CSC212

Data Structure

Lecture 6

Dynamic Classes and the Law of the Big Three

Instructor: George Wolberg
Department of Computer Science
City College of New York

Why Dynamic Classes

- Limitation of our bag class
  - bag::CAPACITY constant determines the capacity of every bag
  - wasteful and hard to reuse
- Solution:
  - provide control over size in running time, by
  - pointers and dynamic memory
  - => dynamic arrays
  - => dynamic classes

Dynamic Classes New Features (Ch 4.3–4)

- Pointers Member Variables
- Dynamic Memory Allocation (where and how)
- Value Semantics (what's new?)
  - assignment operator overloading
  - your own copy constructor
- Introducing **Destructor**
- Conclusion: **the Law of the Big Three**

Pointer Member Variable

- The Static bag

```
// From  bag1.h in Section 3.1
class bag
{
public:
    static const size_t CAPACITY = 20;
    ...
private:
    value_type  data[CAPACITY];
    size_type used;
};
```

- The Dynamic bag

```
// From  bag2.h in Section 4.3
class bag
{
public:
    ...
private:
    value_type  *data;
    size_type used;
    size_type  capacity;
};
```

Invariant of the Dynamic bag

- the number of items is in the member variable used
- The actual items are stored in a partially filled array. The array is a dynamic array, pointed to by the pointer variable data
- The total size of the dynamic array is the member variable capacity

- Invariant is about rules of implementation...

Allocate Dynamic Memory: Where?

- In Old Member Functions
  - constructor – how big is the initial capacity?
  - insert – if bag is full, how many more?
  - +/+= operators – how to combine two bags?
- New Member Functions
  - reserve – explicitly adjust the capacity
- Example
  - constructor with default size

Allocate Dynamic Memory: How?

```
// From  bag2.h in Section 4.3
class bag
{
public:
     static const size_t DEFAULT_CAPACITY = 20;
     bag(size_type init_cap = DEFAULT_CAPACITY);

    ...
private:
    value_type  *data;
    size_type used;
    size_type  capacity;
};
```

- In constructor:
  - why initialize?
  - how?
    - default
    - specific size

```
// From implementation file bag2.cpp
bag::bag(size_type init_cap)
{
   data = new value_type[init_cap];
   capacity = init_cap;
   used = 0;
}
```

Value Semantics

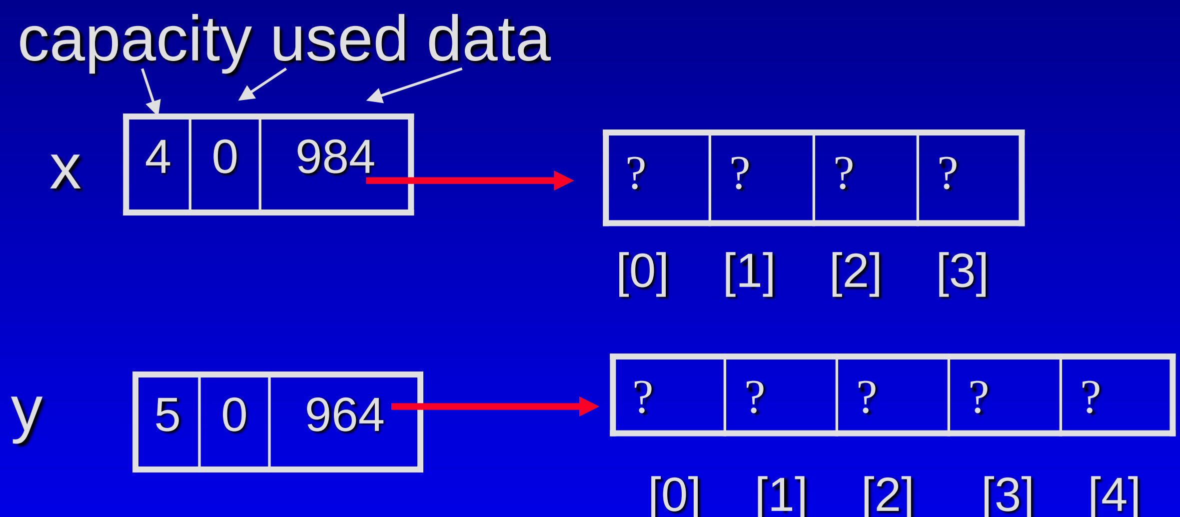
- Assignment operator
  - y = x;
- Copy constructor
  - bag y(x); // bag y = x;

Automatic assignment operator and copy constructor

- copy all the member variables (data, used, capacity) from object x to object y
- but our days of easy contentment are done!

