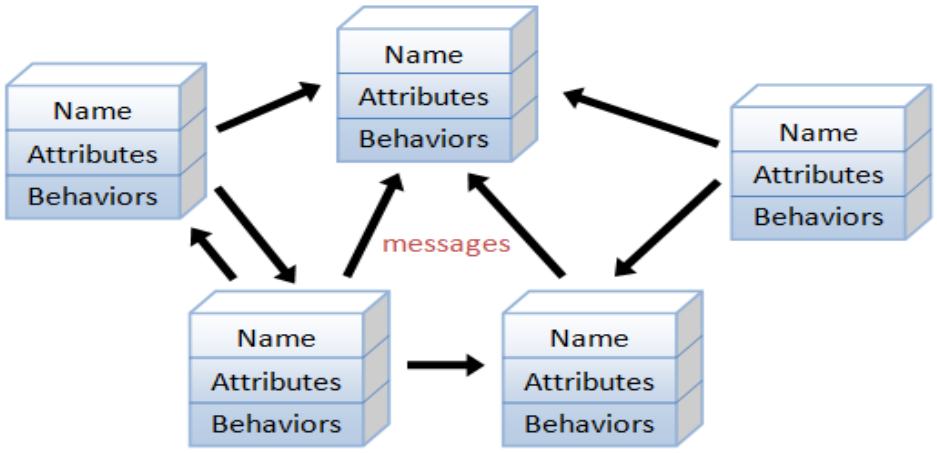
Unit-1/Lecture-1

Object-oriented programming (OOP) [RGPV/June2014(7),June2013(8), June2011(10), Feb2010(10),June2009 (10)]

OOPs languages are designed to overcome these problems.
The basic unit of OOP is a class, which encapsulates both the static attributes and dynamic behaviors within a "box", and specifies the public interface for using these boxes. Since the class is well-encapsulated (compared with the function), it is easier to reuse these classes. In other words, OOP combines the data structures and algorithms of a software entity inside the same box.

OOP languages permit higher level of abstraction for solving real-life problems. The traditional procedural language (such as C and Pascal) forces you to think in terms of the structure of the computer (e.g. memory bits and bytes, array, decision, loop) rather than thinking in terms of the problem you are trying to solve. The OOP languages (such as Java, C++, C#) let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Benefits of OOP

The procedural-oriented languages focus on procedures, with function as the basic unit. You need to first figure out all the functions and then think about how to represent data.
The object-oriented languages focus on components that the user perceives, with objects as the basic unit. You figure out all the objects by putting all the data and operations that describe the user's interaction with the data.
Object-Oriented technology has many benefits:
Ease in software design as you could think in the problem space rather than the machine's bits and bytes. You are dealing with high-level concepts and abstractions. Ease in design leads to more productive software development.
Ease in software maintenance: object-oriented software are easier to understand, therefore easier to test, debug, and maintain.
Reusable software: you don't need to keep re-inventing the wheels and re-write the same functions for different situations. The fastest and safest way of developing a new application is to reuse existing codes – fully tested and proven codes.

Classes and Objects

A **class** is a collection of objects that have common properties, operations and behaviors. A class is a combination of state (data) and behavior (methods). In object-oriented languages, a class is a data type, and objects are instances of that data type. In other words, classes are prototypes from which objects are created.

For example, we may design a class Human, which is a collection of all humans in the world. Humans have state, such as height, weight, and hair color. They also have behavior, such as walking, talking, and eating. All of the state and behavior of a human is encapsulated (contained) within the class human.

An **object** is an instance of a class. Objects are units of abstraction. An object can communicate with other objects using messages. An object passes a message to another object, which results in the invocation of a method. Objects then perform the actions that are required to get a response from the system.

Real world objects all share two characteristics; they all have state and behavior. One-way to begin thinking in an object oriented way is to identify the state and behavior of real world objects. The complexity of objects can differ, some object have more states and more complex behaviors than other object. Compare the state and behavior of a television to the states and behaviors of a car.

Encapsulation[RGPV/June2009 (2)]

Definition: the ability of an object to hide its data and methods from the rest of the world – one of the fundamental principles of OOP. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but as long as the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.

Inheritance

Multiple classes may share some of the same characteristics, and have things in common with one another, but also may have certain properties that make them different. Object oriented programming languages allow classes to inherit commonly used state and ehavior from other classes.

Classes in Java occur in inheritance hierarchies. These hierarchies consist of parent child relationships among the classes. Inheritance is used to specialize a parent class, but creating a child class (example). Inheritance also allows designers to combine features of classes in a common parent.

Merits & demerits of OOPs

OOP stands for object oriented programming. It offers many benefits to both the developers and the users. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The primary advantages are:

Merits:

Through inheritance, we can eliminate redundant code and extend the use of existing classes.

We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

- The principle of data hiding helps the programmer to build secure programs that

cannot be invaded by code in other parts of the program.

- It is possible to have multiple objects to coexist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in an implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects make the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.
- Polymorphism can be implemented i.e. behavior of functions or operators or objects can be changed depending upon the operations.

Demerits:

- It requires more data protection.
- Inadequate for concurrent problems
- Inability to work with existing systems.
- Compile time and run time overhead.
- Unfamiliraity causes training overheads.

S.NO	RGPV QUESTIONS	Year	Marks
Q.2	What are the benefits and risks of object oriented development?	June2010, June-2009	10
Q.3	Write short notes on encapsulation.	June-2009	2
Q.3	Explain the feature of OOP with the help of example.	June2013	8

Unit-1/Lecture-2

Object-Oriented Programming vs. Procedural Programming [RGPV/Feb2010 (10)]

Programs are made up of modules, which are parts of a program that can be coded and tested separately, and then assembled to form a complete program. In procedural languages (i.e. C) these modules are procedures, where a procedure is a sequence of statements. In C for example, procedures are a sequence of imperative statements, such as assignments, tests, loops and invocations of sub procedures. These procedures are functions, which map arguments to return statements.

The design method used in procedural programming is called Top Down Design. This is where you start with a problem (procedure) and then systematically break the problem down into sub problems (sub procedures). This is called functional decomposition, which continues until a sub problem is straightforward enough to be solved by the corresponding sub procedure. The difficulties with this type of programming, is that software maintenance can be difficult and time consuming. When changes are made to the main procedure (top), those changes can cascade to the sub procedures of main, and the sub-sub procedures and so on, where the change may impact all procedures in the pyramid.

