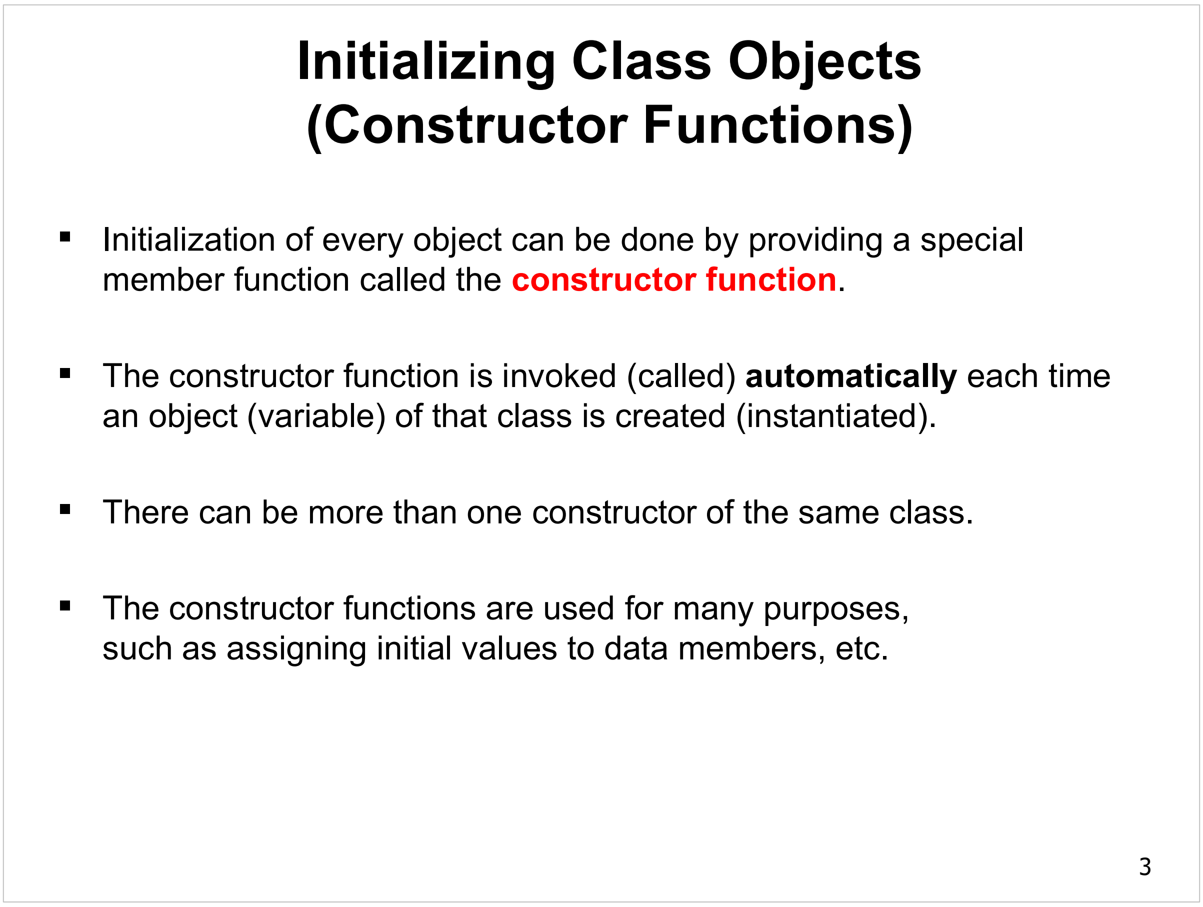
Lecture 4

Constructors and
Destructors

Topics

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



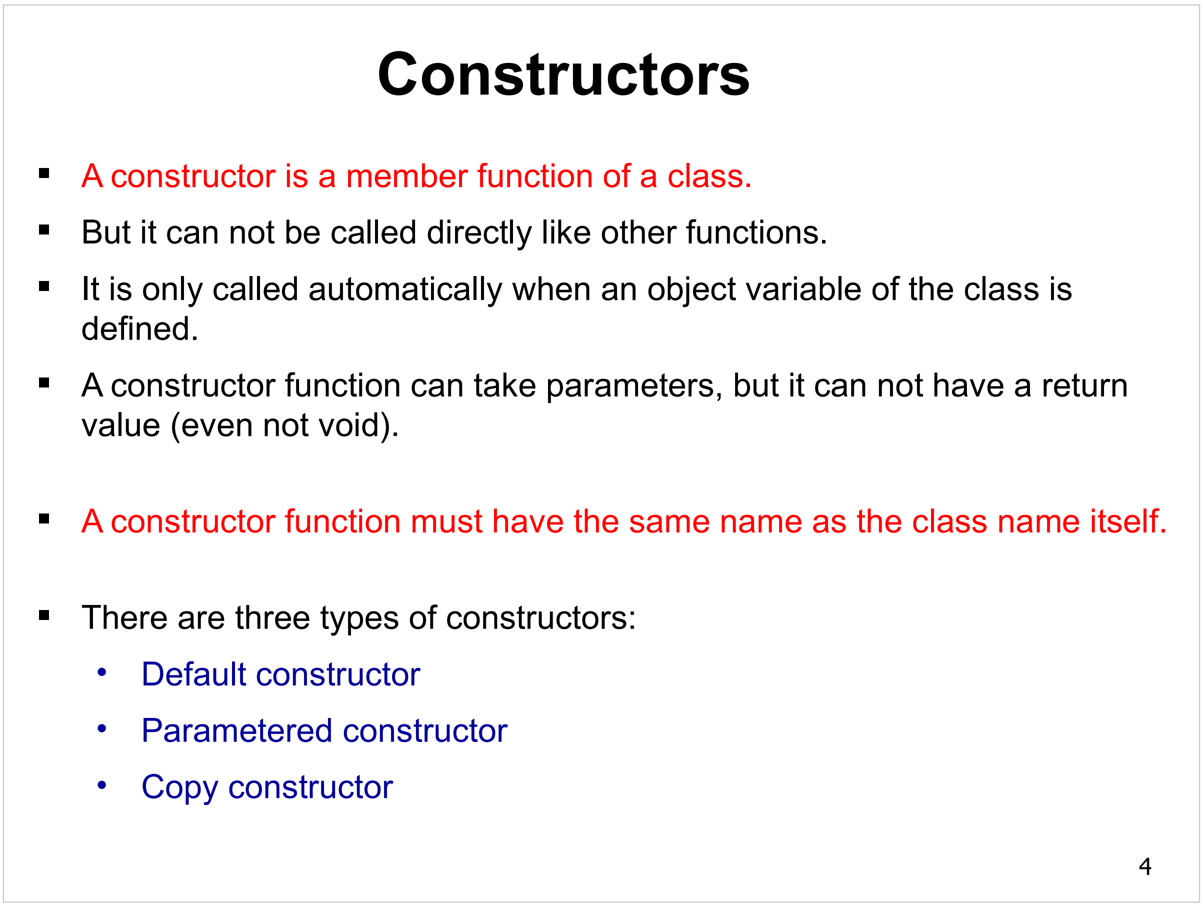
Initializing Class Objects (Constructor Functions)

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

- The constructor function is invoked (called) **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.

