Initializing, Assigning, and Destroying Class Objects

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info

1

Overview

- Constructors
- Initializing Arrays of Objects
- Member Initializers
- Copy Constructors
- Destructors
- `const` (Constant) Objects and Constant Member Functions
- `static` Class Members
- Passing Objects to Functions
- Composition (Objects As Members of Other Classes)
- Dynamic Members
- Working with Multiple Files (Separate Compilation)

2

Constructor

- is a special member function the class designer provides to guarantee initialization of every object
- is invoked automatically each time an object of that class is created (instantiated)
- performs initializations
  - assigning initial values to data members, opening files, establishing connection to a remote computer, etc.
- can take parameters as needed, but it cannot return a value, so it cannot specify a return type (not even `void`).
- has the same name as the class itself

Different Types 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
- Constructor with arguments
- Copy constructor

Default Constructor: Example

- Constructor that defaults all its arguments (or requires no arguments), i.e., a constructor that can be invoked with no arguments

```
class Point {                  // Point class definition
   int x, y;                   // attributes: x- and y-coordinates
 public:
   Point();                    // default constructor
   bool move( int, int );      // move point
   void print();               // print coordinates on the screen
};
// default constructor
Point::Point()
{
   x = 0;                      // assign zero to coordinates
   y = 0;
}
// -------- Main Program -------------
int main()
{
   Point p1, p2;    // default constructor is called twice
   Point *ptr;      // ptr not an object, constructor is NOT called
   ptr = new Point;// object created, default constructor is called
```

See Example e41.cpp

5

Constructor with Arguments

- Like other member functions, a constructor may also have arguments
- Users of the class (client programmers) must provide necessary arguments to the constructor

```
class Point {                  // Point class definition
   int x, y;                   // properties: x-and y-coordinates
 public:
   Point( int, int );          // constructor
   bool move( int, int );      // move point
   void print();               // print coordinates on the screen
};
```

- Users of Point class have to provide two integer arguments while defining objects of that class

6

Constructor with Arguments: Example

```
// Points may not have negative coordinates
Point::Point( int xFirst, int yFirst )
{
   if ( xFirst < 0 )          // if the given value is negative
      x = 0;                  // assign zero to x
   else
      x = xFirst;
   if ( yFirst < 0 )          // if the given value is negative
      y = 0;                  // assign zero to y
   else
      y = yFirst;
}

// -------- Main Program -------------
int main()
{
   Point p1( 20, 100 ), p2( -10, 45 );  // constructor called twice
   Point *ptr = new Point( 10, 50 );    // constructor called once
   Point p3;    // ERROR! No default constructor exists
    :
}
```

If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class.

See Example e42.cpp

Multiple Constructors

- Rules of function overloading also apply to constructors
- A class may have more than one constructor with different numbers and/or types of parameters (you can define several overloaded constructors for a class)

```
Point::Point()                              // default constructor
{
..............                              // body is not important
}

Point::Point( int xFirst, int yFirst ) // constructor with params.
{
.................                          // body is not important
}
```

- Client programmer can create objects in different ways:

```
Point p1;            // default constructor is called
Point p2( 30, 10 );  // constructor with arguments is called
```

- The following statement causes a compiler error because the class does not include a constructor with only one argument

```
Point p3( 10 ); //ERROR! No constructor exists with one argument
```

Default Values of Constructor Arguments

- Like other functions, constructors can specify default arguments

```
class Point {
  public:
     Point( int = 0,  int = 0 ); // Def. vals. must be in decl.
     :
};
Point::Point (int xFirst,  int yFirst)
{
   if ( xFirst < 0 )          // if the given value is negative
      x = 0;                  // assign zero to x
   else     x = xFirst;
   if ( yFirst < 0 )          // if the given value is negative
      y = 0;                  // assigns zero to y
   else    y = yFirst;
}
```

- Now, clients 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 once for each element (object) of the array

```
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, or two arguments)
Point( int = 0,  int = 0 );

// array of points: an array with 3 elements (objects)
Point array[] = { ( 10 ) , ( 20 ) , ( 30, 40 ) };
```

List of initial values

- Alternatively, to make the program more readable

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

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

```
  Objects:     Arguments:
  array[0]     xFirst = 10 , yFirst = 0
  array[1]     xFirst = 20 , yFirst = 0
  array[2]     xFirst = 30 , yFirst = 40
```

Initializing Arrays of Objects

- If class has a default constructor, programmer may define array of objects as

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

- 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, we would use

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

- For objects array[2] and array[4], the default constructor is invoked
- Following statements cause compiler errors