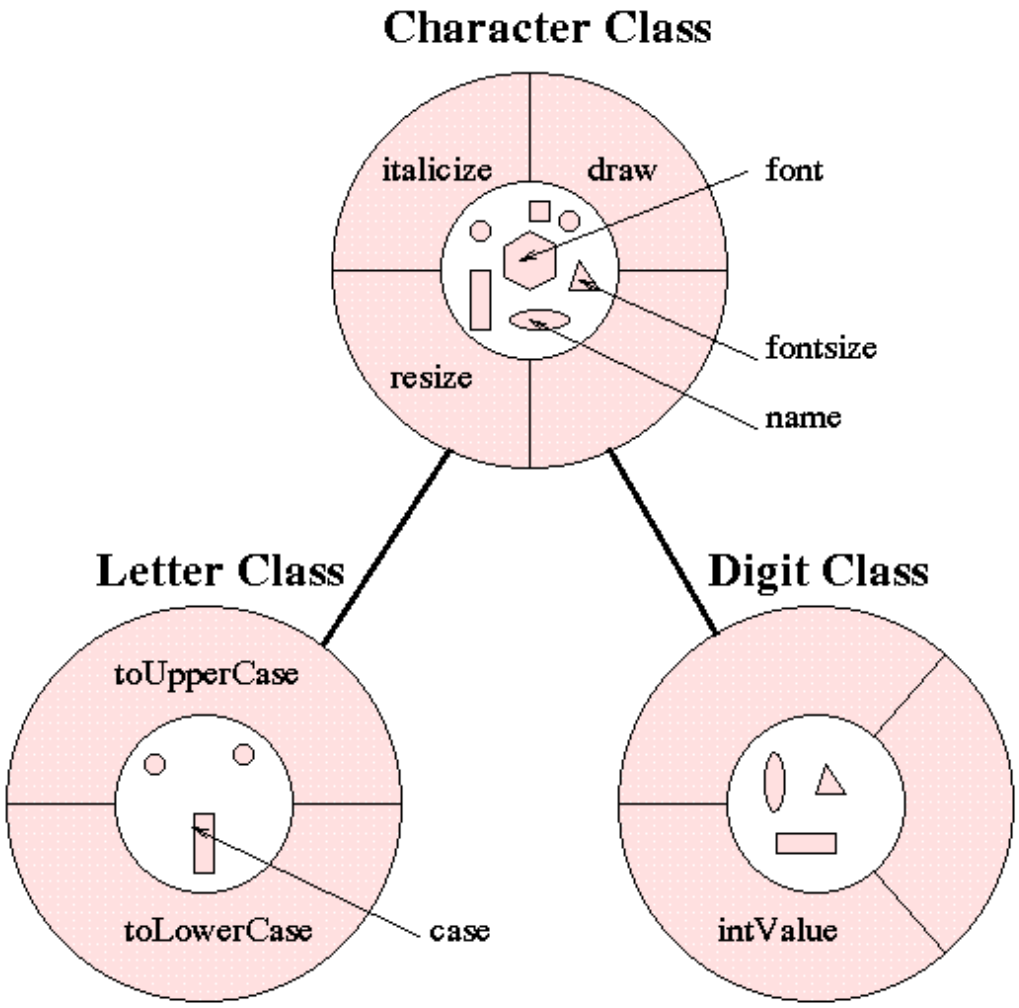
One alternative to procedural programming is object oriented programming. Object oriented programming is meant to address the difficulties with procedural programming. In object oriented programming, the main modules in a program are classes, rather than procedures. The object-oriented approach lets you create classes and objects that model real world objects.

Object Interaction: OOP: Class Hierarchy [RGPV/Feb2010(10),June2009(10)]

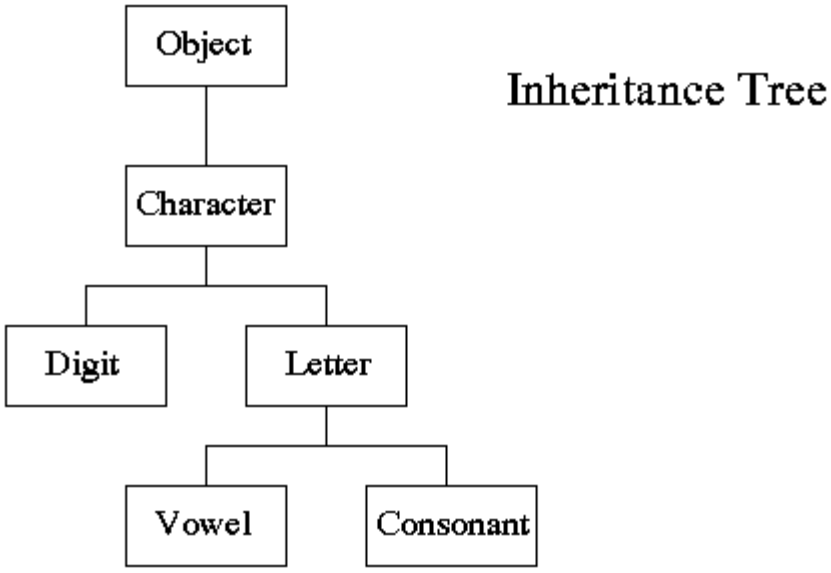
Object Interaction

In object-oriented programming, a class is a template that defines the state and behavior common to objects of a certain kind. A class can be defined in terms of other classes. For example, a truck and a racing car are both examples of a car. Another example is a letter and a digit being both a single character that can be drawn on the screen. In the latter example, the following terminology is used:

The letter class is a subclass of the character class; (alternative names: child class and derived class)
The character class is immediate superclass (or parent class) of the letter class;
The letter class extends the character class.
The third formulation expresses that a subclass inherits state (instance variables) and behaviour (methods) from its superclass (es). Letters and digits share the state (name, font, size, position) and behaviour (draw, resize,) defined for single characters.
The purpose of a subclass is to extend existing state and behavior: a letter has a case (upper and lower case, say stored in the instance variable letter Case) and methods for changing the case (toUpperCase, toLowerCase) in addition to the state that it already has as a character.
However, a digit does not have a case, so the methods toUpperCase, toLowerCase do not belong on the common level of the Character class. There are methods that are special to the digit class. For instance, a digit may be constructed from an integer value between 0 and 9, and conversely, the integer value of a digit may be the result of say the intValue method.
Subclasses can also override inherited behavior: if you had a colored character as subclass of the character class, you would override the definition of the draw method of the character class so that color is taken into account when drawing the character on the screen. This leads to what is called in OOP jargon polymorphism: the same message sent to different objects results in behavior that is dependent on the nature of the object receiving the message.
In graphical terms, the above character example may look as follows:



You are not limited to just one layer of inheritance: for example, the letter class can have on its turn the subclasses vowel and consonant.

