Lecture 4

Constructors and Destructors

Outline

- Default Constructor
- Constructors with Parameters
- Constructor Initializers
- Destructors
- Copy Constructor
- Constant Objects
- Passing Objects to Functions
- Nesting Objects

Initializing Class Objects: Constructor Functions

- Initialization of every object can be done by providing a special **member function** called the **constructor function**.

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

- There can be **more than one constructor** of the same class.

- The constructor functions are used for many purposes, such as assigning initial values to data members, etc.

Constructors

- The constructor function can take parameters,
  but it can not have a return value (even not void).

- The constructor must have the same name as the class itself.

- There are three types of constructors:
  - **Default constructor**
  - **Parametered constructor**
  - **Copy constructor**

Default Constructor

Default constructor requires **no parameters**.

```
class Point
{
   int x, y;
 public:
   Point () { // default constructor
       x=0;
       y=0;
    };
   bool move(int, int);
   void print();
};
```

Initialization of member data during declaration is not allowed.

```
class Point
{
   int x=0, y=0; // Compiler error!
   .....
};
```

```
int main() {
   Point  p1, p2;       // Default constructor is called (invoked) 2 times.

   Point *ptr;          // ptr is not an object, constructor is NOT called yet.
   ptr = new Point; // Object is created, also the default constructor is called now.
}
```

Constructors with Parameters

- Users of the class (client programmers) can supply constructors with necessary argument (parameter) values.

- A class may have more than one constructor with different type of input parameters (**Constructor overloading**).

- The first constructor is the **default constructor.**
- The second constructor is the **parametered constructor.**

```
class Point
{
   int x, y;
 public:
   Point ();
   Point (int, int);
   bool move (int, int);
   void print ();
};
```

```
// Constructor with two parameters
Point : : Point (int  x_in,   int  y_in)
{
    || Point may not have negative coordinates
    if ( x_in < 0 )   || If given value is negative
            x = 0;    || Assigns zero to x
    else    x = x_in;

    if ( y_in < 0 )   || If given value is negative
            y = 0;    || Assigns zero to y
    else    y = y_in;
}
```

Main program

```
int main()
{
    Point  p1 (20, 100),  p2 (-10, 45);       // Constructor is called 2 times

    Point *ptr = new  Point (30, 50);        // Constructor is called once

    Point p3;          //ERROR! There is not a default constructor body

    Point  p4 (10);    //ERROR! There isn't a constructor with one parameter
    ......
}
```

- To prevent the first compiler error, the following default constructor should be defined in class codes.
- There are no code statements inside the block parantheses.

```
Point ()  {}
```

Default Values of Constructor Parameters

- Parameters of constructors may have default values.
- The following constructor can be called with one, two, or no arguments.

```
class Point
{
  public:
     Point (int =0,  int =0);   // Prototype of constructor
     // Default values of parameters are zero.
     ……
};
```

```
Point :: Point (int  x_in,  int  y_in)
{
    if ( x_in < 0 )
            x = 0;
    else    x = x_in;

    if ( y_in < 0 )
            y = 0;
    else     y = y_in;
}
```

```
int main()
{
  Point  p1 (15, 75);    // x=15,  y=75
  Point  p2 (100);       // x=100, y=0
  Point  p3;             // x=0, y=0
}
```

Initializing Arrays of Objects

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

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

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

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

Objects:	Arguments:	
array[0]	x_in = 10	y_in = 0
array[1]	x_in = 20	y_in = 0
array[2]	x_in = 30	y_in = 40

- Alternative syntax : The following makes the program more readable.

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

Constructor Initializers

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

```
class A {
    const  int  n;    // constant data member
    int x;            // nonconstant data member
  public:
    A( ) {            // constructor function
        x = 0;
        n = 0;        // ERROR!   n is constant
    }
};
```

The example below is not correct, either:

```
class A {
    const int n = 0 ;   // ERROR!
    int x;
};
```

Example: Constructor initializer in Default constructor

- For constant data members, a **constructor initializer** must be written.

```
class A {
    const int n;
    int x;

  Public:

    A() : n (0)   // constructor initializer
    // initial value of n is assigned to zero
    {
        x = 0;
     } // end of constructor

};
```

Example: Constructor initializer in Parametered constructor

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

```
class A {
    const int n;
    int x;

  public:
    A (int num1, int num2) : n (num1) , x (num2)  // Constructor initializers
      { }   // Codes section of constructor can be empty
};
```

