Class, Object and Message	**Defining Class and Declaring object**	**Constructors and destructors**	**Object creation in C++, Dynamic memory allocation**	**Overloading Constructors**
Default constructor	**Pointers to classes**	**Message**		

Class, Object and Message

- **Class** is a user defined data type, which holds its **own data members** and **member functions**, which can be accessed and used by creating instance of that class.

- Collection of operations defines the behavior of the class and the data variables are used to maintain the internal state of the object.

- An ***object*** is an **instantiation** of a class (representative of class). Instance variable means an internal variable maintained by the instance.
- In terms of variables, a class would be the type, and an object would be the variable.
- Each instance has its own collection of instance variables. These values should not be changed directly by the clients, but rather should be changed only by methods associated with the class. From the outside, clients can see only the behavior of the objects.

An Object in C++ has three characteristics:

State: Represents data (value) of an object.
Behavior: Represents the behavior (functionality) of an object such as deposit, withdraw etc.
Identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user.

Difference between Class and Object

	Class	Object
1	Class is a container which collection of variables and methods.	object is a instance of class
2	No memory is allocated at the time of declaration	Sufficient memory space will be allocated for all the variables of class at the time of declaration.
3	One class definition should exist only once in the program.	

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

keyword user-defined name

```
class ClassName
{  Access specifier:          //can be private,public or protected
   Data members;              // Variables to be used
   Member Functions() { }     //Methods to access data members
};                            // Class name ends with a semicolon
```

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.

Declaring Objects: To use the data and access functions defined in the class, you need to create objects using Syntax: **ClassName ObjectName;**

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName() .

Encapsulation

The principle of hiding the used data structure of a class (type) and to only expose it's methods to client through a well-defined interface is known as encapsulation.

Classes are generally declared using the keyword **class**, with the following format:

```
class class_name {
    access_specifier_1:
        member1;
    access_specifier_2:
        member2;
    ...
} object_names;
```

- Member is either data structure (indicating state) or behavior (method)
- class_name is a valid identifier for the class (type)
- object_names is an **optional** list of names for objects of this class.
- The body of the declaration can contain members, that can be either **data** or **method** declarations, and optionally access specifiers.

An **access specifier** helps to implement encapsulation and it is one of the following three keywords:

- **private** members of a class are accessible only from within other members of the same class or from their **friends**.
- **protected** members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- **public** members are accessible from anywhere where the object is visible.
- By default, all members of a class declared with the class keyword have **private** access for all its members (if nothing specified means private access).

A class may be thought of consists of two parts :

1. **Interface** : The interface refers to the information that a client must have in order to use the facilities of class (it is so called because it describes how the object is interfaced to the world at large). At a minimum the interface must contain :

    - The name of the **public** member function of the class and its **Prototype**
    - The purpose of each member function

    As in C, the header file (extension .h) describes the interface of the class. In fact, it shows everything except the functions that are defined in another file. An interface file can contain descriptions of **more than one related class**.

Defining an **interface** stored in a separate file **CRectangle.h**

```
class CRectangle {
  int x, y;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

Prototyp

- Object of the CRectangleclass is rect.
- Members x and member y has private access.
- Member functions with public access: set_values() and area()
- We show only their declaration, not their definition.
- In declaration of method parameter type is only provided, parameter name is optional.
- This style of function declaration is known as **prototype** (ANSI Standard)

2. **Implementation** : Actual definition of member functions of the interface (header file) can be placed in a different file known as **implementation** of the class. The implementation describes how the responsibility promised by the interface is achieved. In that case like C we have to include the header file in the implementation file (.cpp) as

    #include "Crectangle.h"

    When interface contains declaration of more than one class then conditional inclusion facility may be required.