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


Member Initializers

- Instead of assignment statements, member initializers can be used to initialize data members of an object
- Using the member initializer is the only way of assigning an initial value to a constant member
- 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
};
```

Member Initializers

- The solution is to use a member 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 using member initializers

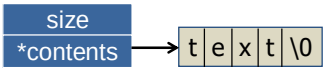
```
class C {
    const int CI;            // constant data member
    int x;                   // nonconstant data member
  public:
    C( int, int);
};

C::C( int a, int b ) : CI( a ), x( b )      // constructor
      { }                                   // 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 when
  - an object goes out of scope or
  - a dynamic object is deleted from memory using the `delete` operator
- A destructor has the same name as the class but with a tilde '~' preceding the class name
- A destructor has no return type and receives no parameters
- A class can have only one destructor

Destructor Example

- Example: A user-defined String class




```
class String {
   int size;               // length (number of chars) of string
   char *contents;         // contents of the string
 public:
   String(const char *);   // constructor
   void print();           // an ordinary member function
   ~String();              // destructor
};
```

- C++ Standard Library contains a string class
- Programmers do not need to write their own String class
- We write this class only to illustrate some concepts

Destructor Example

```
// Constructor : copies the input character array that terminates
// with a null character to the contents of the string
String::String( const char *inData )
{
   size = strlen( inData );        // strlen (cstring library)
   contents = new char[size + 1];  // +1 for null ( '\0' ) char.
   strcpy( contents, inData );     // inData copied to contents
}

void String::print()
{
   cout << contents << " ";
   cout << 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; // destr. called twice
}
```

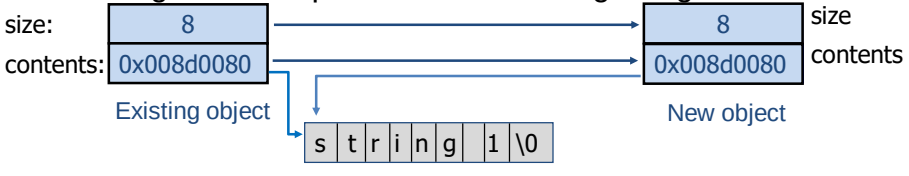
See Example e43.cpp

Copy Constructors

- Sometimes we want to create a new object which is the copy of (has the same data as) an existing object
- Copy constructor
  - is a special type of constructor
  - is used to copy the contents of an object to a new object during construction of that new object
  - its input parameter type is a reference to objects of the same type
  - its input argument is the object that will be copied into the new object
  - is generated automatically by the compiler if the class programmer fails to define one

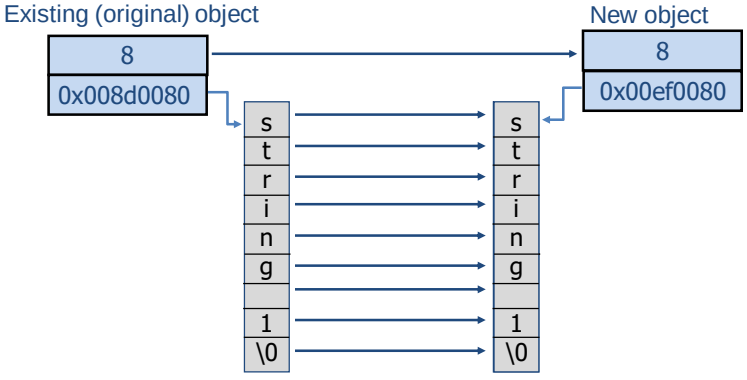
Copy Constructor Generated by Compiler

- 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
- However, if there is a pointer as a class member, a byte-by-byte copy would copy the pointer in the source object to the target object's pointer, and they would both point to the same dynamicaly allocated memory
- Example: copy constructor, generated by the compiler for the String class will perform the following assignment:



Copy Constructors

- Copy constructor generated by the compiler cannot copy memory locations member pointers point to
- Programmer must write his own copy constructor to perform these operations
- Example: User-defined copy constructor



Copy Constructor: Example

```
class String {                  // user-defined String class
   int size;
   char *contents;
 public:
   String( const char * );     // constructor
   String( const String & );   // copy constructor
   void print();               // print the string on the screen
   ~String();                  // destructor
};
String::String( const String &objectIn ) // copy constructor
{
   size = objectIn.size;
   contents = new char[ size + 1 ];      //  +1 for null character
   strcpy( contents, objectIn.contents );
}
int main()                              // test program
{
   String myString( "string 1" );
   myString.print();
   String other = myString;             // copy constructor is invoked
   String more(myString);               // copy constructor is invoked
   ……
```