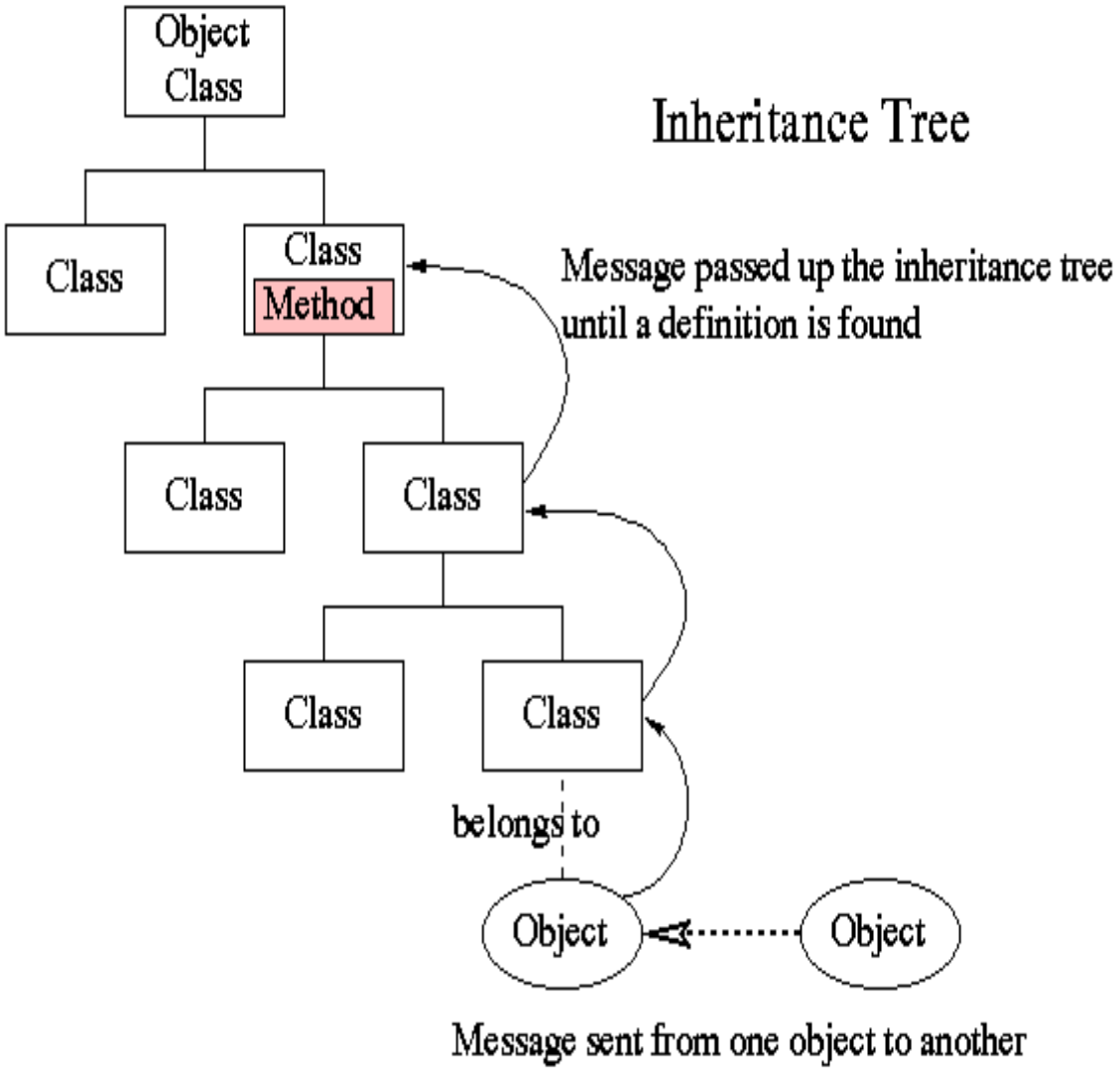


Inheritance Tree

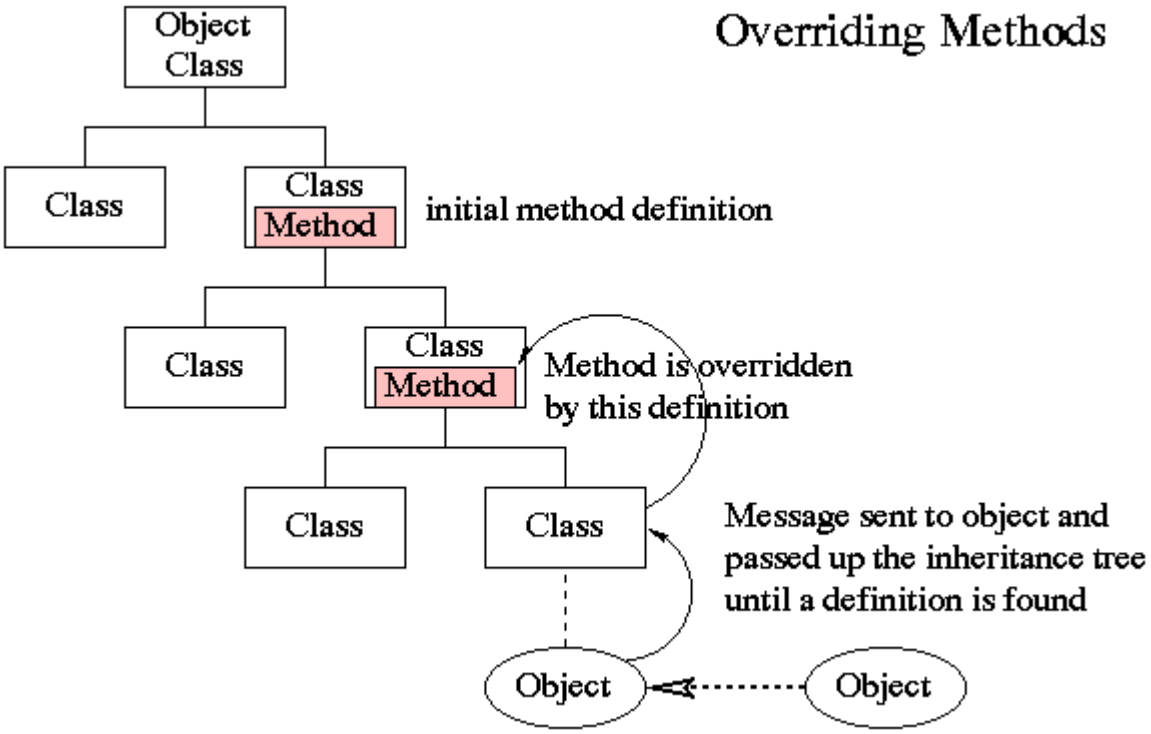
The classes form a class hierarchy, or inheritance tree, which can be as deep as needed. The

