Constructors and The Desctructor

Dr. Sanem Sarıel Talay

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.
- Constructors must be public members of a class.
- There are different types of constructors.
- A constructor that defaults all its arguments or requires no arguments is called default constructor.

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

Constructors with Parameters: Example

```
// A constructor with Parameters
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); //Constructor is called 2 times
   Point *ptr = new Point(10, 50); //Constructor is called once
   Point p3;   // ??
    :
}
```


Constructors with Parameters: Example

```
// A constructor with Parameters
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); //Constructor is called 2 times
   Point *ptr = new Point(10, 50); //Constructor is called once
   Point p3;   // ERROR! There is not a default constructor
    :
}
```

See Example e42.cpp

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 the client of the class can create objects as follows:

- `Point p1(15, 75);`
- `Point p2(100);`
- `Point p3;`

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 the client of the class can create objetcs as follows:

- `Point p1(15, 75);  // x=15, y=75`
- `Point p2(100);  // x=100, y=0`
- `Point p3;  // x=0, y=0`

ISE103 Programming II @ Istanbul Technical University::Department of Computer Engineering, Dr. Sanem Sarıel-Talay

Multiple Constructors

- The rules of function overloading is also valid for constructors. So, a class may have more than one constructor with different signatures.

```
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;
Point p2(30, 10);
Point p3(10);
```


Multiple Constructors

- The rules of function overloading is also valid for constructors. So, a class may have more than one constructor with different signatures.

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

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

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. To invoke a constructor with more than one argument, its name must be given in the list of initial values, to make the program more readable.

```
// Constructor
Point(int x_first,  int y_first = 0) { .... }

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

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

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. To invoke a constructor with more than one arguments, its name must be given in the list of initial values, to make the program more readable.

```
// Constructor
Point(int x_first,  int y_first = 0) { .... }

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

- Three objects of type Point has 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
```

Initializing Arrays of Objects, con't

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

```
Point array[5]= {(10), (20), Point(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() , Point(30,40) };
  // An array with 5 elements. For objects array[2] and
  array[4] the default constructor is invoked.
Point array[5]= { (10), (20), ,Point(30,40)}; // ERROR!
Point array[5]= { (10), (20), () , Point(30,40)}; // ERROR!
```


Constructor Initializers

- Instead of assignment statements, constructors can be used to initialize data members of an object.
- Specially, to assign initial value to a constant member, the only way is to use the constructor.

```
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
};
```

Constructor Initializers, con't

- 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 can be initialized by using constructor initializers

```
class C{
    const int CI;   // constant data member
    int x;          // nonconstant data member
  public:
    C( int a ) : CI(0), x (a)
      { }           // Empty body
};
```

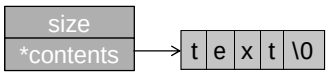

The Destructor

- 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.

Destructor Example

- **Example:** A String class

```
class String{
    int size; // Length (#of chars) of the string
    char *contents; // Contents of the string
 public:
   String(const char *);
   void print();
  ~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.

Destructor Example, con't

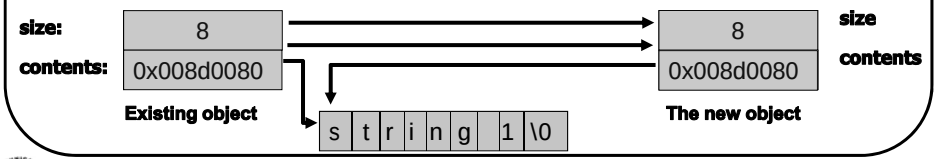
```
//Constructor copies the input char array that terminates
   with a null char to the contents of the string
String::String(const char *in_data){
    size = strlen(in_data);   //strlen is a function of the
                                cstring library
    contents = new char[size +1]; //+1 for null('\0') 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() {
    String string1("string 1");
    String string2("string 2");
    string1.print();
    string2.print();
    return 0;  //destructor is
               called twice
}
```

See Example e43.cpp

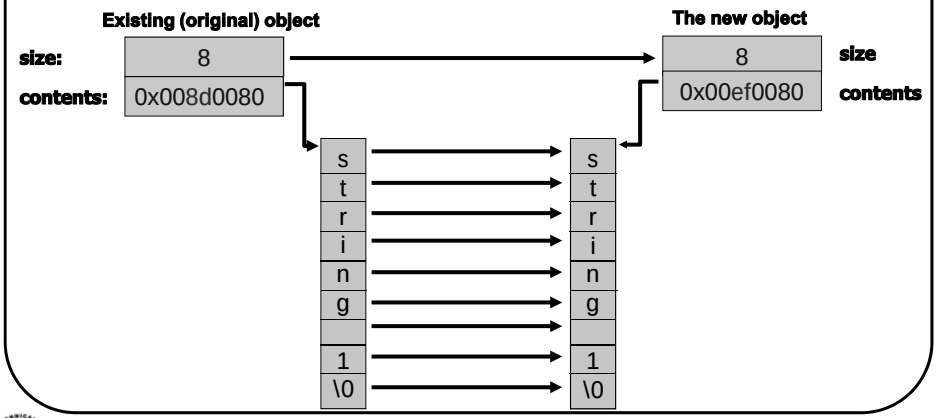
Copy Constructor

- 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 constructor 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 a reference to the object that will be copied into the new object.
- The copy constructor is generated automatically by the compiler if the class author does not 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, this is usually sufficient, but if there is a pointer as a class member so a byte by byte copy would copy only the address and they would both be pointing to the same allocated member. The copy constructor, generated by the compiler for the `String` class will do the following



Copy Constructor, con't

- 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.