```
#  ifndef CARDH
#  define CARDH
#  endif
```

First time the file **card.h** included, the symbol **cardh** is not defined and thus the **ifndef** statement is satisfied. On all subsequent readings of the file the symbol will be known and the file will be skipped.

Note: Prototype contains an argument list containing types of parameters and names are optional.

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

int CRectangle:: area () {return (x*y);}

void CRectangle::set_values (int a, int b)
{
    x = a;   y = b;
}
int main () {
      CRectangle rect;
      rect.set_values (3,4);
      cout << "area: " << rect.area();
      return 0;  }
```

Non-inline function

operator of scope (::, two colons) - used to define a member of a class from outside the class definition itself. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

Here is the complete example of class CRectangle where we have not use separate interface file (.h)

```
// classes example
#include <iostream>
using namespace std;
class CRectangle {
     int x, y;
     public:
         void set_values (int,int);
         int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
    x = a;   y = b;
}
int main () {
      CRectangle rect;
      rect.set_values (3,4);
      cout << "area: " << rect.area();
      return 0;
}
area: 12
```

- Definition of the member function area() included directly within the definition of the CRectangle. Compiler will automatically consider it as **inline member function**.
- Whereas **set_values()** has only its prototype declared within the class, but its definition is outside it. It is treated as normal (not-inline) function.
- Generally only the prototype should be included in the class and later define it -no difference in behavior in both cases.

In the function set_values() , we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

Any of the **public** members of the object **rect** can be accessed just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example: rect.set_values (3,4); myarea = rect.area();

We can declare several objects of a class. For example, following with the previous example of class CRectangle, we could have declared the object **rectb** in addition to the object rect:

```
// example: one class, two objects
#include <iostream>
using namespace std;
class CRectangle {
    int x, y;
    public:
       void set_values (int,int);
       int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
       x = a;     y = b;
}
```

```
int main () {
    CRectangle rect, rectb;
    rect.set_values (3,4);
    rectb.set_values (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
   return 0;
}
rect area: 12
rectb area: 30
```

Here two instances or objects: rect and rectb are created. Although each one of them has its own **member variables, all objects share a single set of member functions.**

Constructors and destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation. This initialization is needed to avoid returning unexpected values during their execution.

Constructor in a class is a special function called automatically after object creation for initialization. This constructor function must have the **same name** as the class, and cannot have any **return type**; not even **void**.

```
// example: class constructor
#include <iostream>
using namespace std;
class CRectangle {
    int width, height;
    public:
       CRectangle (int,int);
       int area () {return (width*height);}
};
CRectangle::CRectangle (int a, int b) {    width = a;     height = b;  }
```

```
int main () {
   CRectangle rect (3,4);     CRectangle rectb (5,6);
   cout << "rect area: " << rect.area() << endl;
   cout << "rectb area: " << rectb.area() << endl;
   return 0;
}
rect area: 12
rectb area: 30
```

Constructors cannot be called **explicitly** as if they were regular member functions. They are only executed when a new object of that class is created.

Following example shows three overloaded constructors to provide more than one style of initialization.

Example of constructor :

```
class Complex {
   public :
     //  Constructor
     Complex ();
     Complex (double);
     Complex (double, double);

     //  Operations
     ........
   private :
     // Data area
     double  realPart;
     double  imaginaryPart;   };
```

Implementation of constructors on the same .cpp file :

```
Complex :: Complex()
{   //  initialize both parts to zero
    realPart = 0.0;    imaginaryPart = 0.0  }

Complex :: Complex(double rp)
{  //  initialize real part with some value
    realPart = rp;    imaginaryPart = 0.0  }
```

```
//  Another way of implementation by providing initializers in
the function heading (initialization list)

Complex :: Complex(double rp) : realPart(rp) ,
                                imaginaryPart(0.0)
{  //  No further initializationnecessary  }
```

Each initialize clause simply names an instance variable and in parentheses lists the value to be used for initialization

A series of declaration invoking above constructors might as follows :

```
Complex  pi = 3.14159;    Complex  e (2.71);
Complex  I (0,1);
```

First two will invoke the constructor Complex(double), third one will use Complex(double, double)

```
Complex  *c;    c = new Complex(3.14159, -1.0);     //  Create object using new
```

Example for initializing const data field by initialize :

Values can be declared as immutable (not allowed to change) in C++ using **const** keyword. Instance variables declared as constant must be initialized with an initialize clause in the constructor header, as a constant value cannot be the target of an assignment statement.

```
class queen {
public :
   //  Constructor
   queen (int, queen *);
........
private :
   //  Data fields
```

```
queen:: queen (int col,  queen *ngh )
        :  column(col),  neighbor(ngh)
{   row = 1;  }
```

```
  int row;
  const  int column;
  queen * neighbor;
……
}
```

Destructor fulfills the opposite functionality.

