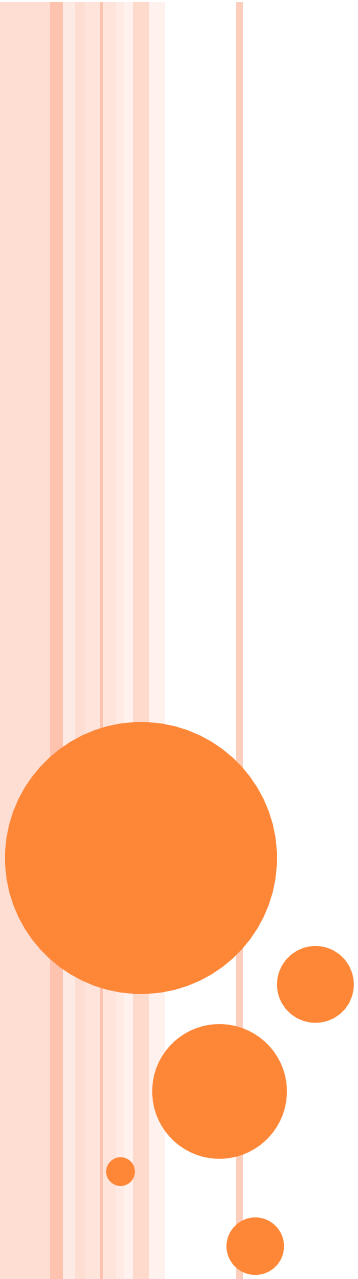


The Big Three

CS A250 – C++ Programming II

The Big Three

- The so-called **Big Three** are:
  - **Copy constructor**
  - **Overloaded assignment operator**
  - **Destructor**

- If any of these is missing from your class, the compiler will create it
  - **But** they might **not** behave as you expected when you have **dynamic variables.**
  - Therefore, if you have **pointers** and the **new** operator, you need to **implement the big three**.

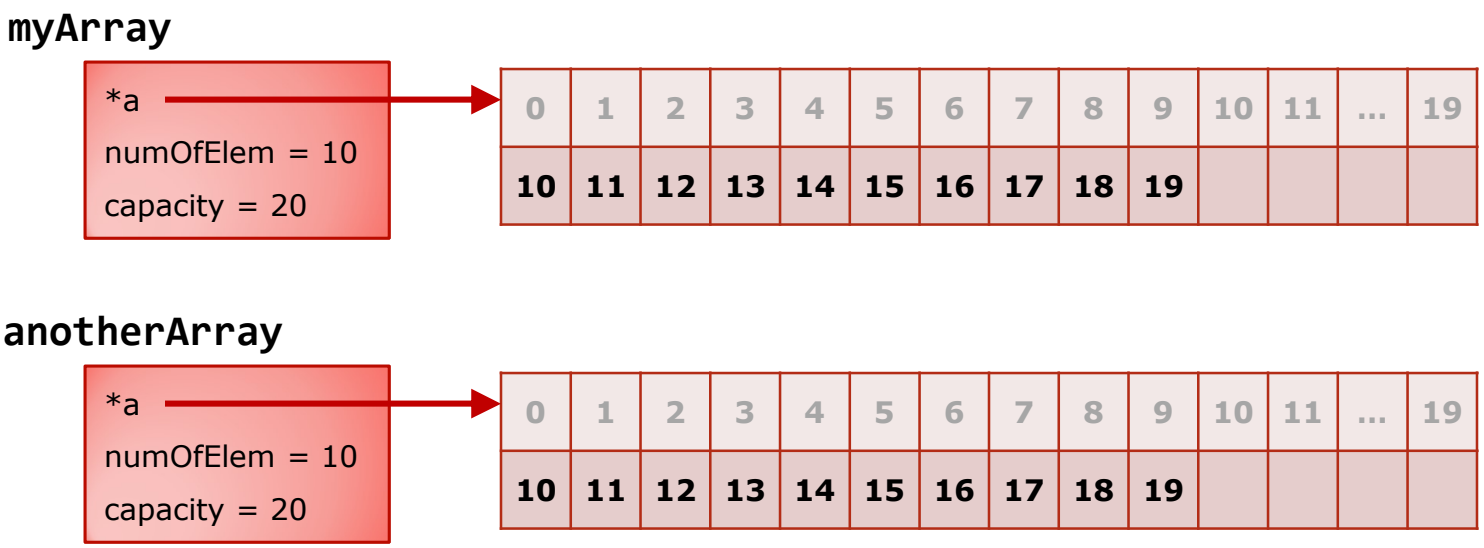
Copy Constructor

Copy Constructor

- A **copy constructor** is an **overloaded** constructor that **creates a new object** that is **identical** to the object that is passed as parameter.

```
DArray myArray(20);

for (int i = 10; i < 20; ++i)
        myArray.addElement(i);

DArray anotherArray(myArray);

                // anotherArray is initialized
                // by the copy constructor
```

```
DArray myArray(20);

for (int i = 10; i < 20; ++i)
        myArray.addElement(i);

DArray anotherArray(myArray);
                // anotherArray is initialized
                // by the copy constructor
```



COPY CONSTRUCTOR (CONT.)