hierarchy of classes in Java has one root class, called Object, which is superclass of any class.

Instance variable and methods are inherited down through the levels. In general, the further down in the hierarchy a class appears, the more specialized its behavior. When a message is sent to an object, it is passed up the inheritance tree starting from the class of the receiving object until a definition is found for the method. This process is called upcasting. For instance, the method toString() is defined in the Object class. So every class automatically has this method. If you want that your particular toString() method looks differently, you can reimplement it in your class. In this way you can override a method in a given class by redefining it in a subclass.
In graphical terms, the inheritance tree and the message handling may look as follows:



The picture showing the overriding of methods, may look as follows:



S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Write the comparison between POP & OOPs.	RGPV Feb-2010	10
Q.2	What is meant by hierarchy of classes?	RGPV Feb-2010	10

Unit-1/Lecture-3

Static Member Function

You cannot have static and nonstatic member functions with the same names and the same number and type of arguments.
Like Static data members, you may access a static member function f() of a class A without using an object of class A.
A static member function does not have a this pointer. The following example demonstrates this:

```
#include <iostream>
using namespace std;
struct X {
private:
  int i;
  static int si;
public:
  void set_i(int arg) { i = arg; }
  static void set_si(int arg) { si = arg; }
  void print_i() {
    cout << "Value of i = " << i << endl;
    cout << "Again, value of i = " << this->i << endl;
  }
  static void print_si() {
    cout << "Value of si = " << si << endl;
//    cout << "Again, value of si = " << this->si << endl;
  }
};
int X::si = 77;      // Initialize static data member
int main() {
  X xobj;
  xobj.set_i(11);
  xobj.print_i();
  // static data members and functions belong to the class and
  // can be accessed without using an object of class X
  X::print_si();
  X::set_si(22);
  X::print_si();
}
```

The following is the output of the above example:

```
Value of i = 11Again, value of i = 11Value of si = 77
Value of si = 22
```

The compiler does not allow the member access operation this->si in function A::print_si() because this member function has been declared as static, and therefore does not have a this pointer.
You can call a static member function using the this pointer of a nonstatic member function. In the following example, the nonstatic member function printall()calls the static member

function f() using the this pointer:

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

class C {
  static void f() {
    cout << "Here is i: " << i << endl;
```

```
 }
 static int i;
 int j;
public:
 C(int firstj): j(firstj)
 { }
 void printall();
};
void C::printall() {
 cout << "Here is j: " << this->j << endl;
 this->f();
}

int C::i = 3;

int main() {

 C obj_C(0);
 obj_C.printall();
}
```

The following is the output of the above example:
Here is j: 0
Here is i: 3
A static member function cannot be declared with the keywords virtual, const, volatile, or const volatile.

A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared. Suppose a static member function f() is a member of class X. The static member function f() cannot access the nonstatic members X or the non static members of a base class of X.

Passing object parameter [RGPV/June2010(10),Dec,Feb2010 (10)]

Procedure to Pass Object to Function

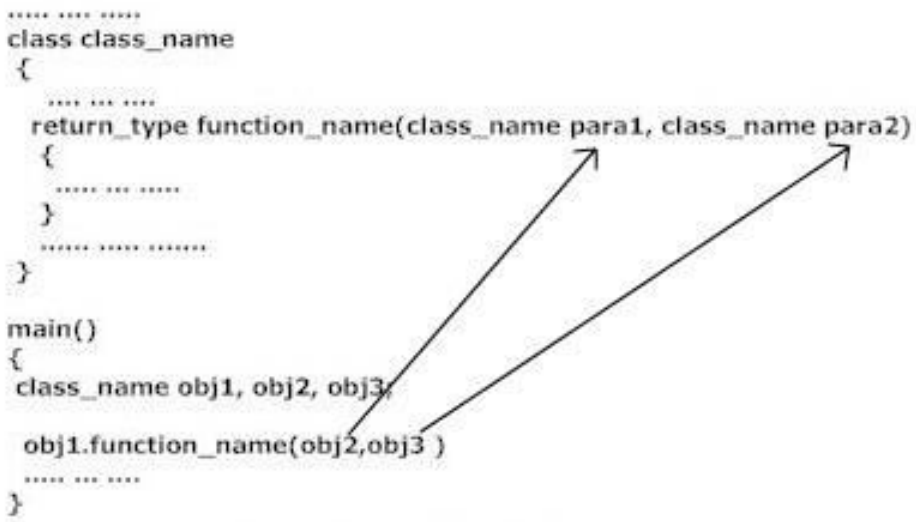


Figure: Passing Object to Function

Example to Pass Object to Function

C++ program to add two complex numbers by passing objects to function.

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

void Read()
     {
       cout<<"Enter real and imaginary number respectively:"<<endl;
       cin>>real>>imag;
     }
     void Add(Complex comp1,Complex comp2)
     {
        real=comp1.real+comp2.real;

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

        imag=comp1.imag+comp2.imag;
 /* Here, imag represents the imag data of object c3 because this function is called using code c3.add(c1,c2); */
     }
     void Display()
     {
        cout<<"Sum="<<real<<"+"<<imag<<"i";
     }
};
int main()
```

```
{
   Complex c1,c2,c3;
   c1.Read();
   c2.Read();
   c3.Add(c1,c2);
   c3.Display();
   return 0;
}
```

Output

Enter real and imaginary number respectively:

12

3

Enter real and imaginary number respectively:

2

6

Sum=14+9iReturning Object from Function

The syntax and

Procedure to return object is similar to that of returning structure from function.