- This function is the constructor's counterpart in the sense that it is invoked when an object ceases to exist.
- A destructor is usually called automatically, but that's not always true.
- The destructors of dynamically allocated objects are not automatically activated, but in addition to that: when a program is interrupted by an exit call, only the destructors of already initialized global objects are called.

Destructors obey the following syntactical requirements:

- a destructor's name is equal to its class name prefixed by a tilde;
- a destructor has no arguments;
- a destructor has no return value.

```
// example on constructors and destructors
#include <iostream>
using namespace std;
class CRectangle {
    int *width, *height;
    public:
       CRectangle (int,int);
       ~CRectangle ();
      int area () {return (*width * *height);}
};
int main () {
   CRectangle rect (3,4), rectb (5,6);
   cout << "rect area: " << rect.area() << endl;
   cout << "rectb area: " << rectb.area() << endl;
   return 0;
}
```

```
CRectangle::CRectangle (int a, int b) {
    width = new int;     height = new int;
    *width = a;          *height = b;
}
CRectangle::~CRectangle () {
    delete width;    delete height;
}
```

rect area: 12
rectb area: 30

Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Constructor that matches the arguments passed on the object declaration will be called automatically.

```
// overloading class constructors
#include <iostream>
```

```
using namespace std;
class CRectangle {
    int width, height;
    public:
      CRectangle ();
      CRectangle (int,int);
      int area (void)
            {return (width*height);}
};
CRectangle::CRectangle () {      width = 5;   height = 5;  }
CRectangle::CRectangle (int a, int b) {    width = a;   height = b;    }
```

```
int main () {
    CRectangle rect (3,4);       CRectangle rectb;
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
rect area: 12         rectb area: 25
```

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
CRectangle rectb; // right        CRectangle rectb(); // wrong!
```

Default constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```
class CExample {
   public:
     int a,b,c;
     void multiply (int n, int m) { a=n; b=m; c=a*b; };
};
```

The compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply declaring them without any arguments: CExample ex;

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

```
class CExample {
     public:
        int a,b,c;
        CExample (int n, int m) { a=n; b=m; };
        void multiply () { c=a*b; };
};
```

So the following object declaration would be correct:

CExample ex (2,3); But, CExample ex; Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.

But the compiler not only creates a default constructor for you if you do not specify your own. It provides **three special member functions** in total that are implicitly declared if you do not declare your own. These are the ***copy constructor***, the ***copy assignment operator***, and the **default destructor**.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object. For CExample, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (const CExample& rv) {     a=rv.a; b=rv.b; c=rv.c;     }
```

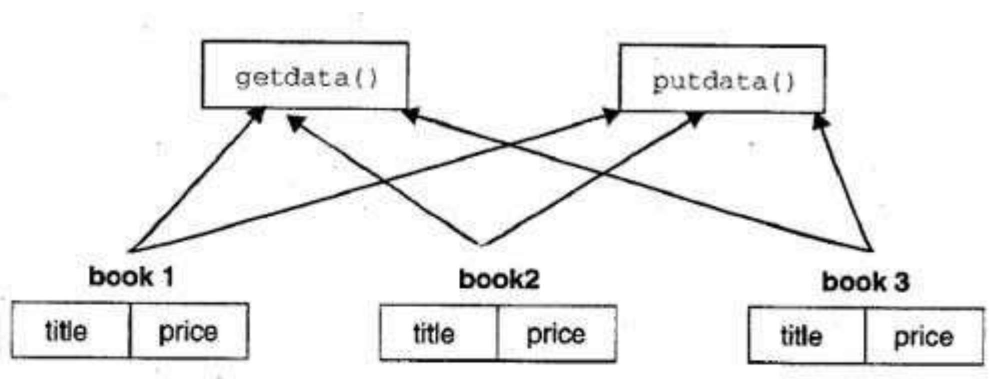
Therefore, the two following object declarations would be correct:

```
CExample ex (2,3);            CExample ex2 (ex); // copy constructor (data copied from ex)
```

Memory Allocation for Objects

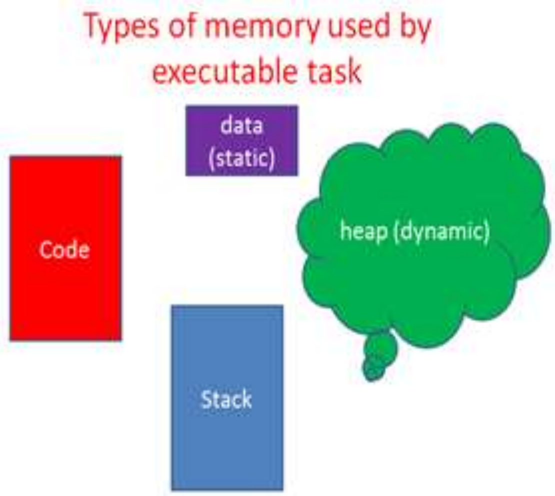
Before using a member of a class, it is necessary to allocate the required memory space to that member. The way the memory space for data members and member functions is allocated is different regardless of the fact that both data members and member functions belong to the same class.

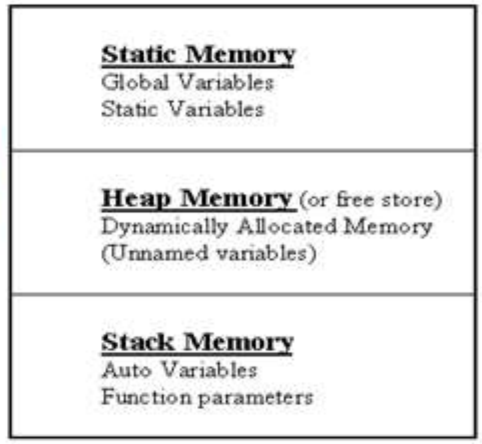
- The memory space is allocated to the data members of a class only when an object of the class is declared, and not when the data members are declared inside the class. Every object declared for the class has an individual copy of all the data members.
- On the other hand, the memory space for the member functions is allocated only once when the class is defined. In other words, there is only a single copy of each member function, which is shared among all the objects. For instance, the three objects, namely, book1, book2 and book3 of the class book have individual copies of the data members title and price. However, there is only one copy of the member functions getdata () and putdata () that is shared by all the three objects



Memory Allocation for the Objects of the Class book

Objects may be created in stack or dynamically created in heap using new operator.





There are a few interesting high level details that are worth knowing about the heap and the stack :

- The stack is a portion of memory (lives in RAM) that is allocated to a thread when it is created (each thread has it's own stack).
- The stack is not very smart. It grows and shrinks similar to a stack (The data structure); The stack space is allocated in RAM when the thread is created and it doesn't change it's size.
- The heap is a portion of memory that is allocated to the process when it is started.
- You can add information to the heap on demand and it can be accessed by all threads in the same process.
- If you try to store more information in the heap than it can hold, the Operating System will look for free space in RAM and give it to you.
- The stack is generally faster because finding available memory in RAM (allocating memory) can be an expensive operation. The stack only needs to do this when a thread is created, because it has a predefined size and it grows and shrinks by moving a pointer up or down the stack. The heap grows as the application requires more data, and memory allocations happen a lot more often.

When a C++ object is created, two events occur:

1. Storage is allocated for the object.
2. The constructor is called to initialize that storage.

Lifetime and visibility of different memory types are :

- **static variable (class)**
  lifetime = program runtime (1)
  visibility = determined by access modifiers
  (private/protected/public)
- **static variable (global scope)**
  lifetime = program runtime (1)
  visibility = the compilation unit it is instantiated in (2)

(1) more exactly: from initialization until deinitialization of the compilation unit (i.e. C / C++ file). Order of initialization of compilation units is not defined by the standard.

(2) Beware: if you instantiate a static variable in a header, each compilation unit gets its own copy.

- **heap variable**
  lifetime = defined by you (new to delete)
  visibility = defined by you (whatever you assign the pointer to)

- **stack variable**
  visibility = from declaration until scope is exited
  lifetime = from declaration until declaring scope is exited

- Storage can be created on the **stack** whenever a particular execution point is reached (an opening brace). That storage is released automatically at the complementary execution point (the closing brace). These stack-allocation operations are built into the instruction set of the processor and are very efficient. However, you have to know exactly **how many variables** you need when you're writing the program so the compiler can generate the right code.

- Storage can be allocated from a pool of memory called the **heap** (also known as the free store). This is called **dynamic memory allocation**. To allocate this memory, a function is called at runtime; this means you can decide at any time that you want some memory and how much you need. You are also responsible for determining when to release the memory, which means the lifetime of that memory can be as long as you choose – it isn't determined by scope.

Often these three regions are placed in a single contiguous piece of physical memory: the static area, the stack, and the heap (in an order determined by the compiler writer). However, there are no rules. As a programmer, these things are normally shielded from you, so all you need to think about is that the memory is there when you call for it.

Dynamic memory allocation in C++ with new and delete

Memory in your C++ program is generally divided into two parts:

- **The stack:** All variables declared inside the function will take up memory from the stack.
- **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

C++ defines two operators to allocate memory and to return it to the `common pool'. These operators are, respectively, **new** and **delete**.