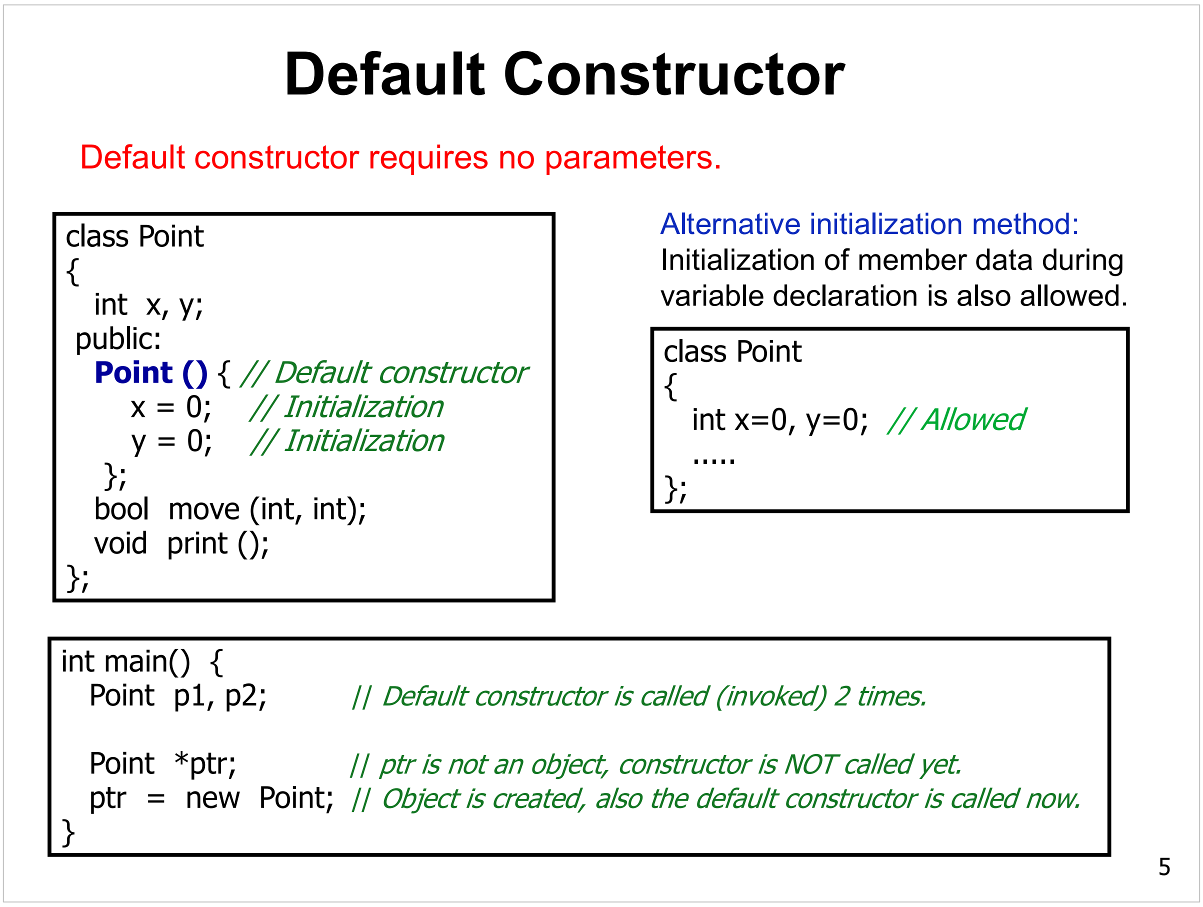


Constructors

- A constructor is a member function of a class.
- But it can not be called directly like other functions.
- It is only called automatically when an object variable of the class is defined.
- A constructor function can take parameters, but it can not have a return value (even not void).

- A constructor function must have the same name as the class name itself.