- This type of constructor has a **call-by-reference parameter** that is of the **same type** of the class
  - The parameter **must** be passed by **reference**
  - The parameter **must** have a **const** modifier

```
DArray(const DArray& anotherArray);
```

IMPLEMENTATION

```
DArray::DArray()
{
   capacity = CAPACITY;
   numOfElem = 0;
   a = new int[capacity];
}
```

Default constructor of the class **DArray**

IMPLEMENTATION (CONT.)

```
DArray::DArray()
{
    capacity = CAPACITY;
    numOfElem = 0;
    a = new int[capacity];
}
```

Default constructor of the class **DArray**

Copy constructor

of the class **DArray**

```
DArray::DArray(const DArray& otherArray)
{
        capacity = otherArray.capacity;
        numOfElem = otherArray.numOfElem;
        a = new int[capacity];

        for (int i = 0; i < numOfElem; ++i)
                a[i] = otherArray.a[i];
}
```

IMPLEMENTATION (CONT.)

```
DArray::DArray()
{
    capacity = CAPACITY;
    numOfElem = 0;
    a = new int[capacity];
}
```

Default constructor of the class **DArray**

You need to initialize **ALL** member variables for the new object, **AND** then copy **all** the elements.

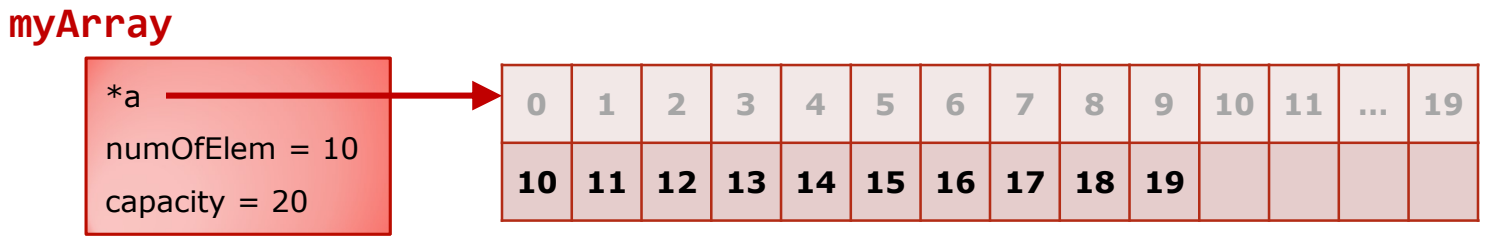
```
DArray::DArray(const DArray& otherArray)
{
        capacity = otherArray.capacity;
        numOfElem = otherArray.numOfElem;
        a = new int[capacity];

        for (int i = 0; i < numOfElem; ++i)
                a[i] = otherArray.a[i];
}
```

Copy constructor

of the class **DArray**

What If…

- What if you do **not** implement the **copy constructor** in a class that uses **pointers**?
  - Assume you have object **myArray**:

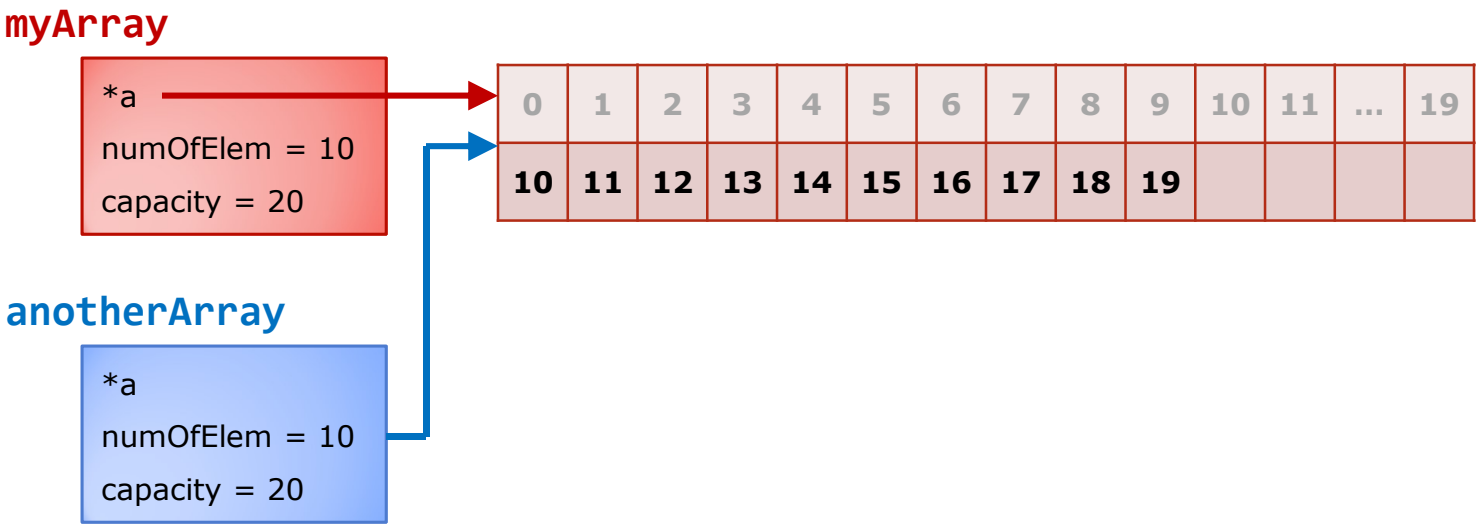


  - You would like to create a new object named **anotherArray**, but you have **not** implemented the copy constructor:

```
DArray anotherArray(myArray);
```

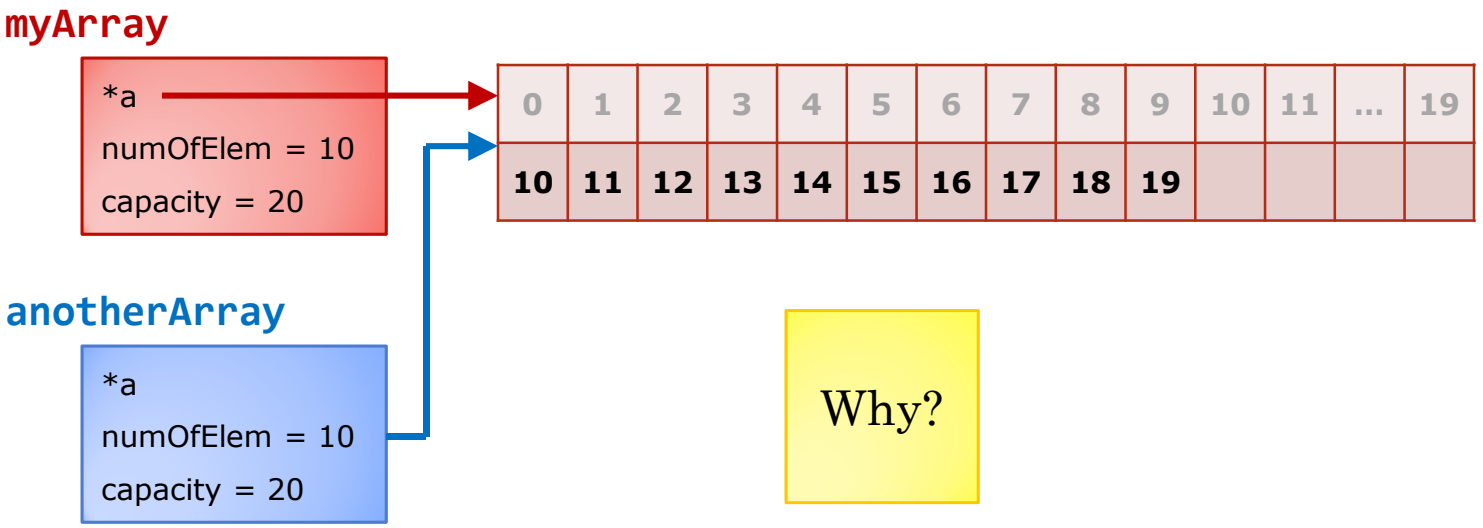
What If…

- The compiler will copy your object, BUT it will make what is called a **shallow copy**.



