## Initializing Class Objects: CONSTRUCTORS

The class designer can guarantee initialization of every object by providing a special member function called the **constructor**.

The constructor is invoked **automatically** each time an object of that class is created (instantiated).

These functions are used to (for example) assign initial values to the data members, open files, establish connection to a remote computer etc.

The constructor can take parameters as needed, but it cannot have a return value (even not void).

The constructor has the same name as the class itself.

There are different types of constructors.

For example, a constructor that defaults all its arguments or requires no arguments, i.e. a constructor that can be invoked with no arguments is called default constructor.

In this section we will discuss different kinds of constructors.

## Default Constructor:

A constructor that defaults all its arguments or requires no arguments, i.e. a constructor that can be invoked with no arguments.

```
class Point{                          // Declaration Point Class
   int x,y;                           // Attributes: x and y coordinates
 public:
   Point();                           // Declaration of the default constructor
   bool move(int, int);               // A function to move points
   void print();                      // to print coordinates on the screen
};
// Default Constructor
Point::Point()
{
   x = 0;                             // Assigns zero to coordinates
   y = 0;
}
// -------- Main Program -------------
int main()
{
   Point p1, p2;              // Default construct is called 2 times
   Point *ptr;                // ptr is not an object, constructor is NOT called
   ptr = new Point;           // Object is created, default constructor is called
```

See Example e41.cpp

## Constructors with Parameters:

Like other member functions, constructors may also have parameters.

Users of the class (client programmer) must supply constructors with necessary arguments.

```
class Point{                    // Declaration Point Class
  int x,y;                      // Properties: x and y coordinates
 public:
  Point(int, int);              // Declaration of the constructor
  bool move(int, int);          // A function to move points
  void print();                 // to print coordinates on the screen
};
```

This declaration shows that the users of the Point class have to give two integer arguments while defining objects of that class.

**Example:**

In the following example, it is assumed that the points are not allowed to have negative coordinates.

```
// A constructor with Parameters
// Points may not have negative coordinates
Point::Point(int x_first, int y_first)
{
   if ( x_first < 0 )                 // If the given value is negative
        x = 0;                        // Assigns zero to x
   else
        x = x_first;
   if ( y_first < 0 )                 // If the given value is negative
        y = 0;                        // Assigns zero to y
   else
        y = y_first;
}

// -------- Main Program -------------
int main()
{
   Point p1(20, 100), p2(-10, 45);    // Construct is called 2 times
   Point *ptr = new Point(10, 50);    // Construct is called once
   Point p3;                          // ERROR! There is not a default constructor
   :
}
```

See Example e42.cpp

## Multiple Constructors

The rules of function overloading are also valid for constructors. So, a class may have more than one constructor with different type of input parameters.

```
Point::Point()                        // Default constructor
{
...............                       // Body is not important
}

Point::Point(int x_first, int y_first)   // A constructor with parameters
{
................                      // Body is not important
}
```

Now, the client programmer can define objects in different ways:

```
Point p1;                             // Default constructor is called
Point p2(30, 10);                     // Constructor with parameters is called
```

The following statement causes an compiler error, because the class does not include a constructor with only one parameter.

```
Point p3(10);                 //ERROR! There isn't a constructor with one parameter
```

## Default Values of Constructor Parameters

Like other functions, parameters of constructors may also have default values.