Failure in auto assignment operator

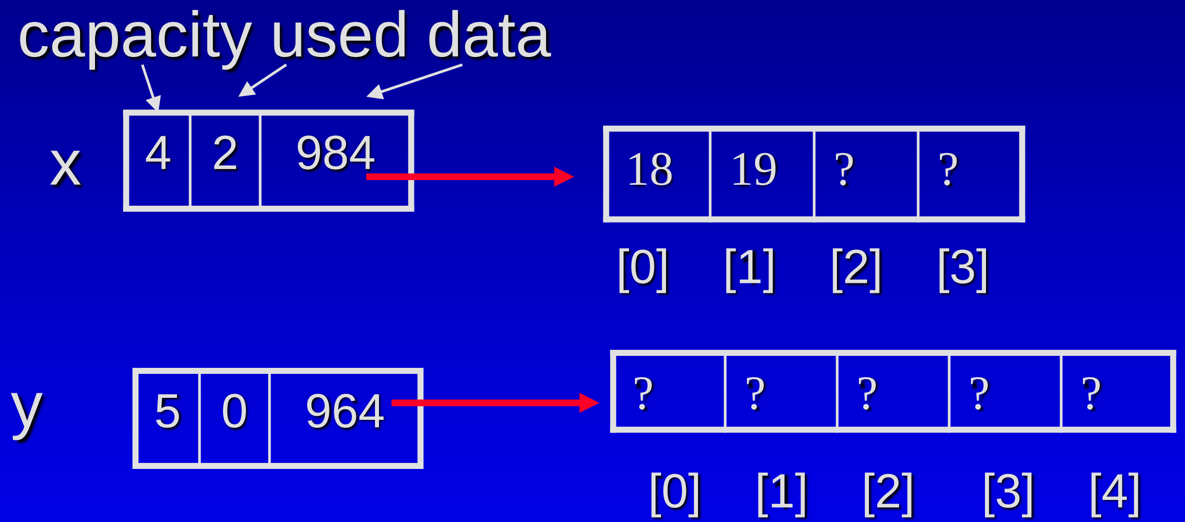
```
bag x(4), y(5);
x.insert(18);
x.insert(19);
y=x;
x.insert(20);
```



Question: What will happen after executing lines 2 – 5?