```
int *ip = new int;
delete ip;
```

An int pointer variable points to memory allocated by operator new. This memory is later released by operator delete.

Here are some characteristics of operators new and delete:

- new and delete are *operators* and therefore do not require parentheses, as required for *functions* like malloc and free;

- new returns a pointer to the kind of memory that's asked for by its operand (e.g., it returns a pointer to an int);
- new uses a *type* as its operand, so it is type safe and correct amount of memory will be allocated
- delete returns void;
- for each call to new a matching delete should eventually be executed, lest a **memory leak occur**;
- delete can safely operate on a 0-pointer (doing nothing);
- in **C++** malloc and friends are *deprecated* and should be avoided.

Keep in mind following thing:

- The object is NEVER created in the heap unless explicitly allocated on the heap using **new** operator.
- Everything else is either stack (which in C++ should be called 'automatic storage') or a statically
- Creating objects on the heap involves additional overhead, both in time and in space. Here's a typical scenario.

- When you create an object with new it allocates enough storage on the heap to hold the object and calls the constructor for that storage. Thus, if you say

```
MyType *fp = new MyType(1,2);
```

This means that a pointer variable fp is created on the stack and it points to a memory in the heap, which contains the object

operator delete

The complement to the new-expression is the delete-expression, which

1. first calls the destructor and
2. then releases the memory (often with a call to free( )).

Just as a new-expression returns a pointer to the object, a delete-expression requires the address of an object.

```
delete fp;
```

delete can be called only for an object created by **new**

If the pointer you're deleting is **zero**, nothing will happen. For this reason, people often recommend setting a pointer to zero immediately after you delete it, to prevent deleting it twice. Deleting an object more than once is definitely a bad thing to do, and will cause problems.

Note : The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

Example to show how new and delete work:

If we compile and run this code,
this would produce the following result:
Value of pvalue : 29495

```
#include <iostream>
using namespace std;
int main ()
{
  double* pvalue  = NULL;     // Pointer initialized with null
  pvalue  = new double;       // Request memory for the variable

  *pvalue = 29494.99;         // Store value at allocated address
  cout << "Value of pvalue : " << *pvalue << endl;
  delete pvalue;        // free up the memory.
  return 0;
}
```

Note that the delete operator does not delete the pointer -- it deletes the memory that the pointer points to!

Dynamically allocating arrays

Declaring arrays dynamically allows us to choose their size while the program is running. To allocate an array dynamically, we use the array form of new and delete (often called new[] and delete[]):