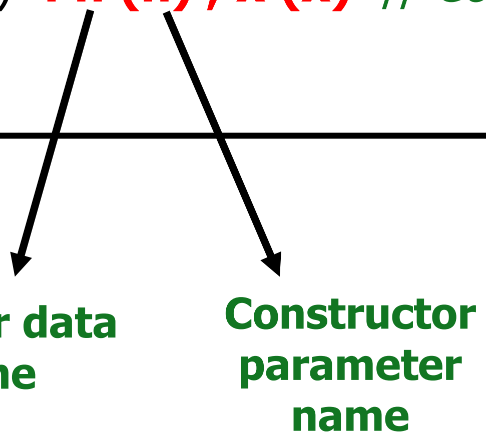
- Two objects are defined in main.

```
int main()
{
    A   obj1 (-5,   7);
    A   obj2 (0,   18);
}
```

Example: Using same names for constructor parameters and member data

- Constructor parameter names and member data names can be the same.

```
class A
{
    const int n;
    int x;

  public:
    A (const  int n,  int x)  : n (n) , x (x)  // Constructor initializers
      { }
};
```



Member data name

Constructor parameter name

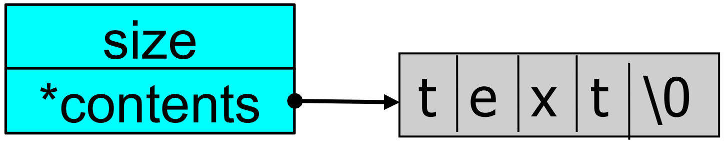
Destructor Function

- The destructor function is called automatically,
  - When each of the objects goes out of scope, or
  - When a dynamic object is deleted from memory by using the delete operator.
- A destructor is defined as having the same name as the class, with a tilde **(~)** symbol preceded to class name.
- A destructor has no return type and receives no parameters.
- A class may have **only one** destructor.

Example : String class

The following is a programmer-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();              // Member function
  ~String();                  // Destructor
};
```



C++ already has a built-in **string** class (written as lowercase).
Programmers don't need to write their own String class.

Parametered constructor of String class

Parametered constructor :
Copies the input character array to the contents of the String.

```
String :: String (const  char *   in_data)
{
   size = strlen (in_data);
   // strlen is a built-in function of the cstring library

   contents = new  char [size + 1];
   // +1 is for the null ( '\0' ) character

   strcpy (contents,  in_data);
   // strcpy is a built-in function of the cstring library
   // input data is copied to the contents member
 }
```

Main program

```
// Destructor
// Memory pointed by contents is deleted

String :: ~ String ()
{
     delete [] contents;
}
```

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

```
int main() {
    String  s1 ("ABC");
    String  s2 ("DEFG");
    // Constructor is called two times

    s1.print();
    s2.print();