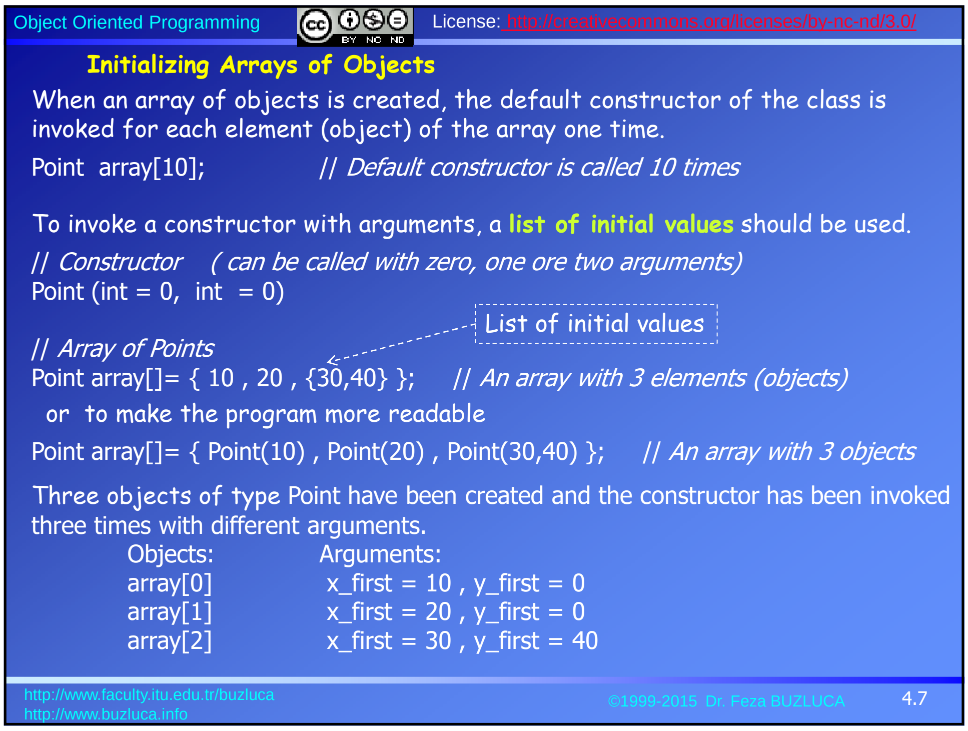
```
class Point{
  public:
     Point (int = 0,  int = 0);        //   Default values must be in the  declaration
     :
};
Point::Point (int x_first,  int y_first)
{
    if ( x_first < 0 )              // If the given value is negative
            x = 0;                  // Assigns zero to x
    else    x = x_first;
    if ( y_first < 0 )              // If the given value is negative
            y = 0;                  // Assigns zero to y
    else    y = y_first;
}
```

Now, client of the class can create objects as follows:

```
Point p1(15, 75);             // x=15, y=75
Point p2(100);                // x=100, y=0
```

This function can be also used as a default constructor

```
Point p3;                     // x=0, y=0
```



## Initializing Arrays of Objects

When an array of objects is created, the default constructor of the class is invoked for each element (object) of the array one time.

```
Point  array[10];            // Default constructor is called 10 times
```

To invoke a constructor with arguments, a **list of initial values** should be used.

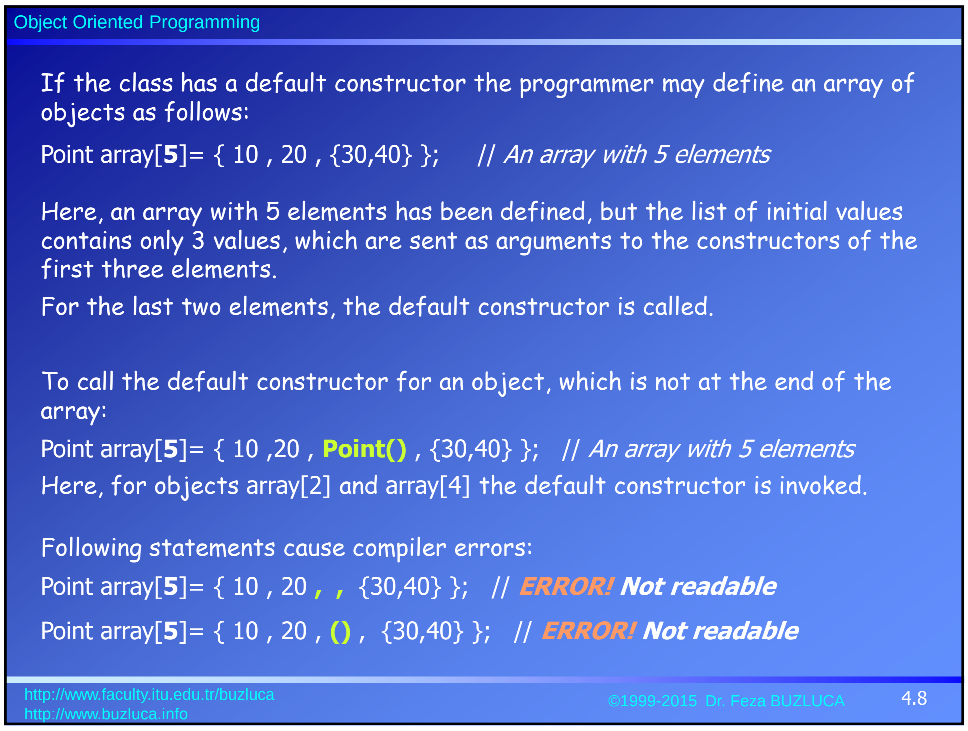
```
// Constructor    ( can be called with zero, one ore two arguments)
Point (int = 0,  int  = 0)
                                            List of initial values
// Array of Points
Point array[]= { 10 , 20 , {30,40} };    // An array with 3 elements (objects)
```

or  to make the program more readable

```
Point array[]= { Point(10) , Point(20) , Point(30,40) };    // An array with 3 objects
```