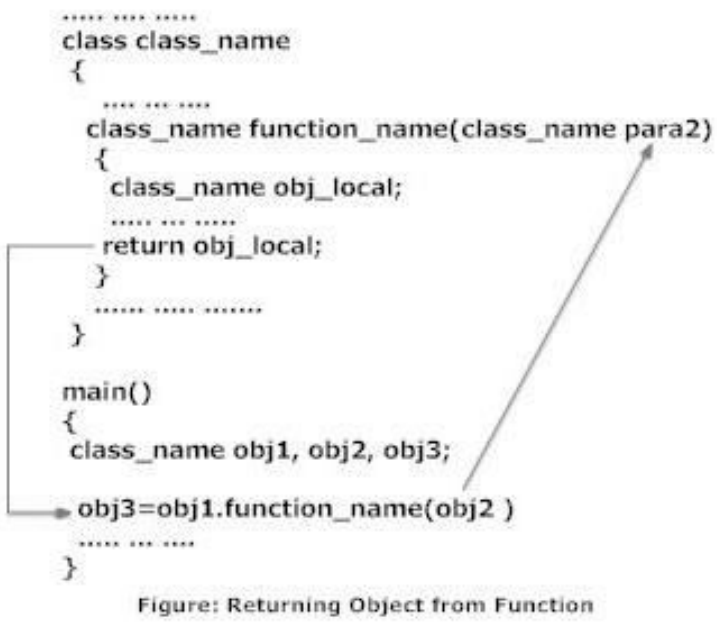


Figure: Returning Object from Function

Example to Return Object from Function

This program is the modification of above program displays exactly same output as above. But, in this program, object is return from function to perform this task.

```
#include <iostream>

using namespace std;

class Complex

{

   private:

      int real;

      int imag;

   public:

      Complex(): real(0), imag(0) { }

      void Read()

       {

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

          cin>>real>>imag;

       }

       Complex Add(Complex comp2)

       {

          Complex temp;

          temp.real=real+comp2.real;

/* Here, real represents the real data of object c1 because this function is called using
code c1.Add(c2) */

          temp.imag=imag+comp2.imag;

/* Here, imag represents the imag data of object c1 because this function is called using
code c1.Add(c2) */

          return temp;

       }

       void Display()

       {

          cout<<"Sum="<<real<<"+"<<imag<<"i";
```

```
    }
};
int main()
{
   Complex c1,c2,c3;
   c1.Read();
   c2.Read();
   c3=c1.Add(c2);
   c3.Display();
   return 0;
}
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Explain the different methods of passing object parameters.	June,Feb-2010	10
Q.2	Define a class to represent distance in felt and inch. Write a C++ program to add two distance object taken from keyboard. Use operator overloading.	June-2011	10

Unit-1/Lecture 4

Friend Function [RGPV/June2014 (2), June2013(12),Dec,June2010 (10)]

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 );

};
```

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne:

```
friend class ClassTwo;
```

Consider the following program:

```
#include <iostream>

 using namespace std;

 class Box

{

  double width;

public:

  friend void printWidth( Box box );

  void setWidth( double wid );

};
```

```
// Member function definition

void Box::setWidth( double wid )

{

   width = wid;

}

// Note: printWidth() is not a member function of any class.

Void printWidth( Box box )

{

  /* Because setWidth() is a friend of Box, it can

   directly access any member of this class */

  cout << “Width of box : “ << box.width <<endl;

}

 // Main function for the program

int main( )

{

  Box box;

   // set box width without member function

 box.setWidth(10.0);

   // Use friend function to print the wdith.

  printWidth( box );

   return 0;}
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Which one between friend function and multiple inheritance function will you prefer and why?	June 2010	10
Q.2	What are friend functions & friend classes? Write a normal function which adds objects of complex no class. Declare this normal function as friend of the complex class.	June2013	12

Unit-1/Lecture 5

Constructor & Destructor [RGPV/June2010(12),Feb2010(10),June2014 (10),June2013(8)]

Constructors are the special type of member function that initializes the object automatically when it is created Compiler identifies that the given member function is a constructor by its name and return type. Constructor has same name as that of class and it does not have any return type.

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

Working of Constructor

In the above pseudo code, temporary() is a constructor. When the object of class temporary is created, constructor is called and x is initialized to 5 and y is initialized to 5.5 automatically.

You can also initialize data member inside the constructor's function body as below. But, this method is not preferred.

```
Temporary(){
  x=5;
  y=5.5;
}
/* This method is not preferred. /*
```

Use of Constructor in C++

Suppose you are working on 100's of objects and the default value of a data member is 0. Initializing all objects manually will be very tedious. Instead, you can define a constructor which initializes that data member to 0. Then all you have to do is define object and constructor will initialize object automatically. These types of situation arises frequently while handling array of objects. Also, if you want to execute some codes immediately after object is created, you can place that code inside the body of constructor.

Constructor Example

```
/*Source Code to demonstrate the working of constructor in C++ Programming */

/* This program calculates the area of a rectangle and  displays it. */

#include <iostream>

using namespace std;
```

```
class Area
{
   private:
     int length;
     int breadth;

   public:
     Area(): length(5), breadth(2){ }   /* Constructor */
     void GetLength()
     {
        cout<<"Enter length and breadth respectively: ";
        cin>>length>>breadth;
     }
     int AreaCalculation() {  return (length*breadth);  }
     void DisplayArea(int temp)
     {
        cout<<"Area: "<<temp;
     }
};
int main()
{
   Area A1,A2;
   int temp;
   A1.GetLength();
   temp=A1.AreaCalculation();
   A1.DisplayArea(temp);
   cout<<endl<<"Default Area when value is not taken from user"<<endl;
```

```
   temp=A2.AreaCalculation();