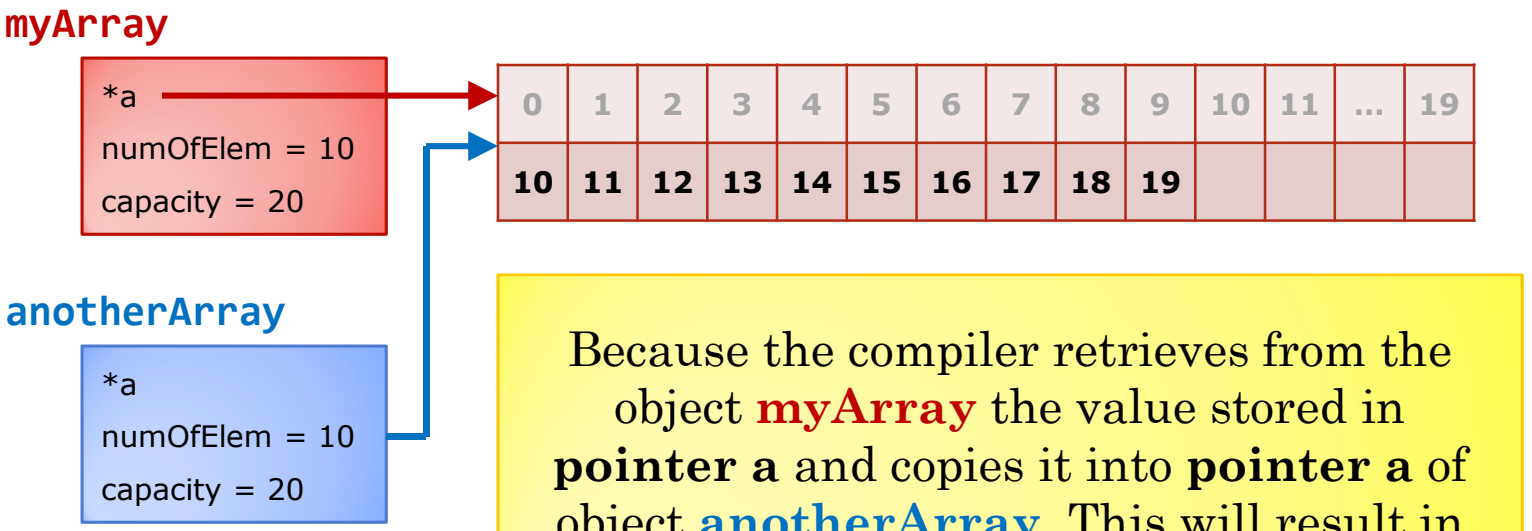
What If…

- The compiler will copy your object, BUT it will make what is called a **shallow copy**.



What If…

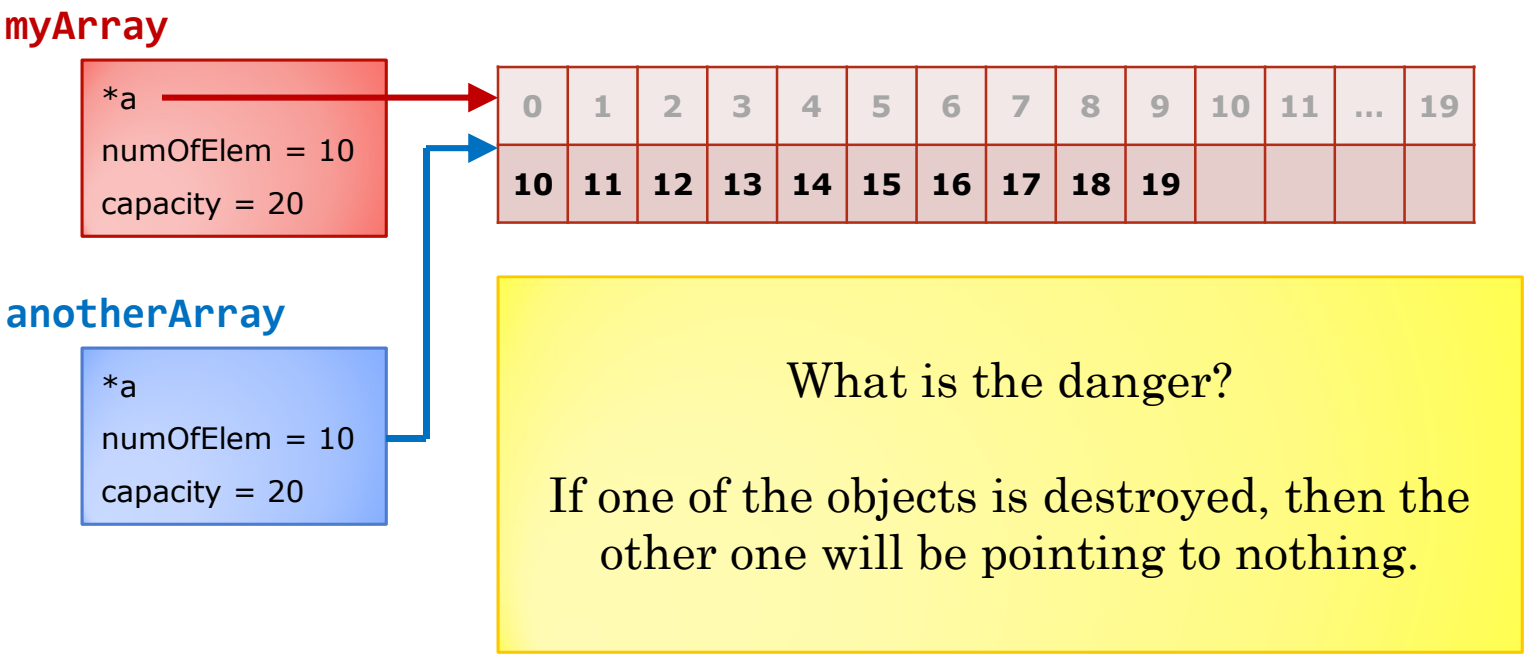
- The compiler will copy your object, BUT it will make what is called a **shallow copy**.