Three objects of type Point have been created and the constructor has been invoked three times with different arguments.

| Objects: | Arguments: |
|---|---|
| array[0] | x_first = 10 , y_first = 0 |
| array[1] | x_first = 20 , y_first = 0 |
| array[2] | x_first = 30 , y_first = 40 |



If the class has a default constructor the programmer may define an array of objects as follows:

```
Point array[5]= { 10 , 20 , {30,40} };     // An array with 5 elements
```

Here, an array with 5 elements has been defined, but the list of initial values contains only 3 values, which are sent as arguments to the constructors of the first three elements.

For the last two elements, the default constructor is called.

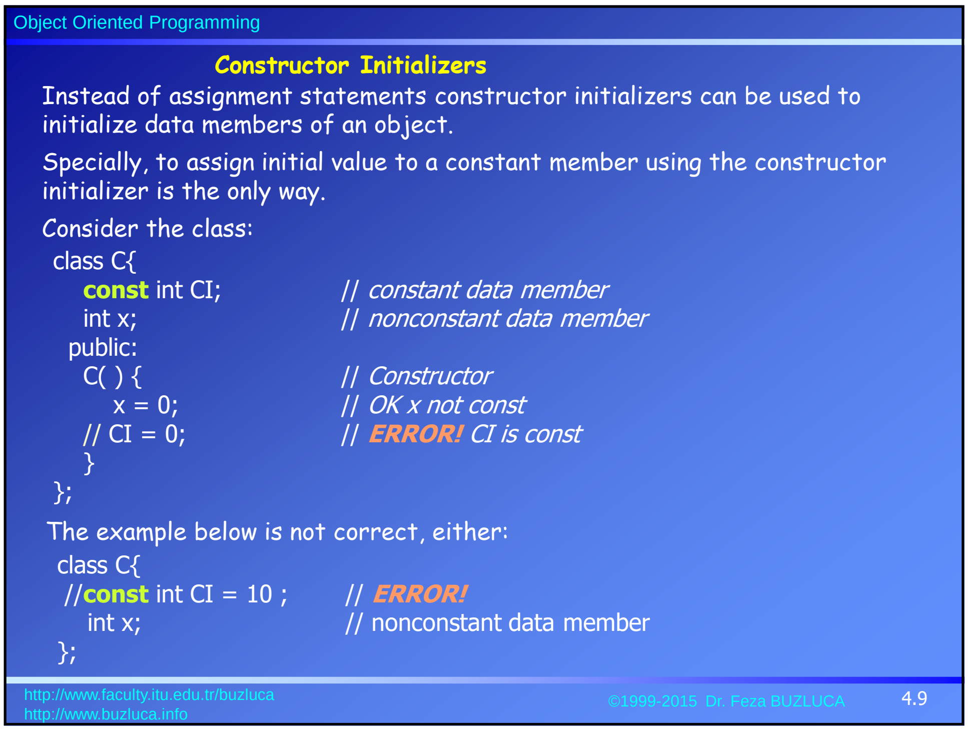
To call the default constructor for an object, which is not at the end of the array:

```
Point array[5]= { 10 ,20 , Point() , {30,40} };   // An array with 5 elements
```

Here, for objects array[2] and array[4] the default constructor is invoked.

Following statements cause compiler errors:

```
Point array[5]= { 10 , 20 , , {30,40} };   // ERROR! Not readable

Point array[5]= { 10 , 20 , () ,  {30,40} };   // ERROR! Not readable
```



## Constructor Initializers

Instead of assignment statements constructor initializers can be used to initialize data members of an object.