   A2.DisplayArea(temp);

   return 0;

}
```

Explanation

In this program, a class of name Area is created to calculate the area of a rectangle. There are two data members' length and breadth. A constructor is defined which initializes length to 5 and breadth to 2. And, we have three additional member functions GetLength(), AreaCalculation() and DisplayArea() to get length from user, calculate the area and display the area respectively.

When, objects A1 and A2 are created then, the length and breadth of both objects are initialized to 5 and 2 respectively because of the constructor. Then the member function GetLength() is invoked which takes the value of length and breadth from user for object A1. Then, the area for the object A1 is calculated and stored in variable temp by calling AreaCalculation() function. And finally, the area of object A1 is displayed. For object A2, no data is asked from the user. So, the value of length will be 5 and breadth will be 2. Then, the area for A2 is calculated and displayed which is 10.

Output

```
Enter length and breadth respectively: 6
7
Area: 42
Default Area when value is not taken from user
Area: 10
```

Constructor Overloading

Constructor can be overloaded in similar way as function overloading. Overloaded constructors have same name(name of the class) but different number of argument passed. Depending upon the number and type of argument passed, specific constructor is called. Since, constructor are called when object is created. Argument to the constructor also should be passed while creating object. Here is the modification of above program to demonstrate the working of overloaded constructors.

```
/* Source Code to demonstrate the working of overloaded constructors */

#include <iostream>

using namespace std;

class Area

{

   private:

      int length;

      int breadth;
```

```
   public:
     Area(): length(5), breadth(2){ }        // Constructor without no argument
     Area(int l, int b): length(l), breadth(b){ } // Constructor with two argument
     void GetLength()
     {
        cout<<"Enter length and breadth respectively: ";
        cin>>length>>breadth;
     }
     int AreaCalculation() {  return (length*breadth);  }
     void DisplayArea(int temp)
     {
        cout<<"Area: "<<temp<<endl;
     }
};
int main()
{
   Area A1,A2(2,1);
   int temp;
   cout<<"Default Area when no argument is passed."<<endl;
   temp=A1.AreaCalculation();
   A1.DisplayArea(temp);
   cout<<"Area when (2,1) is passed as arguement."<<endl;
   temp=A2.AreaCalculation();
   A2.DisplayArea(temp);
   return 0;
}
```

Explanation of Overloaded Constructors

For object A1, no argument is passed. Thus, the constructor with no argument is invoked which initializeslength to 5 and breadth to 2. Hence, the area of object A1 will be 10. For object A2, 2 and 1 is passed as argument. Thus, the constructor with two argument is called which initializes length to l(2 in this case) and breadth to b(1 in this case.). Hence the area of object A2 will be 2.

Output

Default Area when no argument is passed.
Area: 10
Area when (2,1) is passed as argument.
Area: 2

Default Copy Constructor

A object can be initialized with another object of same type. Let us suppose the above program. If you want to initialize a object A3 so that it contains same value as A2. Then, this can be performed as:

```
….
int main() {
  Area A1,A2(2,1);
  Area A3(A2);    /* Copies the content of A2 to A3 */
   OR,
  Area A3=A2;     /* Copies the content of A2 to A3 */
}
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What is the constructor? Explain different type of constructors.	Feb2010, June2014	10
Q.2	Explain why do we need to use constructors? Explain a copy constructor with an example.	June2013	8

Unit-1/Lecture 6

Destructor [RGPV/June2010(12),Jun e2009 (10)]

Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.
A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example:

```
class X {
public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared const, volatile, const volatile orstatic. A destructor can be declared virtual or pure virtual.
If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.
The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for A:

```
A::~A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class A before defining the implicitly declared destructor of A
A destructor of a class A is trivial if all the following are true:
It is implicitly defined

All the direct base classes of A have trivial destructors
The classes of all the nonstatic data members of A have trivial destructors
If any of the above are false, then the destructor is nontrivial.
A union member cannot be of a class type that has a nontrivial destructor.
Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class A or a member of A has a destructor, and a class derived from A does not declare a destructor, a default destructor is generated.

The default destructor calls the destructors of the base class and members of the derived class.
The destructors of base classes and members are called in the reverse order of the completion of their constructor:

The destructor for a class object is called before destructors for members and bases are called.
Destructors for nonstatic members are called before destructors for base classes are called.
Destructors for nonvirtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block.
Destructors are implicitly called when an automatic object (a local object that has been declared auto or register, or not declared as static or extern) or temporary object passes out of scope. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the delete operator for objects created with the new operator.

For example:

```
#include <string>

class Y {
private:
  char * string;
  int number;
public:
  // Constructor
  Y(const char*, int);
  // Destructor
  ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
  string = strcpy(new char[strlen(n) + 1 ], n);
  number = a;
}

int main () {
  // Create and initialize
  // object of class Y
  Y yobj = Y("somestring", 10);

  // …

  // Destructor ~Y is called before
  // control returns from main()
}
```

You can use a destructor explicitly to destroy objects, although this practice is not recommended. However to destroy an object created with the placement newoperator, you can explicitly call the object's destructor. The following example demonstrates this:

```
#include <new>
#include <iostream>
using namespace std;
class A {
  public:
    A() { cout << "A::A()" << endl; }
    ~A() { cout << "A::~A()" << endl; }
};
int main () {
  char* p = new char[sizeof(A)];
  A* ap = new (p) A;
  ap->A::~A();
  delete [] p;
}
```

The statement A* ap = new (p) A dynamically creates a new object of type A not in the free store but in the memory allocated by p. The statement delete [] p will delete the storage allocated by p, but the run time will still believe that the object pointed to by ap still exists until you explicitly call

the destructor of A(with the statement ap->A::~A()).

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Write a program that includes all varities of constructors in it & also display the use of each one.	RGPV Dec-2007	10
Q.2	What are the constructors & destructors? Explain copy constructor & parameterized constructor with example.	June2010	12

Unit-1/Lecture-7

Characteristics of Constructor [RGPV/Dec2010 (10)]

Use the Features window to change the features of a constructor, including its arguments and initialization code. Double-click the constructor in the IBM® Rational® Rhapsody® browser to open its Features window.

About this task

On this General tab, you define the general features for a constructor through the various controls on the tab. Notice that the signature for the constructor is displayed at the top of the General tab of the Features window.