See Example e44.cpp

Constant Objects and Const Member Functions

- Programmer may use the keyword const to specify that an object is not modifiable
- Any attempt to modify the object (to change the attributes) directly or indirectly (by calling a function) causes a compiler error
- Example:

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

- C++ disallows member function calls for const objects unless the member functions themselves are also declared const
- Programmer may declare some functions that do not modify any data (attributes) of the object as const
- Only const functions can operate on const objects

Constant Objects and Const Member Functions: Example

```
class Point {          // Point class definition
   int x, y;           // attributes: x- and y-coordinates
 public:
   Point( int, int );        // constructor
   bool move( int, int );    // move points
   void print() const;       // constant function: print
                             // coordinates on the screen
};
```

Constant Objects and Const Member Functions: Example

```
// constant function: print 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 );       // nonconstant point
   cp.print();         // OK. Const func. operates on const obj.
   cp.move( 30, 15 );  // ERROR! Nonconst func. on const obj.
   ncp.move(100, 45 ); // OK. ncp is nonconst
   return 0;
}
```

See Example e45.cpp

Constant Objects and Const Member Functions

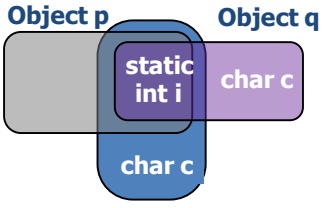
- A const method can invoke only other const methods because a const method is not allowed to alter an object's state either directly or indirectly, that is, by invoking some nonconst method.

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

static Data 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;
    :
}
```



See Example e46.cpp

static Data Members

- Static data members exist even no objects of the class exist
- Static data members can be declared public or private
- To access a public static class member when no objects of the class exist, use the class name and binary scope resolution operator
  - Example: A::i = 5;
- To access private static class member when no objects of the class exist, provide a public static member function, and call the function by prefixing its name with the class name and scope resolution operator
- Static data members must be initialized once (and only once) at file scope

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 the stack. The data may be large, wasting storage. The copying itself takes time.
- If the class contains a copy constructor, the compiler uses this function to copy the object into the stack

See Example e47.cpp

Passing Objects to Functions as Arguments

- We should pass the argument by reference because we do not want an unnecessary copy to be created
- To prevent the function from accidentally modifying the original object, we make the parameter a const reference

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

See Example e48.cpp

Remember: Local variables cannot be returned by reference.

Avoiding Temporary Objects

- In the previous example, within the `add` function, a temporary object (`result`) was defined to add two complex numbers
- Because of this object, constructor and destructor were called
- Avoiding the creation of a temporary object within `add()` saves time and memory space

```
ComplexT ComplexT::add(const ComplexT& c)
{
    double  reNew, imNew;
    reNew = re + c.re;
    imNew = im + c.im;
    return ComplexT(reNew, imNew);   // constructor is called
}
```

See Example e49.cpp

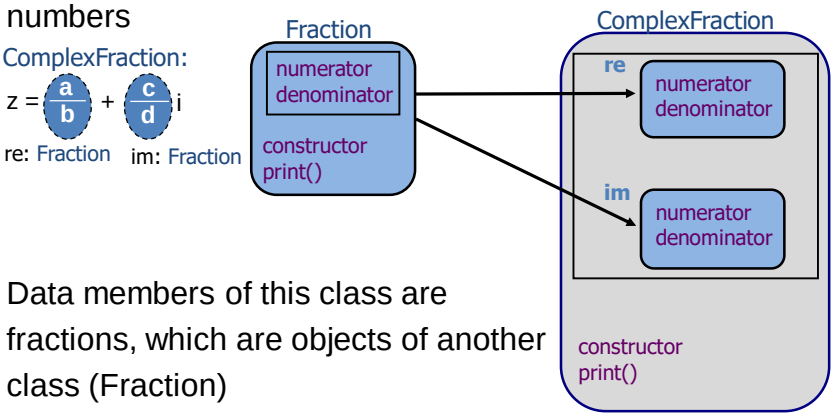
- The only object that is created is the return value in the stack, which is always necessary when returning by value
- This could be a better approach, if creating and destroying individual member data items are faster than creating and destroying a complete object

Composition: Objects as Members of Other Classes

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

Composition: Objects as Members of Other Classes

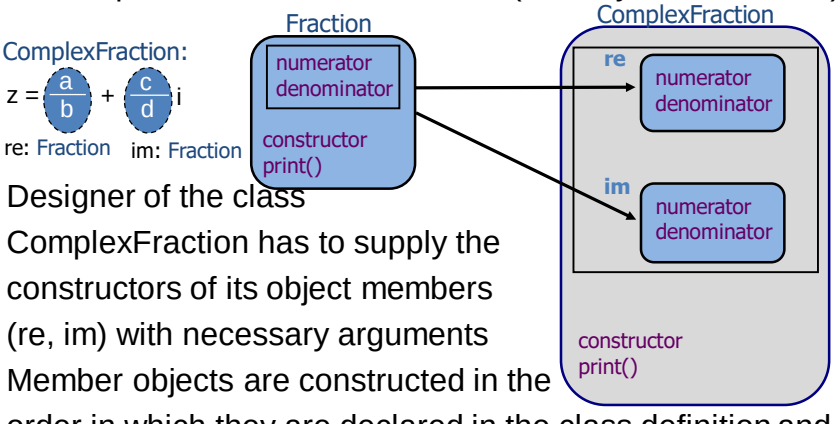
- Class is designed (ComplexFraction) to define complex numbers



- Data members of this class are fractions, which are objects of another class (Fraction)

Composition

- Relation between Fraction and ComplexFraction is called "has-a" relationship
  - ComplexFraction has a Fraction (actually two Fractions)



- Designer 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 in the class definition and before the enclosing class object is constructed

Example: Fraction Class