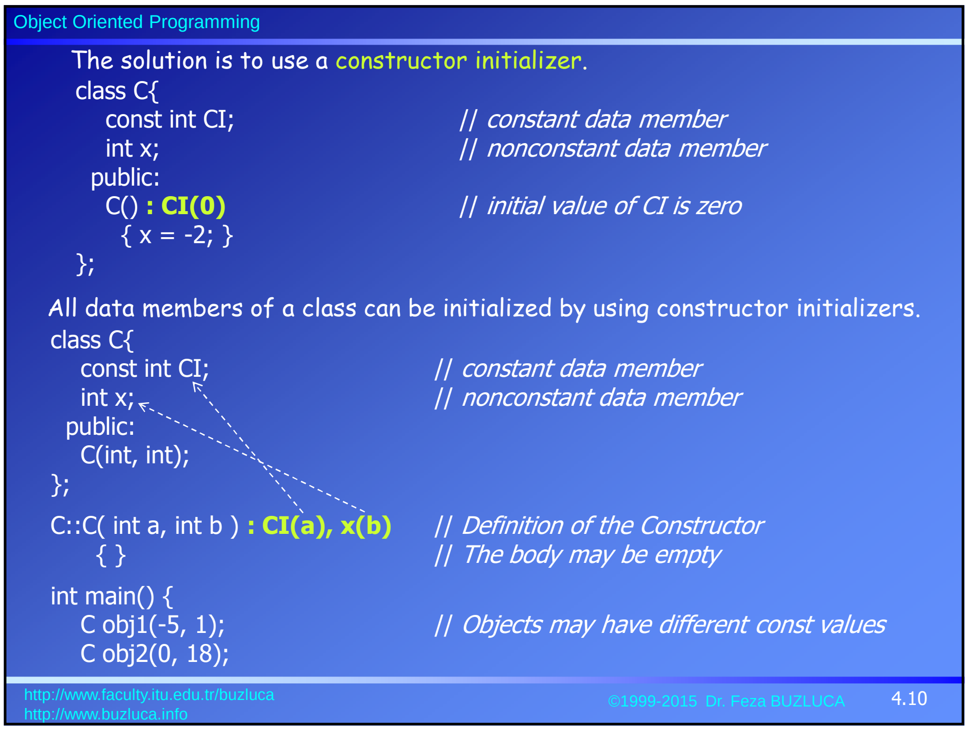
Specially, to assign initial value to a constant member using the constructor initializer is the only way.

Consider the class:

```
class C{
    const int CI;             // constant data member
    int x;                    // nonconstant data member
  public:
    C( ) {                    // Constructor
        x = 0;                // OK x not const
    // CI = 0;                // ERROR! CI is const
    }
};
```

The example below is not correct, either:

```
class C{
  //const int CI = 10 ;       // ERROR!
    int x;                    // nonconstant data member
};
```



The solution is to use a constructor initializer.

```
class C{
    const int CI;                 // constant data member
    int x;                        // nonconstant data member
  public:
    C() : CI(0)                   // initial value of CI is zero
      { x = -2; }
};
```

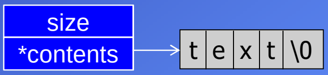
All data members of a class can be initialized by using constructor initializers.

```
class C{
    const int CI;                 // constant data member
    int x;                        // nonconstant data member
  public:
    C(int, int);
};

C::C( int a, int b ) : CI(a), x(b)    // Definition of the Constructor
      { }                             // The body may be empty

int main() {
    C obj1(-5, 1);                    // Objects may have different const values
    C obj2(0, 18);
```

## DESTRUCTORS

- The destructor is called automatically
  1. when each of the objects goes out of scope or
  2. a dynamic object is deleted from memory by using the delete operator.
- A destructor is characterized as having the same name as the class but with a tilde '~' preceded to the class name.
- A destructor has no return type and receives no parameters.
- A class may have only one destructor.

**Example:** A user defined String class



```
class String{
    int size;                    // Length (number of chars) of the string
    char *contents;              // Contents of the string
 public:
   String(const char *);         // Constructor
   void print();                 // An ordinary member function
  ~String();                     // Destructor
};
```

Actually, the standard library of C++ contains a **string** class. Programmers don't need to write their own String class. We write this class only to show some concepts.

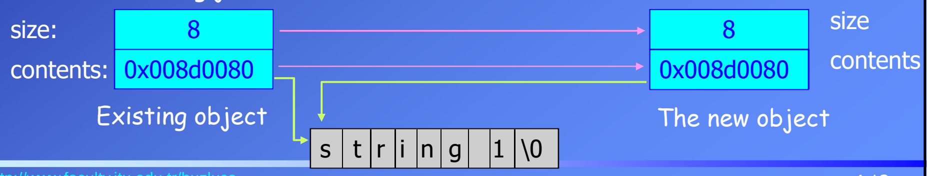
```
// Constructor : copies the input character array that terminates with a null character
// to the contents of the string

String::String(const char *in_data)
{
    size = strlen(in_data);           // strlen is a function of the cstring library
    contents = new char[size +1];     // +1 for null ( '\0' ) character
    strcpy(contents, in_data);        // input_data is copied to the contents
 }

void String::print()
{
    cout << contents << " " << size << endl;
}

// Destructor
// Memory pointed by contents is given back
String::~String()
{
     delete[] contents;
}
```

```
int main()          // Test program
{
    String string1("string 1");
    String string2("string 2");
    string1.print();
    string2.print();
    return 0;        // destructor is called twice
}
```