Because the compiler retrieves from the object **myArray** the value stored in **pointer a** and copies it into **pointer a** of object **anotherArray**. This will result in both pointers pointing to the same array.

What If…

- The compiler will copy your object, BUT it will make what is called a **shallow copy**.



What is the danger?

If one of the objects is destroyed, then the other one will be pointing to nothing.

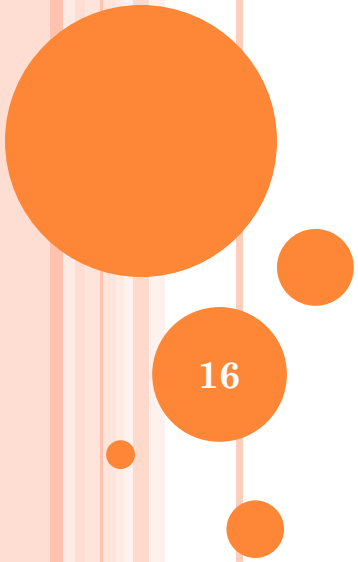
COPY CONSTRUCTOR (CONT.)

- The copy constructor is called ***automatically*** when
  - A function returns a **value** of the class type

```
DArray func(…); // Function returns a copy
```

  - An argument is plugged in for a **call-by-value parameter** of the class type.

```
void func(DArray anotherArray);
// Parameter is passed by value
```



Overloaded Assignment Operator

Assignment Operator

- A **default** **overloaded assignment** operator is available, **BUT**
  - If **dynamic variables** are used, you must overload the **assignment operator**.

- **Overloading the assignment operator** also allows for some specialized uses
  - Example:

```
object1 = object2 = object3;
```

- Function **must** be a **member of the class**
  - **Cannot** be a non-member and **cannot** be a friend.

Preventing Self Assignment

- It is important to ***prevent*** **self assignment**
  - This is to avoid that the **operator=** deletes the dynamic memory associated with the object before the assignment is completed.
  - This would lead to "*fatal runtime errors*".

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
              if (capacity != rightSide.capacity)
              {
                   delete [ ] a;  //release space
                   a = new int[rightSide.capacity]; //re-create array
                   capacity = rightSide.capacity;
              }
              for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                    a[i] = rightSide.a[i];
              noOfElem = rightSide.noOfElem ;
     }
     else
              cerr << "Attempted assignment to itself.";

     return *this;
}
```

Let's look at this in detail…

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
    if                                  id self assignment
    {
                                        Side.capacity)
            {
                delete [ ] a;  //release space
                a = new int[rightSide.capacity]; //re-create array
                capacity = rightSide.capacity;
            }
            for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                 a[i] = rightSide.a[i];
            noOfElem = rightSide.noOfElem;
    }
    else
            cerr << "Attempted assignment to itself.";

    return *this;
}
```

We return a **reference** to the object.

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
              if (capacity != r
              {
                   delete [ ] a;
                   a = new int[r                    create array
                   capacity = ri
              }
              for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                    a[i] = rightSide.a[i];
              noOfElem = rightSide.noOfElem;
     }
     else
              cerr << "Attempted assignment to itself.";

     return *this;
}
```

Make sure the calling object and the parameter are **not** the same object.

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
              if (capacity != rightSide.capacity)
              {
                   delete [ ] a;  //release space
                   a = new int[rightSide.capacity]; //re-create array
                   capacity = rightSide.capacity;
              }
              for (int i
                    a[i]
              noOfElem =
     }
     else
              cerr << "A

     return *this;
}
```

The **parameter object** needs to have the **same capacity** of the **calling object**.

If not, clear the memory that holds the array parameter and re-create a new array with the same capacity of the parameter object.