- In theName field you specify the name of the constructor. The default name is the name of the class it creates. To enter a detailed description of the constructor, use the Description tab.
- If the Name field is inaccessible, click the L button to open the Name and Label window to change the name, if any.
- In the Stereotype list you specify the stereotype of the attribute, if any.

o To select from a list of current stereotypes in the project, click the Select Stereotype button .

o To sort the order of the selected stereotypes, click the Change Stereotype Order button .

Note: The COM stereotypes are constructive; that is, they affect code generation.

- In the Visibility list you specify the visibility of the reception (Public, Protected, or Private), if available. The default value is Public.
- In the Initializer field you enter code if you want to initialize class attributes or super classes in the constructor initializer. To access the text editor, click the Ellipses button .

For example, to initialize a class attribute called a to 5, type the following code:

a(5)

- Note: In C++, this assignment is generated into the following code in the class implementation file to initialize the data member in the constructor initializer rather than in the constructor body:

```
//---------------------------------------------------------
// A.cpp
//---------------------------------------------------------
A::A() : a(5) {
  //#[operation A()
  //#]
};
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What are Global constructor and destructor?	Dec-2010	10

Unit1/Lecture-8

Method Lookup [RGPV/June2009(7)]

Scope and Visibility
Consider this little program.

```
int i;

int main() {

  int i;

  i=9;

}
```

There are two is - one inside main and one outside. Which i is set to 9? By default the "nearest" one is, the one inside main (we'll come back to what is meant by "nearest" later in this article). The other i isn't visible (it's covered up by main's i) but it's "in scope" so it can be accessed. C++'s scope resolution operator (::) is used when specifying where to look for a variable in such circumstances. In this case the scope to be searched is the one that encloses the current one. Using ::i accesses the variable there.

Namespaces
A namespace defines a scope. It's like a context which determines the meaning of a symbol. Just as the meaning of "Cambridge" will change depending on whether you're in a UK or US context, so the variable accessed in a C++ program by the symbol "i" will depend on the context, as we've seen above.
Some simple languages only have one namespace - all function names, variable names, etc belong to the same context. Some other languages have several independent namespaces (one for variable names, one for function names, etc) making it possible to have both a variable and function with the same name, but the number and role of these namespaces are fixed.
C++ has some fixed namespaces, but it also has named namespaces and lets users create new namespaces. It also offers control over which of these named namespaces will be used when the meaning of a symbol is required.
In ANSI C++ the standard library facilities (like cout, string, etc) are kept inside the namespace called std, which by default isn't consulted. Namespaces are created and entities put into them by using namespace. E.g.

```
namespace test {

  int i;

};
```

creates a namespace called test (if one hasn't already been created) and puts i into it. Then test::i (the same notation that you'd use were test an object) will access the i variable. The command using namespace test will make available all the things inside test so that test:: isn't necessary. Let's see this in action

```
namespace test {

  int i;

};

int i;

int main() {

  i=9;
```

}In this program main can only see one i, the other is hidden inside the test namespace. The latter is in scope and can be accessed using test::i. What about the following though?

```
namespace test {

  int i;

};

using namespace test; //this line's been added

int i;

int main() {

  i=9;
```

}Here both is are visible from main. In fact they clash so this program won't compile. My compiler says The declarations "int i" and "int test::i" are both visible and neither is preferred under the name lookup rules.

Classes

Functions can be in a class or free standing. Whenever a function call is processed there may be several available functions in scope with the same name. C++ performs a look-up using a well-defined strategy in order to decide which function to call. Sometimes it can't decide which function is best to call, in which case the compiler complains about an "ambiguous call". Usually there are no problems - the programmer and compiler agree on the best option. In the following for example, there's a free-standing fun(float f) as well as one in the class. The fun(f) call in main calls the free-standing one. The call from inside the class calls the "nearer" one inside the class.

```
void fun(float f){};

class classy {

public:

   void fun(float f){};

   void fun2(float f){fun(f);};

};

int main() {

   float f;

   fun(f);

   classy c;

   c.fun2(f);
}
```

Functions and Overloading [RGPV/June2010 (10)]

Functions add a complication because it's possible to have many functions with the same name all visible without clashing as long as they take different arguments. In the following example fun is overloaded - which isn't a problem!

```
void fun(int i) {};

void fun(int i, int j) {};

int main () {

  fun(3);

  fun(5,7);

}
```

The following's perhaps a little trickier.

```
void fun(float f) {};

int main () {

  int i=3;

  fun(i);

}
```

There's no function called fun that takes an integer so fun(float f) is called without complaint. In the next example fun(int) is supplied, so this will be the prefered candidate.

```
void fun(float f) {};

void fun(int f) {}; // added line

int main () {

  int i=3;

  fun(i);

}
```

None of that should be too disturbing, but what about the following? Which f function is called? The first might look like the closest match, but it's the second that's called, because the char* to bool conversion is built into C/C++, and matches using standard conversions take precedence over user-defined ones.

```
#include <iostream>

#include <string>

using namespace std;

void f(string a, string b, bool c = false) {

 cout << "called 3 arg function" << endl;

};

void f(string a, bool c = false) {
```

```
 cout << "called 2 arg function" << endl;
};
int main()
{
 f("one", "two");
}
```

Inheritance

Often you'll need to add extra functionality to an existing class. C++ provides a mechanism to build new classes from old ones

```
 class Base {
 public:
   int value1;
 };
 class More : public Base {
 public:
   int value2;
 };
 int main() {
   Base b;
   b.value1=7;
   More m;
   m.value1=7;
   m.value2=9;
 }
```

Here More inherits the members of Base so m has 2 members - value1 and value2. Members can be functions or variables. The following, which uses functions where the previous example used variables, works ok.

```
 class Base {
 public:
   void fun1(){};
 };
 class More : public Base {
```

```
public:
  void fun2(){};
};
int main() {
  Base b;
  b.fun1();
  More m;
  m.fun1();
  m.fun2();
}
```

Now we come to our first "interesting program". Suppose we give both functions the same name but different arguments. What happens?