See Example e43.cpp

## Copy Constructor

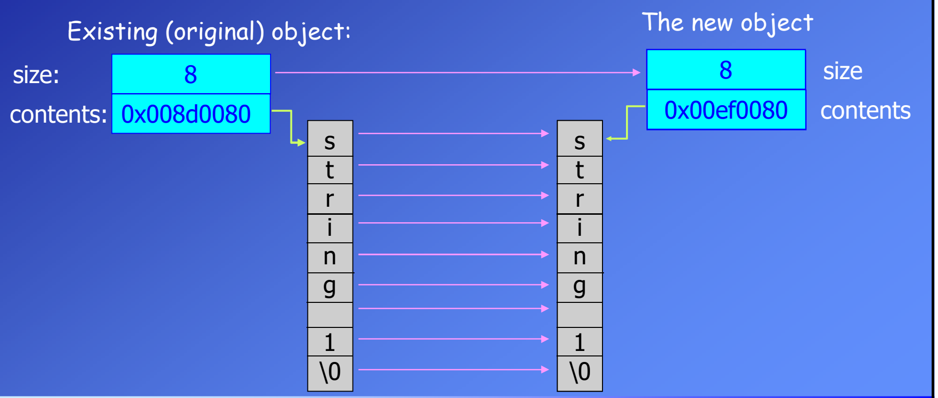
- Sometimes we want to create a new object, which is the copy (has the same data) of an existing object.
- Copy constructor is a special type of constructors and used to copy the contents of an object to a new object during construction of that new object.
- The type of its input parameter is a *reference* to objects of the same type. The input argument is the object that will be copied into the new object.
- The copy constructor is generated automatically by the compiler if the class author fails to define one.
- If the compiler generates it, it will simply copy the contents of the original into the new object as a byte by byte copy.
- For simple classes with no pointers, that is usually sufficient, but if there is a pointer as a class member so a byte by byte copy would copy the pointer from one to the other and they would both be pointing to the same allocated member.
- For example the copy constructor, generated by the compiler for the String class will do the following job:



The copy constructor, generated by the compiler can not copy the memory locations pointed by the member pointers.

The programmer must write its own copy constructor to perform these operations.

Example: User-written copy constructor



## **Example:** The copy constructor of the String class

```
class String{                               // User defined String class
    int size;
    char *contents;
 public:
    String(const char *);                   // Constructor
    String(const String &);                 // Copy Constructor
    void print();                           // Prints the string on the screen
    ~String();                              // Destructor
};

String::String(const String &object_in)     // Copy Constructor
{
    size = object_in.size;
    contents = new char[size + 1];          //  +1 for null character
    strcpy(contents, object_in.contents);
}

int main()                                  // Test program
{
    String my_string("string 1");
    my_string.print();
    String other = my_string;               // Copy constructor is invoked
    String more(my_string);                 // Copy constructor is invoked
    ......
```

See Example e44.cpp

## Constant Objects and Const Member Functions

The programmer may use the keyword **const** to specify that an object is not modifiable.

Any attempt to modify (to change the attributes) directly or indirectly (by calling a function) causes a compiler error.

For example:

```
const ComplexT CZ(0,1);    // Constant object
```

C++ compilers totally disallow any member function calls for const objects.

The programmer may declare some functions as **const**, which do not modify any data (attributes) of the object.

Only const functions can operate on const objects.

**Example:**

```
class Point{                // Declaration Point Class
    int x, y;               // Attributes: x and y coordinates
  public:
    Point(int, int);        // Declaration of the constructor
    bool move(int, int);    // A function to move points
    void print() const;     // constant function: prints coordinates on the screen
};
```

```
// Constant function: It prints the coordinates on the screen
void Point::print() const
{
    cout << "X= " << x << ", Y= " << y << endl;
}
// -------- Test Program -------------
int main()
{
    const Point cp(10,20);          // constant point
    Point ncp(0,50);                // non-constant point
    cp.print();                     // OK. Const function operates on const object
    cp.move(30,15);                 // ERROR! Non-const function on const object
    ncp.move(100,45);               // OK. ncp is non-const
    return 0;
}
```

A const method can invoke only other const methods, because a const method is not allowed to alter an object's state either directly or indirectly, that is, by invoking some nonconst method.

See Example e45.cpp

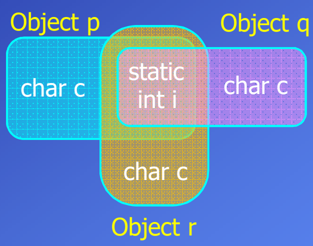
Declare necessary methods as constant to prevent errors and to allow users of the class to define constant objects.