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



Default Constructor

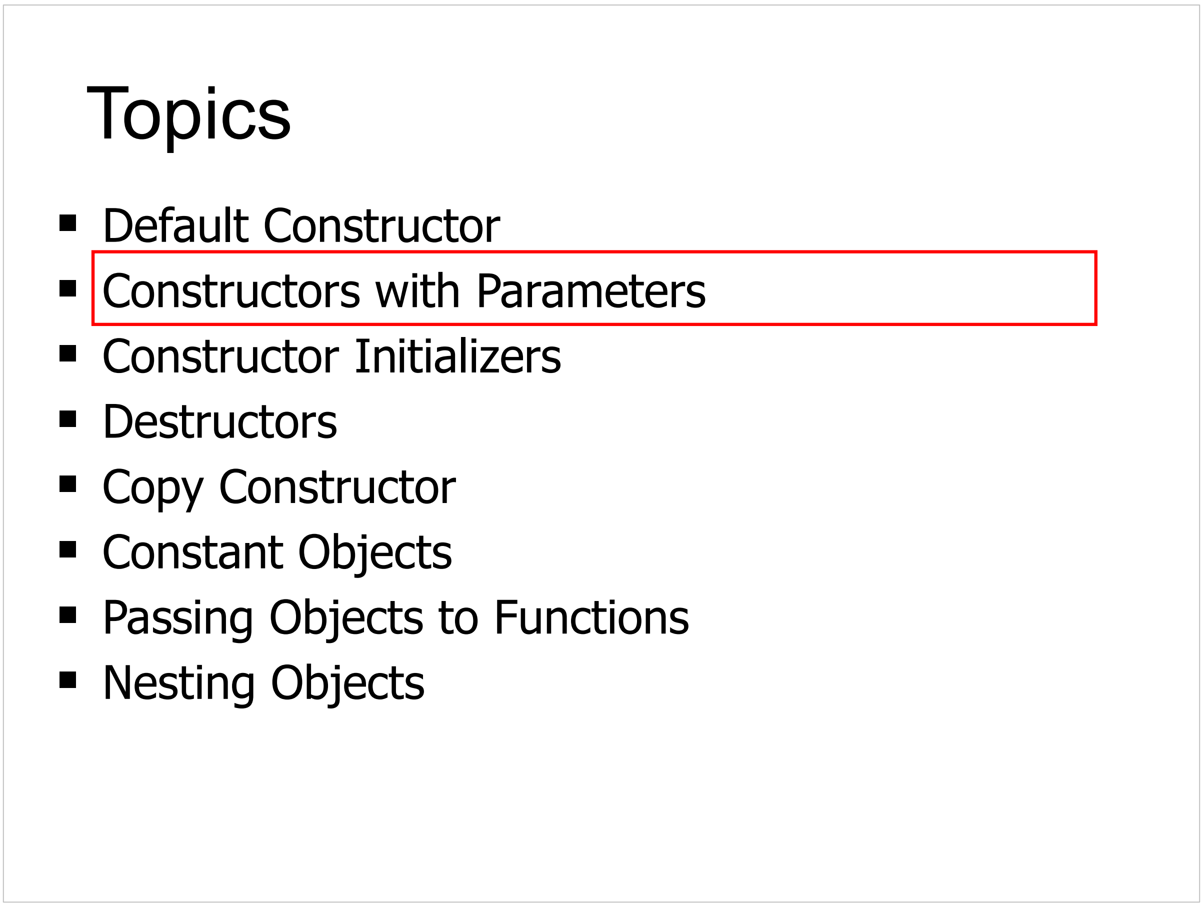
Default constructor requires no parameters.

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

Alternative initialization method:
Initialization of member data during variable declaration is also allowed.

```
class Point
{
   int x=0, y=0;  // Allowed
   .....
};
```

```
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.
}
```



Topics

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

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 default constructor.
- The second constructor is parametered constructor.

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

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

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

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;          //Compiler error. There is not a default constructor code block

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

Default constructor with empty code block

- To prevent the first compiler error in main program above, the following **default constructor** should be defined in the Point class.
- There are no code statements inside the block paranthesis of default constructor.
- Empty code block is written as **{ }**.

```
class Point
{
   int x, y;
 public:
   Point () {}              // Default constructor with empty code block
   Point (int, int);        // Parametered constructor prototype
   bool  move (int, int);   // Prototype
   void  print ();          // Prototype
};
```

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  xn,  int  yn)
{
    if ( xn < 0 )
            x = 0;
    else    x = xn;

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

```
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]	xn = 10	yn = **0**
array[1]	xn = 20	yn = **0**
array[2]	xn = 30	yn = 40

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

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

Topics

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

Constructor Initializers

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

```
class A {
    const  int  n;   // constant data member
    int x;           // non-constant data member
  public:
    A ( ) {          // constructor function
        x = 0;       // initialization
        n = 0;       // Compiler error (n is defined as constant)
    }
};
```

- The variables can be initialized during their declarations.

```
class A {
    const int n = 0 ;   // OK
    int x = 0;          // OK
     . . . .
};
```

Example: Constructor initializer in Default constructor

- For constant data members, a **constructor initializer** can be written.
- The colon symbol (:) is used as constructor initializer.