     // At the end of program,
     // destructor is called two times
}
```

Copy Constructor

- Copy constructor is used to copy the members of an object to a new object.

- The type of its input parameter is a ***reference*** to objects of the same type.

- The input parameter is the object that will be copied into the new object.

- There are two types of Copy constructor.
  - **Compiler-provided**
  - **User-written**

Compiler-provided Copy Constructor

- There is a compiler-provided default copy constructor.

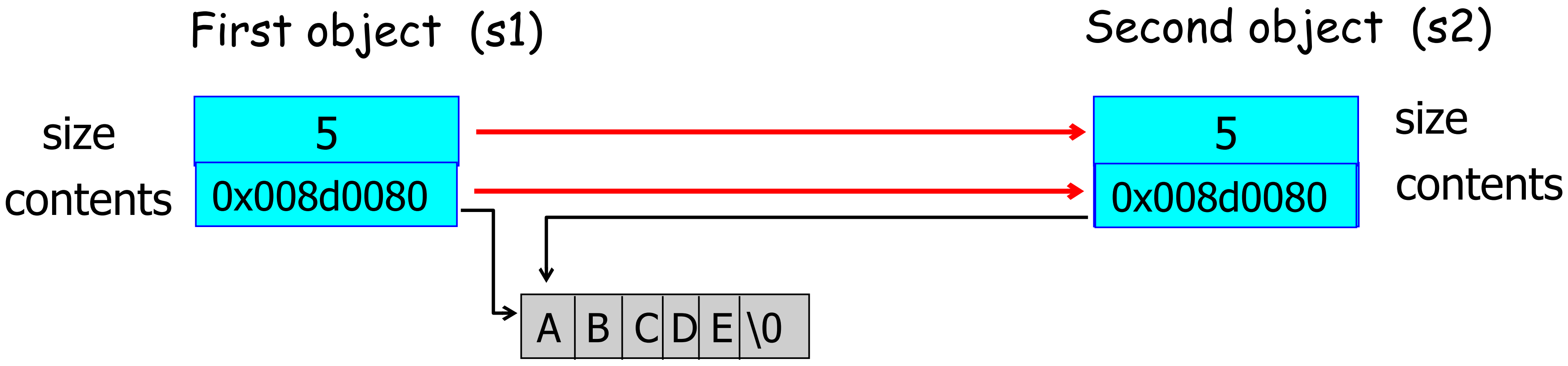
- Compiler-provided copy constructor will simply copy the contents of the original into the new object, as a **byte-by-byte copy**.

- If there is a **pointer** as a class member, so a byte-by-byte copy would copy only the pointer from one to the other.

- In result, they would both be pointing to the **same** allocated member data.

Compiler-provided Copy Constructor

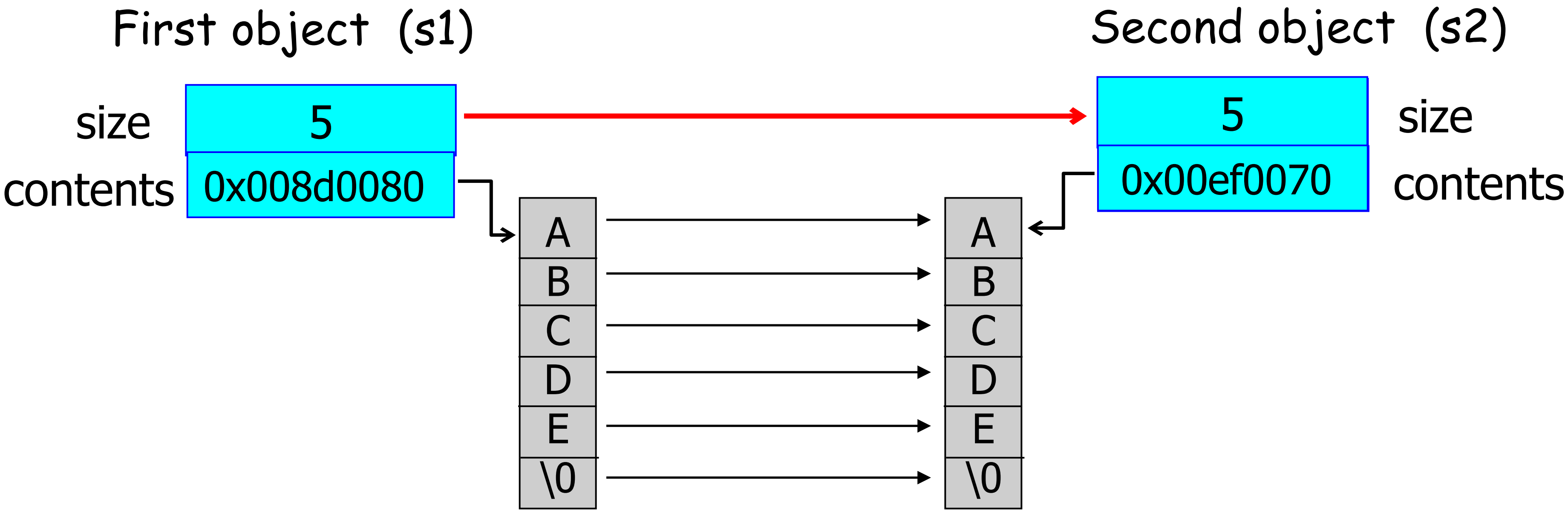
```
int main()
{
    String  s1 ("ABCDE");  // Normal constructor is called
    s1.print();

    String  s2 = s1;
    // Compiler-provided copy constructor is called in assignment
}
```



After copying, two objects are sharing the same contents address. Data are not duplicated.

User-written Copy Constructor

- The default copy constructor, generated by the compiler can not duplicate the data in the memory locations pointed by the **member pointers**.
- Therefore, programmer must write his own copy constructor function.



After copying, two objects have different contents address, with duplicated data.

Example: User-written copy constructor