## Static Class Members

Normally, each object of a class has its own copy of all data members of the class.

In certain cases only one copy of a particular data member should be shared by all objects of a class. A static data member is used for this reason.

```
class A{
   char c;
   static int i;
};
int main()
{
    A   p, q, r;
    :
}
```



Static data members exist even no objects of that class exist.
Static data members can be public or private.
To access public static data when no objects exist use the class name and binary scope resolution operator. For example **A::i= 5;**
To access private static data when no objects exist, a public **static member function** must be provided.
They must be initialized **once** (and only once) at file scope.

See Example e46.cpp

## Passing Objects to Functions as Arguments

Objects should be passed or returned by reference unless there are compelling reasons to pass or return them by value.

Passing or returning by value can be especially inefficient in the case of objects. Recall that the object passed or returned by value must be *copied* into stack and the data may be large, which thus wastes storage. The copying itself takes time.

If the class contains a copy constructor the compiler uses this function to copy the object into stack.

See Example e47.cpp

We should pass the argument by reference because we don't want an unnecessary copy of it to be created. Then, to prevent the function from accidentally modifying the original object, we make the parameter a **const reference**.

See Example: e48.cpp

```
ComplexT & ComplexT::add(const ComplexT& z)
{
    ComplexT  result;          // local object
    result.re = re + z.re;
    result.im = im + z.im;
    return result;             // ERROR!
}
```

**Remember**, local variables can not be returned by reference.

## Avoiding Temporary Objects

In the previous example, within the add function a temporary object (result) is defined to add two complex numbers.

Because of this object, constructor and destructor are called.

Avoiding the creation of a temporary object within add() saves time and memory space.

```
ComplexT ComplexT::add(const ComplexT& c)
{
    double  re_new,im_new;
    re_new = re + c.re;
    im_new = im + c.im;
    return ComplexT(re_new, im_new);          // Constructor is called
}
```

The only object that's created is the return value in stack, which is always necessary when returning by value.

This could be a better approach, if creating and destroying individual member data items is faster than creating and destroying a complete object.

See Example: e49.cpp

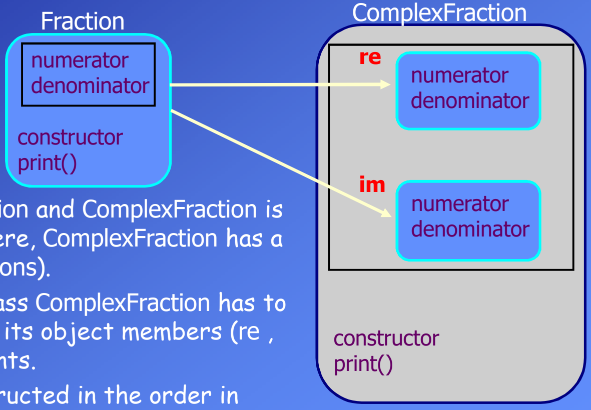
## NESTING OBJECTS: Classes as Members of Other Classes

A class may include objects of other classes as its data members.

In the example below, a class is designed (ComplexFraction) to define complex numbers. The data members of this class are fractions which are objects of another class (Fraction).

ComplexFraction:

$$z = \underbrace{\frac{a}{b}}_{\text{re:Fraction}} + \underbrace{\frac{c}{d}}_{\text{im:Fraction}} i$$



The relation between Fraction and ComplexFraction is called "**has a relation**". Here, ComplexFraction has a Fraction (actually two Fractions).

Here, the author of the class ComplexFraction has to supply the constructors of its object members (re , im) with necessary arguments.

Member objects are constructed in the order in which they are declared and before their enclosing class objects are constructed.

**Example:** A class to define fractions

```
class Fraction{                              // A class to define fractions
    int numerator, denominator;
 public:
    Fraction(int, int);                      // CONSTRUCTOR
    void print() const;
};

Fraction::Fraction(int num, int denom)       // CONSTRUCTOR
{
    numerator = num;
    if (denom==0) denominator = 1;
    else denominator = denom;
 }

void Fraction::print() const                 // Print function
{
    cout << numerator << "/" << denominator << endl;
}
```

**Example:** A class to define complex numbers. It contains two objects as members

```
class ComplexFraction{        //  Complex numbers, real and imag. parts are fractions
    Fraction re, im;          //  objects as data members of another class
 public:
    ComplexFraction(int,int);           // Constructor
    void print() const;
 };
ComplexFraction::ComplexFraction(int re_in, int im_in) : re(re_in, 1) , im(im_in, 1)
 {
     :
 }
 void ComplexFraction::print() const
 {
     re.print();  // print of Fraction is called
     im.print(); // print of Fraction is called
 }
 int main()
 {
      ComplexFraction cf(2,5);
      cf.print();
      return 0;
 }
```

Data members are initialized

When an object goes out of scope, the destructors are called in reverse order: The enclosing object is destroyed first, then the member (inner) object.