```
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 initializers in Parametered constructor

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

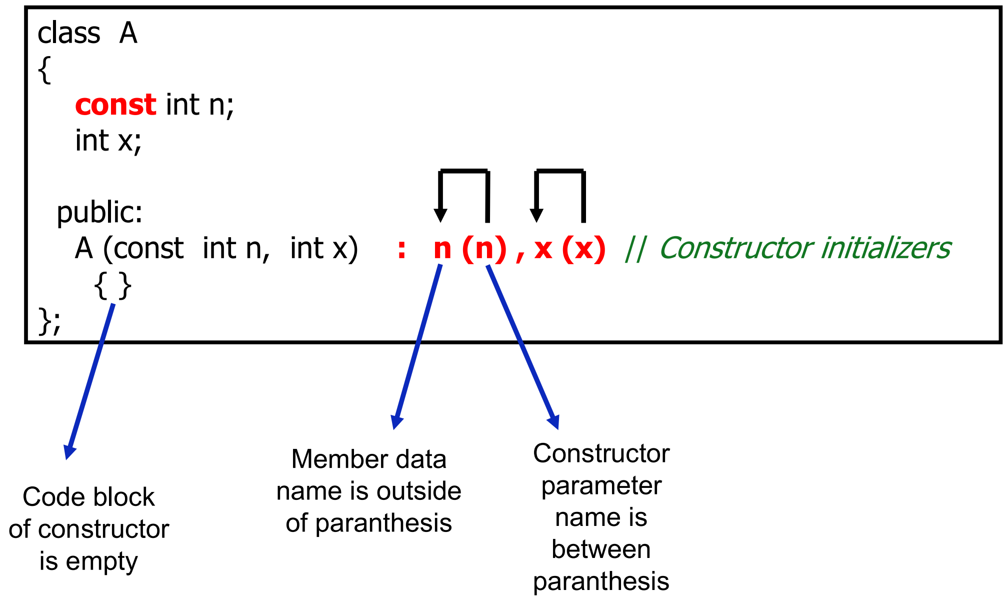
```
class A {
    const int n;
    int x;

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

- Two objects are defined in main.

```
int main()
{
    A   a1  (-5,   7);
    A   a2  (0,   18);
}
```

Example: Using same names for constructor parameters and for 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
      { }
};
```

Code block of constructor is empty

Member data name is outside of paranthesis

Constructor parameter name is between paranthesis

Topics

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

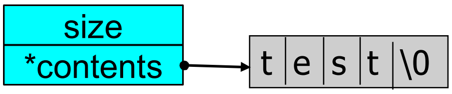
Destructor Function

- 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 *);   // Parametered 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

<u>Parametered constructor :</u>
Function copies the input character array (data) to the contents of the String.

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

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

   strcpy (contents,  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
}
```

Screen output