```
class String {                      // User defined String class
    int size;
    char *contents;

 public:
    String (const char *);           // Normal Constructor
    String (const  String &);        // Copy Constructor (user-written)
    void print();                    // Prints the string on screen
    ~String();                      // Destructor
};
```

```
// Copy Constructor (user-written)

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

Main program

```
int main()                    // Test program
{
    String  s1 ("ABCDE");

    s1.print();

    String  s2 = s1;   // Copy constructor is invoked (user-written)

    String  s3 (s1);    // Copy constructor is invoked (user-written)

     s2.print();

     s3.print();
}
```

Const Member Function

- Programmer may declare some member **functions** as **const**, which do not modify any data of the object.

```
class Point
{
   int x, y;

 public:
   Point (int, int);
   bool move(int, int);
   void print() const;      // Constant function
};
```

```
// Constant function
void  Point :: print () const
{
    cout << "X= " << x << ", Y= " << y << endl;
}
```

Constant Object

- Programmer may use the keyword **const** to specify that an **object** is not modifiable.

```
int main()      // Test program
{
    const  Point  A(10, 20);     // A is a constant object
    A.print();                   // OK. Const function operates on const object
    A.move(30, 15);              // ERROR! Non-const function on const object
                                 // A is not modifiable

    Point  B(0, 50);             // B is a non-constant object
    B.print();                   // OK
    B.move(100, 45);             // OK
}
```

Static Class Members

- In certain cases, <u>only one copy</u> of a particular data member should be shared by all objects of a class.
- A static data member is used for this reason.
- Static data members exist even no objects of that class exist.
- To access public static data without an object, use the class name and the scope operator. For example **A :: x = 5;**

```
class A {
  public:
      static int x;
};

int  A :: x ;   //Required definition

int main() {
    A   p, q, r;
    A :: x  =  5;
    cout <<  A :: x ;
    cout <<  p.x ;
    cout <<  q.x ;
    cout <<  r.x ;
}
```

- Objects p,q,r share the same member data x.
- Program displays the same outputs.

Passing Objects to Functions as Arguments

- As a general rule, when calling a function, objects should be passed **by-reference**.
- In this way, an unnecessary copy of an object is not passed as argument.
- Also to prevent the function from modifying the original object, we make the parameter a **const reference.**

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

```
int main() {
  ComplexT   z1(1, 2) ,    z2(0.5, -1) ,    z3;
  // Three objects are defined
  z3 = z1.add( z2 );   // pass z2 object as argument
  z3.print();
}
```

Avoiding Temporary Objects within Functions

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

- Because of the temporary local object, constructor and destructor are called.

- Avoiding a local temporary object within the add function saves 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, then whole object is returned
}
```

Nesting Objects :
Objects as Members of Other Classes

- A class may include objects of other classes as its data members.
- In the following example, School class includes an array of Student class objects.

```
class School
{
  public:
    Student  st [200];

 School(); //constructor
 void print_school();
}
```

```
class Student
{
 public:
    int      ID;
    string  firstname;
    string  lastname;

    Student (int, string, string); //constructor
    void print_student();
}
```

Student class Member Functions

```
// Constructor
Student :: Student  (int  ID,
                     string  fname,
                     string  lname)
{
   this->ID    =  ID;
   firstname   =  fname;
   lastname    =  lname;
 }
```

```
void  Student :: print_student ()
{
   cout << ID << " "
        << firstname << "  "
        << lastname << endl;
}
```

School class Member Functions

```
//  Default Constructor
School :: School ()
{
   for (int  i=0;  i < 200;  i++)
   {
       st [i].ID            =  0 ;
       st [i].firstname  =  "" ;
       st [i].lastname   =  "" ;
   };
}
```

```
School :: print_school ()
{
   for (int  i=0;  i < 200;  i++)
   {
       if ( st [i].ID != 0] )
           st [i].print_student();
   }
}
```

Calling print function of st[i]

Main Program

```
int main()
{
  School  Sch; //Definition invokes the constructor of School

  // Add 3 students with constructors parameters
  Sch.st [0]  =  Student (111, "AAA", "BBB");
  Sch.st [1]  =  Student (222, "CCC", "DDD");
  Sch.st [2]  =  Student (333, "EEE", "FFF");

  Sch.print_school ();     //Calling print function of school Sch
}
```