```
class Base {
public:
  void fun1(){};
};
class More : public Base {
public:
  void fun1(int i){};
};
int main() {
  Base b;
  b.fun1();
  More m;
  m.fun1();
  m.fun1(5);
}
```

b.fun1() poses no problem. One might expect m.fun1() to call the Base's function and m.fun1(5) to callMore's function (i.e. expect fun1 to be overloaded). In fact the code doesn't compile - void fun1() is masked by void fun1(int). With an extra line it will compile

```
class Base {
public:
```

```
  void fun1(){};
};
class More : public Base {
public:
using Base::fun1; // added line
  void fun1(int i){};
};
int main() {
  Base b;
  b.fun1();
  More m;
  m.fun1();
  m.fun1(5);
}
```

Function Lookup

And here's another surprising situation. The following compiles, but why?

```
namespace test {
  class T {};
  void f(T){};
};
test::T parm;
int main() {
  f(parm);  // OK: calls test::f
}
```

Here we have a namespace called test inside which there's a class T and a function that takes one argument of type T. Outside the namespace a variable called parm is created. Note that the test:: is needed to get hold of the T within this namespace. In main a function f is called. Even though there's no test::before the function name, and no previous using namespace test line, the program compiles.

This is a situation where Koenig lookup (also called Argument-Dependent name Lookup - ADL) is used. If you supply a function argument that isn't a built-in type (here parm, of type test::T), then to find the function name the compiler is required to look in the namespace (in this case test) that contains the argument's type as well as in the usual places.

Ordinary name look-up searches for qualified names in the nearest enclosing scope where the name is used, and if not found, the look-up proceeds in successively enclosing scope until the name is found. Even if the name is not appropriate for the given use, the look-up search proceeds no further through the hierarchies. At this point ADL finishes the job.

This explains why the example in the previous section failed whereas the one in this section

succeeded, but the look-up mechanism seems to be defeating one of the purposes of namespaces - the ability to hide entities. However, there's a case for saying that once T is brought out into the open, then associated routines should become visible too. There are also pragmatic and safety reasons why ADL is used.
Here's a simple program

```
#include <iostream>

#include <string>

int main() {

  std::string hello = "Hello, world";

  std::cout << hello;

}
```

This is analogous to the previous program: std::string is like the test::T of the earlier example.operator<< is a free function that the compiler can only find using ADL (operator<< can't be a member function because it requires a stream as the left-hand argument). Without ADL the final line would be awkward to express.
Here's another simple fragment

```
char x;

void f() {

  int x;

  x = 'a';

}
```

C/C++ has always set the function's x in this situation although the other x is a closer match type-wise. ADL conforms with this traditional behaviour.
Here's a situation involving classes. In this fragment, the g function calls the class's f routine.

```
class X {

  int f(int);

  int g() { f('a'); }

}
```

But suppose that during program development a global function f(char) were added - what f function should g call then? It would be an unpleasant shock if the global function were called - you don't want the internals of classes to be quite so vulnerable to external changes.
A final note from Victor Bazarov on comp.lang.c++ - ADL applies only to function names, not variables. The only other thing that has arguments in C++ is templates. But ADL doesn't apply to them. In this example

```
namespace test {

  enum foo { f };

  template<foo f> class bar {};
```

```
}

int main() {

  bar<test::f> barf;
```

}main's bar isn't going to be looked up in test even though its argument is fully qualified and found in the test namespace.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Distinguish between overloading & overriding.	RGPV June-2010	10

Unit-1/Lecture-9

Inheritance [RGPV/Feb2010 (10)], [RGPV/June2011 (10)]

The mechanism that allows us to extend the definition of a class without making any physical changes to the existing class is inheritance.
Inheritance lets you create new classes from existing class. Any new class that you create from an existing class is called **derived class**; existing class is called **base class**.
The inheritance relationship enables a derived class to inherit features from its base class. Furthermore, the derived class can add new features of its own. Therefore, rather than create completely new classes from scratch, you can take advantage of inheritance and reduce software complexity.

Forms of Inheritance

Single Inheritance: It is the inheritance hierarchy wherein one derived class inherits from one base class.

Multiple Inheritance: It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es).

Hierarchical Inheritance: It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.

Multilevel Inheritance: It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

Hybrid Inheritance: The inheritance hierarchy that reflects any legal combination of other four types of inheritance.
In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format :

```
class derived_class: memberAccessSpecifier base_class
{
...
};
```

Where derived_class is the name of the derived class and base_class is the name of the class on which it is based. The member Access Specifier may be public, protected or private. This access specifier describes the access level for the members that are inherited from the base class.

Member Access Specifier	**How Members of the Base Class Appear in the Derived Class**
Private	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become private members of the derived class.
	Public members of the base class become private members of the derived class.
Protected	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become protected members of the derived class.
Public	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected

	members of the derived class.
	Public members of the base class become public members of the derived class.

In principle, a derived class inherits every member of a base class except constructor and destructor. It means private members are also become members of derived class. But they are inaccessible by the members of derived class.

Following example further explains concept of inheritance :

```
class Shape
{
protected:
        float width, height;
public:
        void set_data (float a, float b)
        {
                width = a;
                height = b;
        }
};

class Rectangle: public Shape
{
public:
        float area ()
        {
                return (width * height);
        }
};

class Triangle: public Shape
{
public:
        float area ()
        {
                return (width * height / 2);
        }
};

int main ()
{
        Rectangle rect;
        Triangle tri;
        rect.set_data (5,3);
        tri.set_data (2,5);
        cout << rect.area() << endl;
        cout << tri.area() << endl;
        return 0;
}
```

output :

```
15
5
```

The object of the class Rectangle contains :
width, height inherited from Shape becomes the protected member of Rectangle. set_data() inherited from Shape becomes the public member of Rectangle area is Rectangle's own public member.

The object of the class Triangle contains :
width, height inherited from Shape becomes the protected member of Triangle.
set_data() inherited from Shape becomes the public member of Triangle
area is Triangle's own public member set_data () and area() are public members of derived class and can be accessed from outside class i.e. from main().

Polymorphism[RGPV/June2009 (10)]

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: [RGPV/June2013(5),June2010(10)]

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.

S.NO	RGPV QUESTIONS	Year	Marks
Q.2	Explain the multiple inheritances.	June2011,June 2010	10
Q.3	What is pure virtual function? Why do you need them and how they are defined?	Dec-2012, June2010	10
Q.4	What is virtual base class? Explain with suitable example.	June2013	5