```
class Fraction {                    // 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 fraction
{
   cout << numerator << "/" << denominator << endl;
}
```


Example: Complex Number Class

```
class ComplexFraction {         // complex number, 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 reIn, int imIn ) : re( reIn, 1 ) , im( imIn, 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.

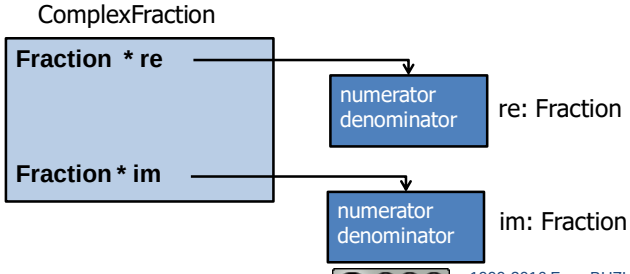
See Example e410.cpp **See Example e411.cpp**

Dynamic Members: Pointers

- Data members of a class may also be pointers to objects (instead of static 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 pointers (addresses) of member objects are included in objects of ComplexFraction
- Member objects re and im must be created separately



Dynamic Members: Pointers

- In this case, enclosing object must either initialize member objects (memory allocation) by itself or get the addresses of its members as parameters
- If memory allocation is performed in the constructor, then these locations will 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
};
```

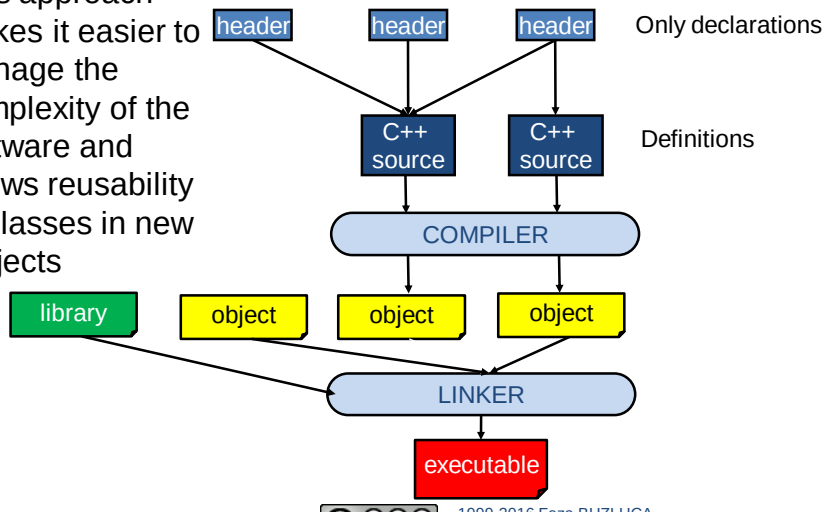
```
// destructor
ComplexFraction::~ComplexFraction()
{
    delete re;
    delete im;
}
```

```
// constructor
ComplexFraction::ComplexFraction( int reIn, int imIn )
{
   re = new Fraction( reIn, 1 );
   im = new Fraction( imIn, 1 );
}
```

See Example e412.cpp

Working with Multiple Files

- It is good programming practice to write each class or a collection of related classes in a separate file
- This approach makes it easier to manage the complexity of the software and allows reusability of classes in new projects



Working with Multiple Files: Separate Compilation

- 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

Working with Multiple Files: Separate Compilation

- These tools use something similar to a makefile, generally called a project file, but the programming environment maintains this file so you do not 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 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