```
int nSize = 12;
int *pnArray = new int[nSize];    // note: nSize does not need to be constant!
pnArray[4] = 7;
delete[] pnArray;
```

One of the most common mistakes that new programmers make when dealing with dynamic memory allocation is to use **delete** instead of **delete[]** when deleting a dynamically allocated array. Do not do this! Using the scalar version of delete on an array can cause data corruption or other problems.

Memory leaks

Dynamically allocated memory effectively has no scope. That is, it stays allocated until it is explicitly deallocated or until the program ends. However, the pointers used to access dynamically allocated memory follow the scoping rules of normal variables. This mismatch can create interesting problems. Consider the following function:

```
void doSomething()  {    int *pnValue = new int;   }
```

- This function allocates an integer dynamically, but never frees it using delete.
- Because pointers follow all of the same rules as normal variables, when the function ends, pnValue will go out of scope.
- Because pnValue is the only variable holding the address of the dynamically allocated integer, when pnValue is destroyed there are no more references to the dynamically allocated memory.

This is called a **memory leak**. As a result, the dynamically allocated integer cannot be deleted, and thus cannot be reallocated or reused.

- Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well. Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash.

Dynamic Memory Allocation for Objects:

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

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

class Box
{
  public:
    Box() {
      cout << "Constructor called!" <<endl;
    }
    ~Box() {
      cout << "Destructor called!" <<endl;
    }
};
int main( )
{
  Box* myBoxArray = new Box[4];
  delete [] myBoxArray; // Delete array
  return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result:

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example: CRectangle * prect; is a pointer to an object of class CRectangle.

To refer directly to a member of an object pointed by a pointer we can use the **arrow operator** (->) of indirection. Here is an example with some possible combinations:

```
// pointer to classes example
#include <iostream>
using namespace std;
```

```
class CRectangle {
    int width, height;
    public:
        void set_values (int, int);
        int area (void) {return (width * height);}
};
void CRectangle::set_values (int a, int b) {
        width = a;    height = b;
}
```

```
int main () {
    CRectangle a, *b, *c;
    CRectangle * d = new CRectangle[2];
    b= new CRectangle;        c= &a;
    a.set_values (1,2);        b->set_values (3,4);
    d->set_values (5,6);       d[1].set_values (7,8);
    cout << "a area: " << a.area() << endl;
    cout << "*b area: " << b->area() << endl;
    cout << "*c area: " << c->area() << endl;
    cout << "d[0] area: " << d[0].area() << endl;
    cout << "d[1] area: " << d[1].area() << endl;
    delete[] d;
    delete b;
    return 0;
}
```

Output of above program

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

Why no **delete c** in the code ?

c is pointing the same object as a and a is not dynamically allocated. So as soon as function ends a will be deleted automatically. c pointer will automatically out of scope after the end of function.

expression	can be read as
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
x[0]	first object pointed by x
x[1]	second object pointed by x
x[n]	(n+1)th object pointed by x

Summary on how can you read some pointer and class operators (*, &, ., ->, [ ]) that appear in this example

The this pointer

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

- The keyword **this** represents the address of the object, on which the **nonstatic** member function is being called.
- The '**this**' pointer is passed as a hidden argument to all nonstatic member function calls and is available as **a local variable** within the body of all nonstatic functions.
- '**this**' pointer is a constant pointer that holds the memory address of the **current object**.

- **'this'** pointer is not available in static member functions as static member functions can be called without any object (with class name).
- For a class X, the type of this pointer is 'X* ' (pointer to X).
- **Friend** functions do not have a this pointer, because friends are not members of a class. Only nonstatic member functions have a this pointer.

'this' can appear in various contexts as shown below:

- **When local variable's name is same as member's name**

```
#include<iostream>
using namespace std;
class Test
{
   private:   int x;   int y;
   public:
      void setX (int x)
      {
         // The 'this' pointer is used to retrieve the
         object's x  hidden by the local variable 'x'
         this->x = x;
      }
      void print() { cout << "x = " << x << endl; }
};
int main()
{
  Test obj;          int x = 20;
  obj.setX(x);     obj.print();
  return 0;
}
```

Output:
x = 20

- **For constructors, initializer list can also be used when parameter name is same as member's name.**