```
ABC    3
DEFG   4
```

Topics

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

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.

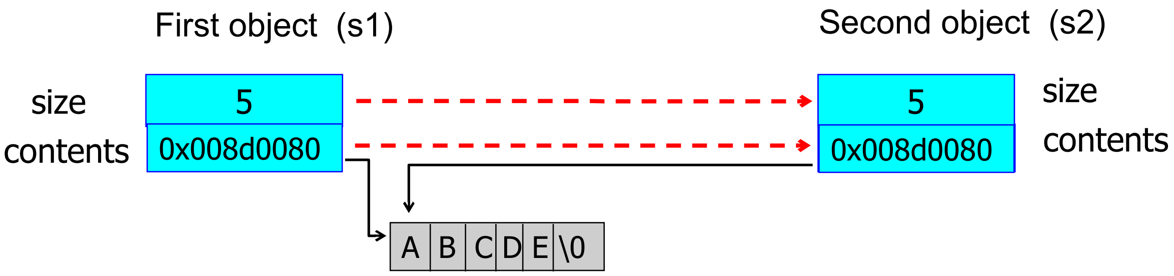
- Compiler-provided copy constructor will simply copy the contents of the original into the new object, as a **byte-by-byte** copy.

- If there is a **pointer** as a class member, the 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");  // Parametered constructor is called
   s1 . print();

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



First object (s1)

Second object (s2)

size

contents

5

0x008d0080

5

0x008d0080

size

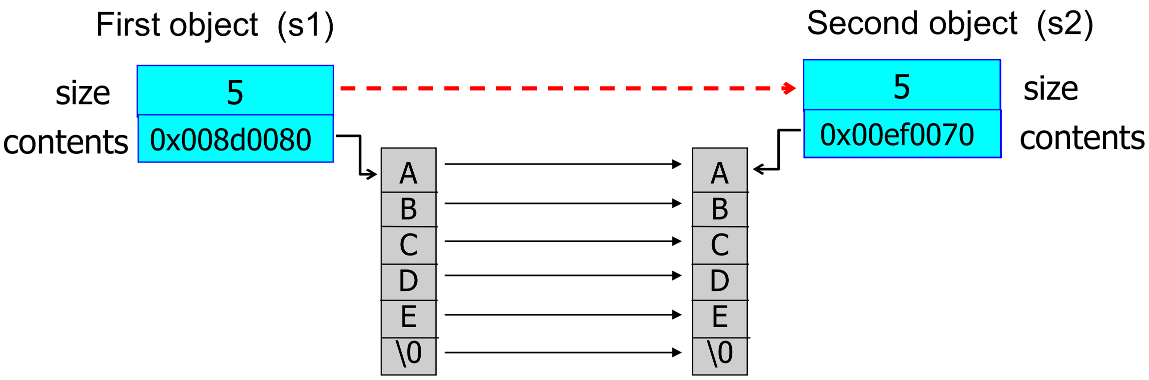
contents

A B C D E \0

After copying, two objects are
sharing the same memory location.
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.
- User-written copy constructor is useful, if data duplication is required.



First object (s1)

Second object (s2)

size

contents

5

0x008d0080

5

0x00ef0070

size

contents

A B C D E \0

A B C D E \0

After copying, two objects have
different memory locations.
Datas are duplicated.

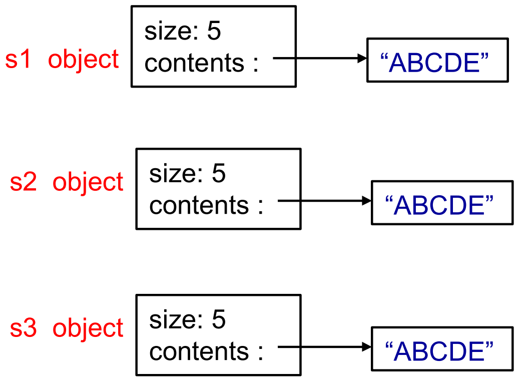
Example: User-written copy constructor

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

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

Main program

```
int main()                    // Test program
{
    String   s1 ("ABCDE"); // Parametered constructor is invoked
    String   s2  =  s1;     // Copy constructor is invoked (user-written)
    String   s3 (s1);       // Copy constructor is invoked (user-written)

    s1 . print();
    s2 . print();
    s3 . print();
}
```



Screen output

```
ABCDE     5
ABCDE     5
ABCDE     5
```

Topics

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

Const Member Function

- Programmer may declare some member **functions** of a class as **constant.**
- A const function does 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;
     x=0; y=0;  // Compiler errors
}
```

Constant Object

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

```
int main()
{
   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 some 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.
- 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;**
- Static variable does not mean that its data is constant.

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

//Required definition in global scope
int   A :: x  =  5;

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

- Objects p, q, r share the same member data x.
- Program displays the same outputs (data value 6) four times.

Screen output

```
6
6
6
6
```

Topics

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

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 ();
}
```

Screen output

```
1.5     1
```

Avoiding Temporary Objects within Functions

- In the previous example, within the **add member function**, a temporary local object (result variable) 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
}
```

Topics

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

Nesting Objects

(Objects as Members of Other Classes)

- A class may include objects of other classes as its data members.
- 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 ()
{
   cout << "List of students : \n";
   for (int  i=0;  i < 200;  i++)
   {
      if ( st [i] . ID != 0] )
         st [i] . print_student ();
   }
}
```

Calling the print member function of Student class

Main Program

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

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

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

Screen output

```
List of students :
111  AAA  BBB
222  CCC  DDD
333  EEE  FFF
```