endl;

  // Save the address of second object
  ptrBox = &Box2;
```

```
  // Now try to access a member using member access operator
  cout << "Volume of Box2: " << ptrBox->Volume() << endl;


  return 0;
}
```

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

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

1. **Explain difference between copy constructor and object assignment**

Consider the following C++ program.

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

using namespace std;

class Test
{
public:
  Test() {}
  Test(const Test &t)
  {
    cout<<"Copy constructor called "<<endl;
  }
  Test& operator = (const Test &t)
  {
    cout<<"Assignment operator called "<<endl;
  }
};

int main()
{
 Test t1, t2;
 t2 = t1;
 Test t3 = t1;
 getchar();
 return 0;
```

}
Run on IDE
Output:
Assignment operator called
Copy constructor called

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object (see this G-Fact). And assignment operator is called when an already initialized object is assigned a new value from another existing object.

```
t2 = t1;  // calls assignment operator, same as "t2.operator=(t1);"
Test t3 = t1;  // calls copy constructor, same as "Test t3(t1);"
```