Copy Constructor: Example

- **Example:** The copy constructor of the `String` class

```
class String{
    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
    cout<< "Copy Constructor has been invoked" << endl;
    size = object_in.size;
    contents = new char[size + 1];         // +1 for null character
    strcpy(contents, object_in.contents);
}
int main(){
    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 of the object. Only **`const`** functions can operate on **`const`** objects.

```
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
};
```

Constant Objects and Const Member Functions, con't

```
// Constant function: It prints the coordinates on the screen
void Point::print() const{
     cout << "X= " << x << ", Y= " << y << endl;
}
// -------- Main 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;
}
```

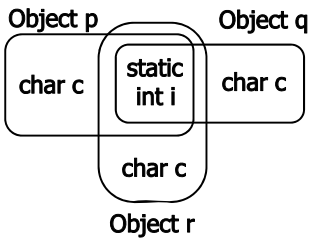
A const method can invoke only other const methods, because a const method is not allowed to alter an object's state either directly or indirectly, that is, by invoking some non-const method.

See Example e45.cpp

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 Class Members, con't

- 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) in the scope of the file.

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

Passing Objects Reference to Functions

- 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`**.

```
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

Avoiding Temporary Objects

- In the previous example, within the add function a temporary object 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.

See Example e49.cpp

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

Nesting Objects

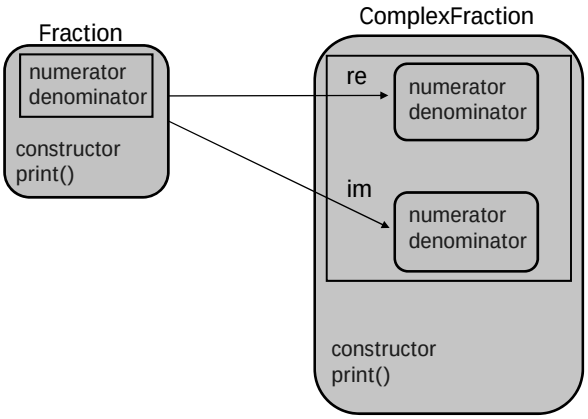
- A class may include objects of other classes as its data members.
- In the example below, a class is designed (`ComplexFraction`) to define complex numbers. The data members of this class are fractions which are objects of another class (`Fraction`). `ComplexFraction:`

$$z = \frac{a}{b} + \frac{c}{d}\, i$$

re:Fraction im:Fraction

- 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.

Nesting Objects: Example



Nesting Objects: Example, con't

Fraction class:

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


Nesting Objects: Example, con't

```
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){
      :
 }
```

Data members are initialized

Nesting Objects: Example, con't

```
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;
}
```

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

See Example e410.cpp

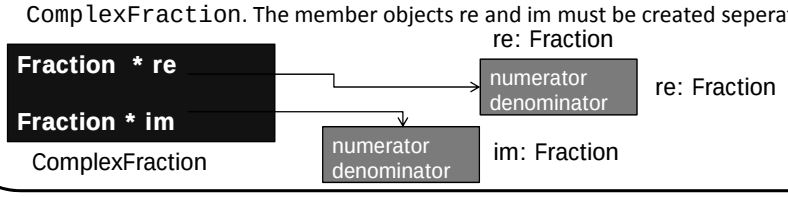
See Example e411.cpp

Dynamic 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 seperately.



Dynamic Members: Example

- In this case, the 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 should be released in the destructor.

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


Dynamic Members: Example, con't

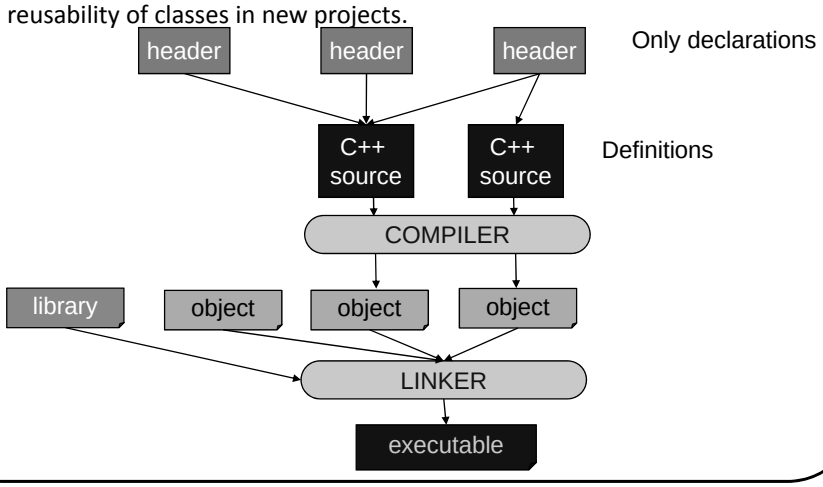
```
// Constructor
ComplexFraction::ComplexFraction(int re_in, int im_in){
    re= new Fraction(re_in,1);
    im= new Fraction(im_in,1);
}

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

See Example e412.cpp

Multiple Code 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.



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. 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).

See Example e413.zip

- 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.

Header File Standards

- Header files contain the class declarations
- Definitions of the methods are written in the corresponding .cpp files
- For the multiple file compilation case, to include each header only once define a preprocessor flag.
  - If the flag isn't set, the file wasn't included and you should set the flag (so the class can't get re-declared) and declare the class.
  - If the flag was set then that type has already been declared so you should just ignore the code that declares it.

Header File Standards, con't

- ```
  #ifndef HEADER_FLAG
  #define HEADER_FLAG
  // Type class declaration here...
  #endif //HEADER_FLAG
  ```

- The name HEADER_FLAG can be any unique name, but a reliable standard to follow is to capitalize the name of the header file and replace periods with underscores (leading underscores, however, are reserved for system names).
  - Example: FRACTION_H
  ```