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
             if (capacity != rightSide.capacity)
             {
                 delete [ ] a;  //release space
                 a = new int[righ                    array
                 capacity = right
             }
             for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                  a[i] = rightSide.a[i];
             noOfElem= rightSide.noOfElem;
     }
     else
             cerr << "Attempted assignment to itself.";

     return *this;
}
```

Start copying elements…

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
              if (capacity != rightSide.capacity)
              {
                   delete [ ] a;  //release space
                   a = new int[rightSide.capacity]; //re-create array
                   capacity = rightSide.capacity;
              }
              for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                    a[i] = rightSide.a[i];
              noOfElem = rightSide.noOfElem;
     }
     else
              cerr << "At

     return *this;
}
```

Update the **number of elements** of the **calling object**.

Overloading the Assignment Operator

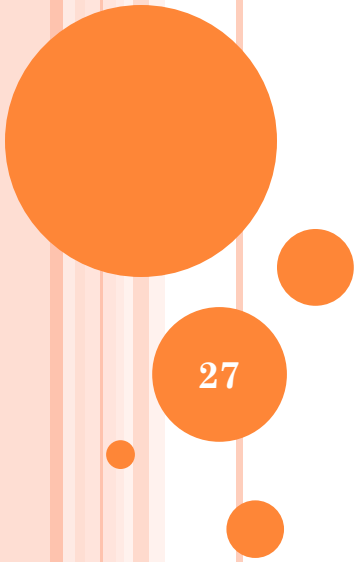
```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
              if (capacity != rightSide.capacity)
              {
                   delete [ ] a;  //release space
                   a = new int[rightSide.capacity]; //re-create array
                   capacity = rightSide.capacity;
              }
              for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                    a[i] = rightSide.a[i];
              noOfElem = rightSide.noOfElem;
     }
     else
              cerr << "Attempted assignment to itself.";

     return *this;
}
```

CAUTION!

- The **overloaded assignment operator** works **only** if the object is declared as a **separate statement**:

```
DArray a1;
// insert elements in a1
DArray a2;
a2 = a1;
```

- This is because the object **a2** needs to be constructed first.



Copy Constructor or Overloaded Assignment Operator?

Copy Constructor or Assignment?

```
DArray a1(20);
// add elements


DArray a2(a1);


DArray a3;
a3 = a1;


DArray a4 = a1;
```

Which function is invoked?

Copy Constructor or Assignment?

```
DArray a1(20);
// add elements

DArray a2(a1);

DArray a3;
a3 = a1;

DArray a4 = a1;
```

The copy constructor

Copy Constructor or Assignment?

```
DArray a1(20);
// add elements

DArray a2(a1);

DArray a3;
a3 = a1;

DArray a4 = a1;
```

The copy constructor

Which function is invoked?

Copy Constructor or Assignment?

```
DArray a1(20);
// add elements

DArray a2(a1);

DArray a3;
a3 = a1;

DArray a4 = a1;
```

The copy constructor

The overloaded assignment operator

Copy Constructor or Assignment?

```
DArray a1(20);
// add elements

DArray a2(a1);

DArray a3;
a3 = a1;

DArray a4 = a1;
```

The copy constructor

The overloaded assignment operator

Which function is invoked?

Copy Constructor or Assignment?

```
DArray a1(20);
// add elements

DArray a2(a1);

DArray a3;
a3 = a1;

DArray a4 = a1;
```

The copy constructor

The overloaded assignment operator

The copy constructor. Why?

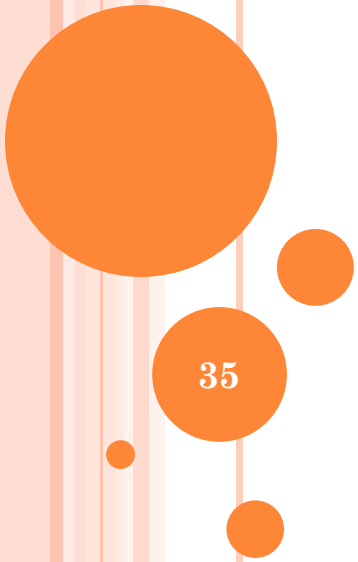
Copy Constructor or Assignment?

```
DArray a1(20);
// add elements

DArray a2(a1);

DArray a3;
a3 = a1;

DArray a4 = a1;
```

The copy constructor

The overloaded assignment operator

The copy constructor. Why?
Object **a4** needs to be constructed; this is the job of the copy constructor.



Destructors

DESTRUCTORS

- A **destructor** is called automatically when an object of the class passes out of scope.

- You need to always make sure that your destructor deletes any **dynamically-allocated variables** and any **static variables**.

DESTRUCTORS (CONT.)