```
T(int x) : x(x),   y(this->x)   { }
```

Constructor

- uses parameter x to initialize member x
- uses member x to initialize member y

To return reference to the calling object

```
T& T:: Somefunc ()
{
  // Some processing
  return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to chain function calls on a single object.

```
#include<iostream>
using namespace std;
class Test
{
    private:   int x;   int y;
    public:
       Test(int x = 0, int y = 0) { this->x = x; this->y = y; }   //  Constructor
       Test& setX(int a) { x = a; return *this; }      //  return reference of current object
       Test& setY(int b) { y = b; return *this; }     //  return reference of current object
```

```
        void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1(5, 5);
     // Chained function calls.  All calls modify the same object.
    obj1.setX(10).setY(20);
    obj1.print();
    return 0;
}
```

```
Output:
x = 10 y = 20
```

Message

A running program is a pool of objects where objects are created, destroyed and interacting. This interacting is based on messages which are sent from one object to another asking the recipient to apply a method on itself.

- A message is always given to some object, called the receiver.
- The action performed in response to the message is not fixed, but may differ depending upon the class of the receiver. That is, different objects may accept the same message, and yet perform different actions.
- Three identifiable parts of any message passing expression :
  - Receiver (the object to which the message is being sent)
  - The message selector (the text that indicates the particular message being sent)
  - The arguments using in responding to the message

For example, we could use

```
Integer i;        /* Define a new integer object */
i.setValue(1);    /* Set its value to 1 */
```

to express the fact, that the integer object i should set its value to 1. This is the message "Apply method setValue with argument 1 on yourself." sent to object i. In C++ message is sending using dot operator. In some other object-oriented languages might use other notations, for example "->".

Sending a message asking an object to apply a method is similar to a procedure call in "traditional" programming languages. However, in object orientation there is a view of autonomous objects which communicate with each other by exchanging messages. Objects react when they receive messages by applying methods on themselves. They also may deny the execution of a method, for example if the calling object is not allowed to execute the requested method.

In our example, the message and the method which should be applied once the message is received have the same name: We send "setValue with argument 1" to object i which applies "setValue(1)". Consequently, invocation of a method is just a reaction caused by receipt of a message. This is only possible, if the method is actually known to the object.

Nested Class

- A nested class is a class which is declared in another enclosing class.
- A nested class is a member and as such has the same access rights as any other member.
- The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

```
#include<iostream>
using namespace std;
class Student
{
    private:
        string regdno;
        string branch;
        int age;
    public:
        class Name
        {
            private:
                string fname;
                string mname;
                string lname;
            public:
                string get_name()  {   return fname+" "+mname+" "+lname;   }
                void set_name(string f, string m, string l)  {
                        fname = f;      mname = m;    lname = l;
                }
        };
};
int main()
{
    Student::Name n;
    n.set_name("P", "S", "Suryateja");
    cout<<"Name is: "<<n.get_name();
    return 0;
}
```

Quoting draft of C++11 (N3290):

9.7 Nested class declarations [class.nest]
1 A class can be declared within another class. A class declared within another is called a nested class.

9.8 Local class declarations [class.local]
1 A class can be declared within a function definition; such a class is called a local class.
There is no concept of inner class specified in C++ standard.

Output for the above program is as follows:
Name is: P S Suryateja

What is the difference between C++ and Java inner classes?

- The main difference is that C++ nested classes, unlike Java inner classes, do not capture the instance of the outer class. In other words, C++ nested classes are pretty much similar to Java's static inner classes.
- For instance, the following code is broken:

```
class Outer {
 public:
  class Inner {
   public:
    int get_x() {
      return x;    // ERROR!
    }
  };
  int x;
};
```

But, this one works:

```
class Outer {
 public:
  class Inner {
   public:
    int get_x() {
      return x;
    }
  };
  static int x;
};
```

No error in this line