See Example: e410.cpp

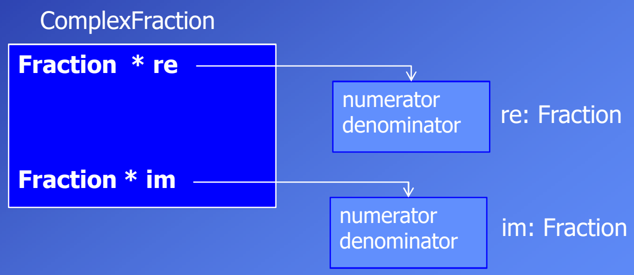
See Example: e411.cpp

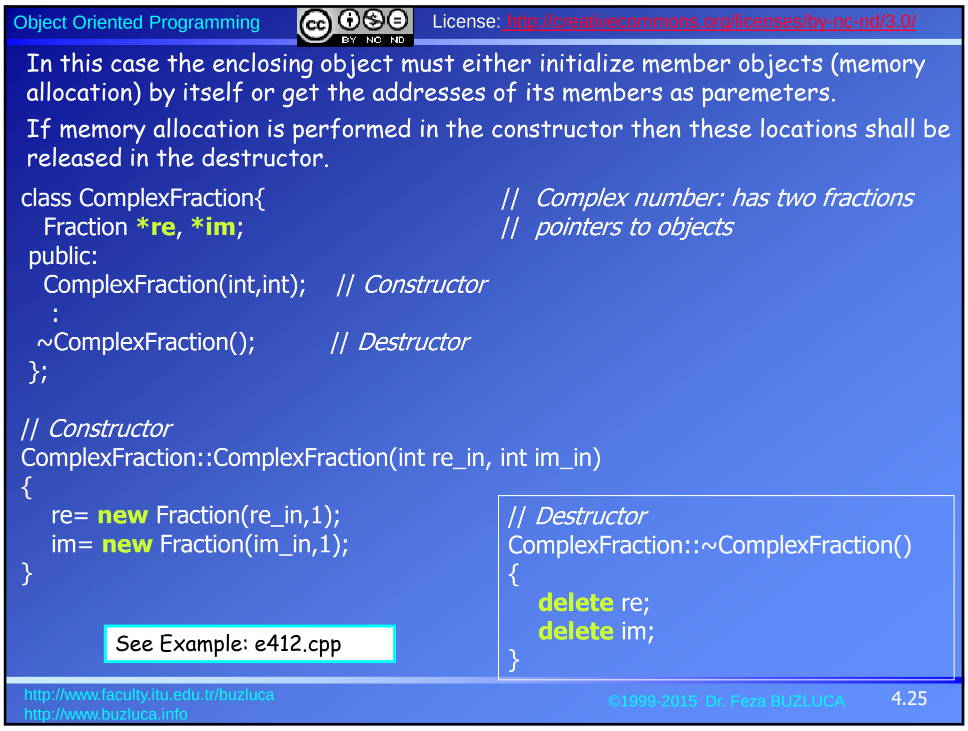
## Dynamic Memebers (Pointers as members)

Instead of static objects, data members of a class may also be pointers to objects.

```
class ComplexFraction{       //  Complex numbers, real and imag. parts are fractions
    Fraction *re, *im;       //  pointers to objects as data members of another class
 public:
    :
 };
```

Now, only the pointers (addresses) of member objects are included in objects of ComplexFraction. The member objects re and im must be created separately.