- If you create a **dynamic variable** ***v*** and a **dynamic array** ***a***, you need to include in the destructor:

```
delete v;
delete [ ] a;
```

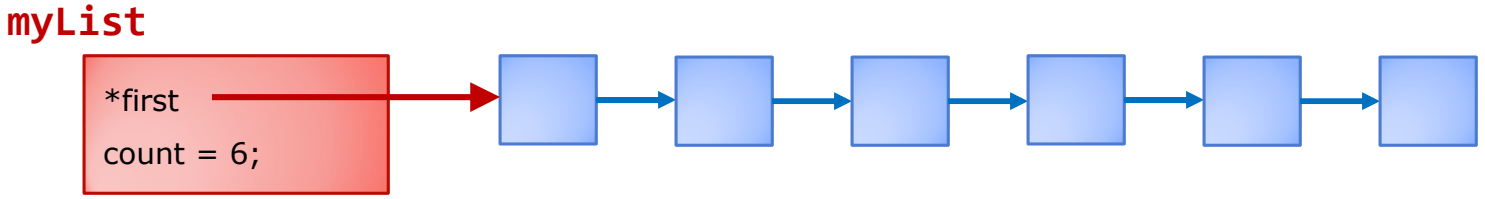
Note: If you forget the **subscript operator [ ]** when deleting the array, you will be deleting **only** the *first* element in the array.

DESTRUCTOR VS. destroyList FUNCTION

- What is the difference?
  - **destroyList**
    - Created to systematically **delete dynamic items to which the object is pointing.**
    - **Re-set all member variables** of the object to **default values**.
    - Keeps the object (you are simply **emptying** the object).
  - **Destructor**
    - Called by compiler when the object **goes out of scope**.
    - Will **delete** the object but **not** the dynamic items the object is pointing to.

DESTRUCTOR VS. destroyList FUNCTION (CONT.)

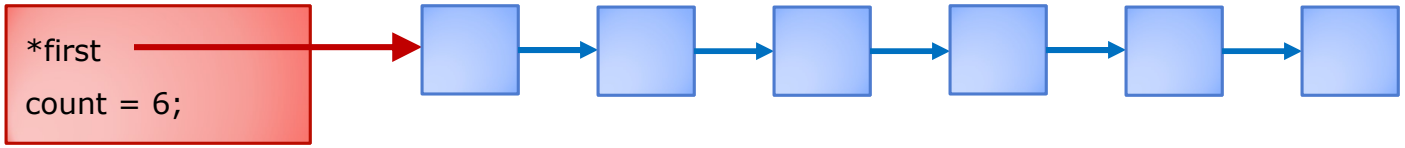
- What is the difference?
  - Given the object **myList**



  - The **destroyList** will need to:
    - destroy (delete from memory) each node and
    - reset the member variables to a default value.

DESTRUCTOR VS. destroyList FUNCTION (CONT.)

myList *(before calling the destroyList function)*



(after calling destroyList…)

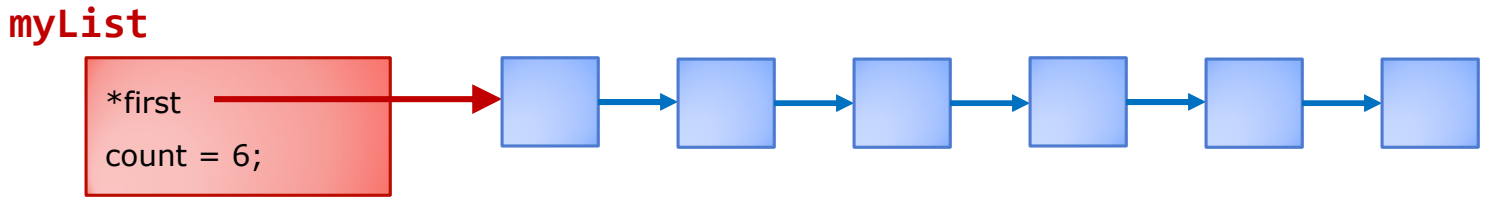
```
void AnyList::destroyList( )
{
      Node *temp = first;
      while (temp != nullptr)
      {
            first = first->getNext();
            delete temp;
            temp = first;
      }
      count = 0;
}
```

The object **myList** will still exist, but it will have no nodes.

myList

*first = NULL
count = 0;

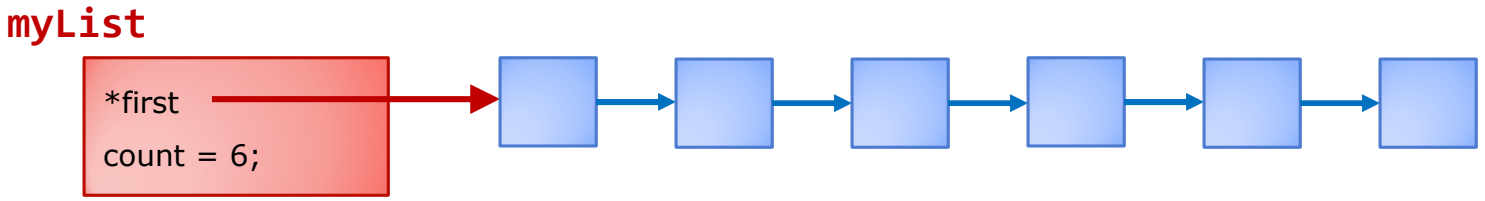
DESTRUCTOR VS. destroyList FUNCTION (CONT.)



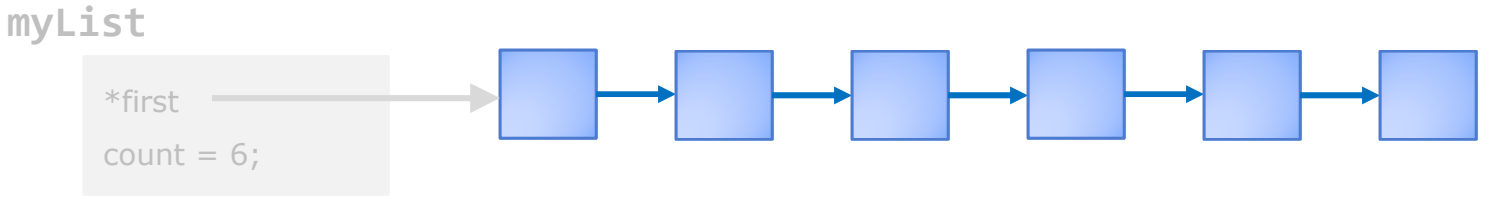
- What if you called the **destructor** without using the **destroyList** function?

```
void AnyList::~AnyList()
{ }
```

DESTRUCTOR VS. destroyList FUNCTION (CONT.)

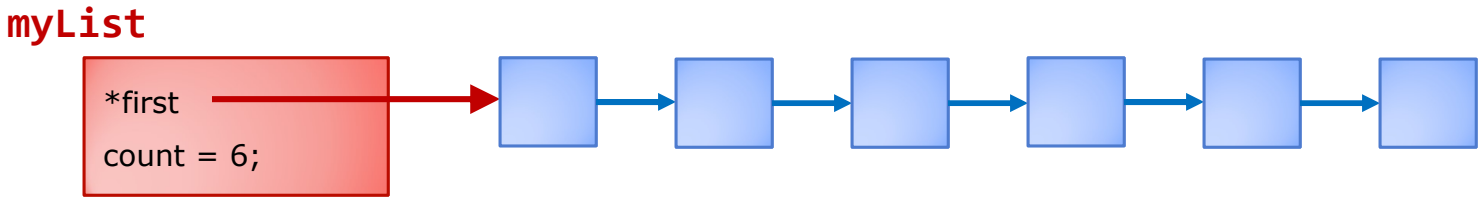


- What if you called the **destructor** without using the **destroyList** function?

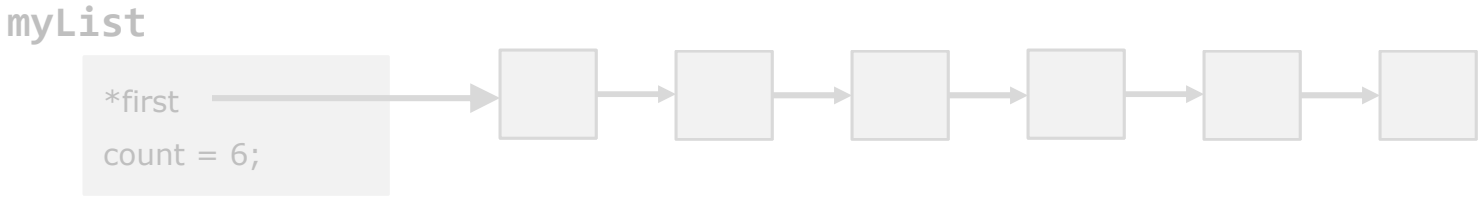


The object **myList** will be deleted, but **not** the **dynamic nodes**.

DESTRUCTOR VS. destroyList FUNCTION (CONT.)



- What if you called the **destructor** without using the **destroyList** function?



This is why we need to call the **destroyList** function directly from the **destructor**, to destroy the nodes as well.

```
void AnyList::~AnyList()
{
    destroyList();
}
```

Example 2

- Project: The Big Three

The Big Three (end)