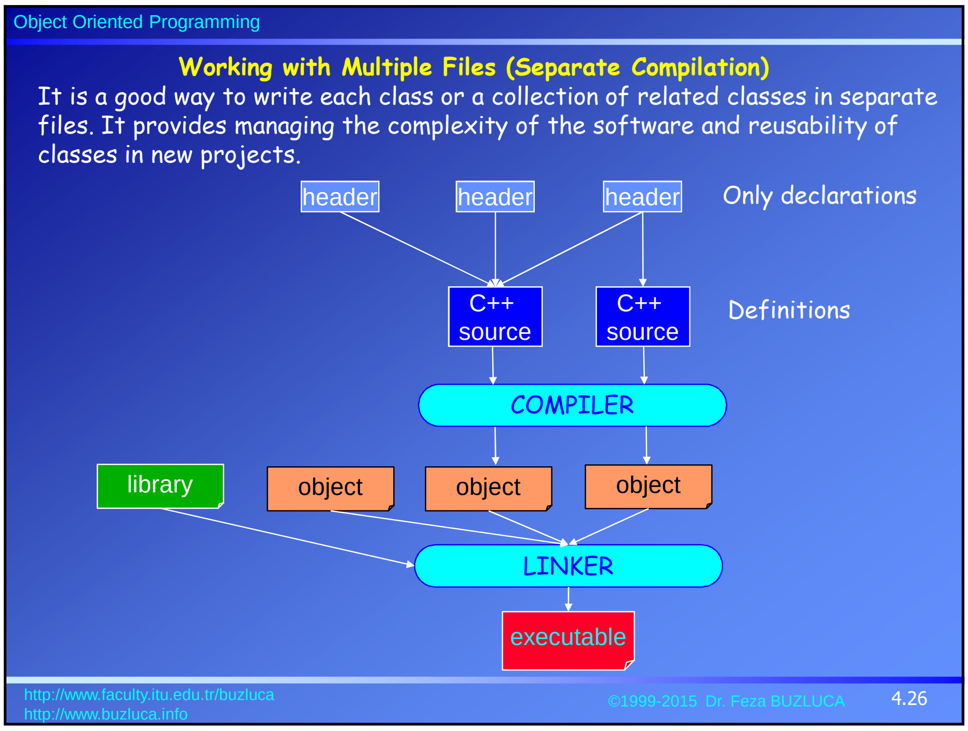
In this case the enclosing object must either initialize member objects (memory allocation) by itself or get the addresses of its members as paremeters.

If memory allocation is performed in the constructor then these locations shall be released in the destructor.

```
class ComplexFraction{                    //  Complex number: has two fractions
   Fraction *re, *im;                     //  pointers to objects
 public:
   ComplexFraction(int,int);    // Constructor
     :
  ~ComplexFraction();          // Destructor
 };

// Constructor
ComplexFraction::ComplexFraction(int re_in, int im_in)
{
    re= new Fraction(re_in,1);
    im= new Fraction(im_in,1);
}
```

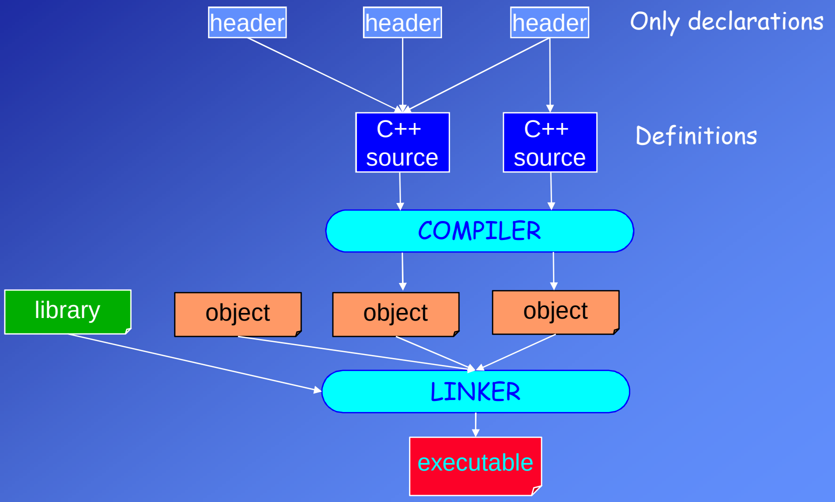
```
// Destructor
ComplexFraction::~ComplexFraction()
{
    delete re;
    delete im;
}
```

See Example: e412.cpp



## Working with Multiple Files (Separate Compilation)

It is a good way to write each class or a collection of related classes in separate files. It provides managing the complexity of the software and reusability of classes in new projects.



When using *separate compilation* you need some way to automatically compile each file and to tell the linker to build all the pieces along with the appropriate libraries and startup code into an executable file.

The solution, developed on Unix but available everywhere in some form, is a program called **make**.

Compiler vendors have also created their own project building tools.

These tools ask you which files are in your project and determine all the relationships themselves.

These tools use something similar to a **makefile**, generally called a *project file*, but the programming environment maintains this file so you don't have to worry about it.

The configuration and use of project files varies from one development environment to another, so you must find the appropriate documentation on how to use them (although project file tools provided by compiler vendors are usually so simple to use that you can learn them by playing around).

We will write the example e410.cpp about fractions and complex numbers again. Now we will put the class for fractions and complex numbers in separate files.

See Example: e413.zip