Failure in auto assignment operator

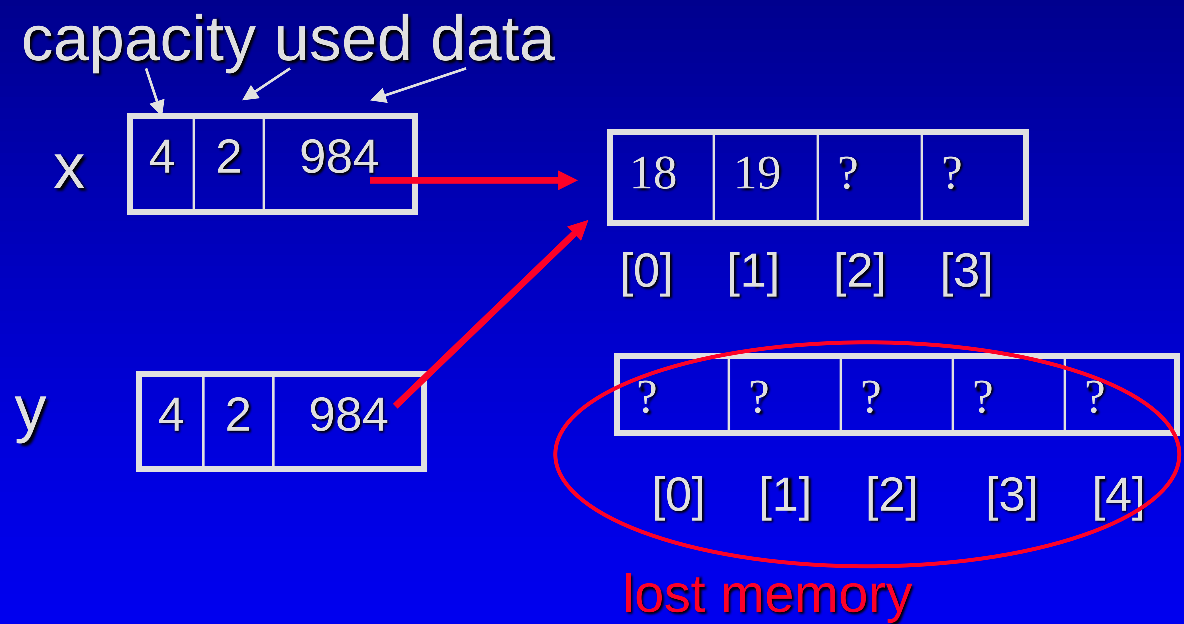
```
bag x(4), y(5);
x.insert(18);
x.insert(19);
y=x;
x.insert(20);
```



Question: What will happen after executing lines 2 – 5?

Failure in auto assignment operator

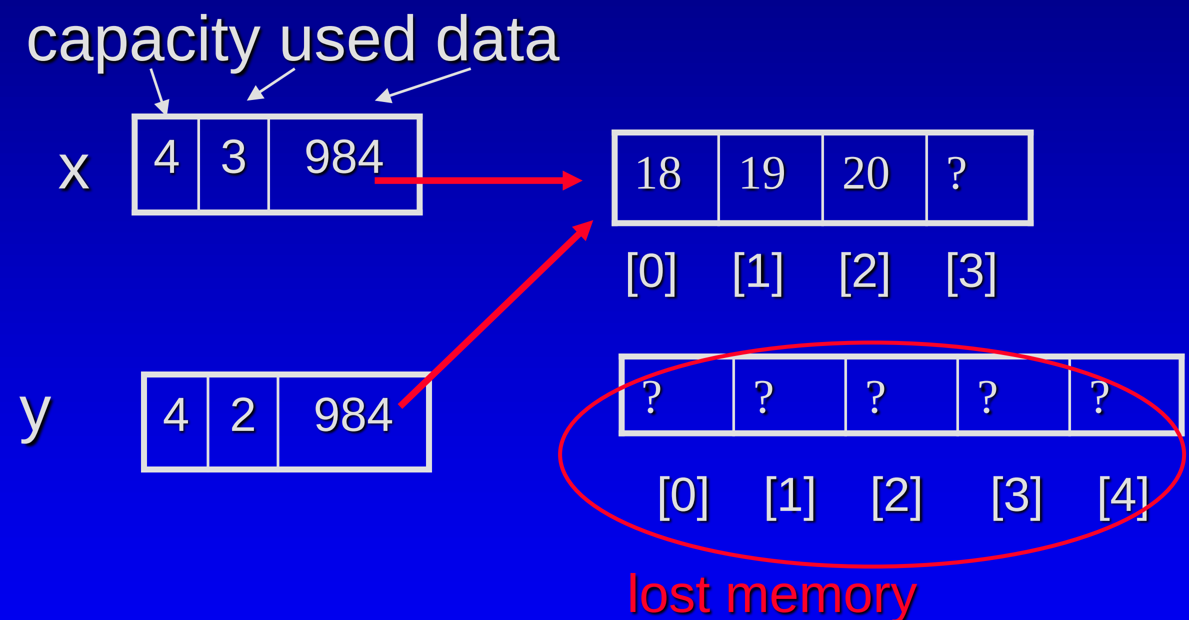
```
bag x(4), y(5);
x.insert(18);
x.insert(19);
y=x;
x.insert(20);
```



Question: What will happen after executing lines 2 – 5?

Failure in auto assignment operator

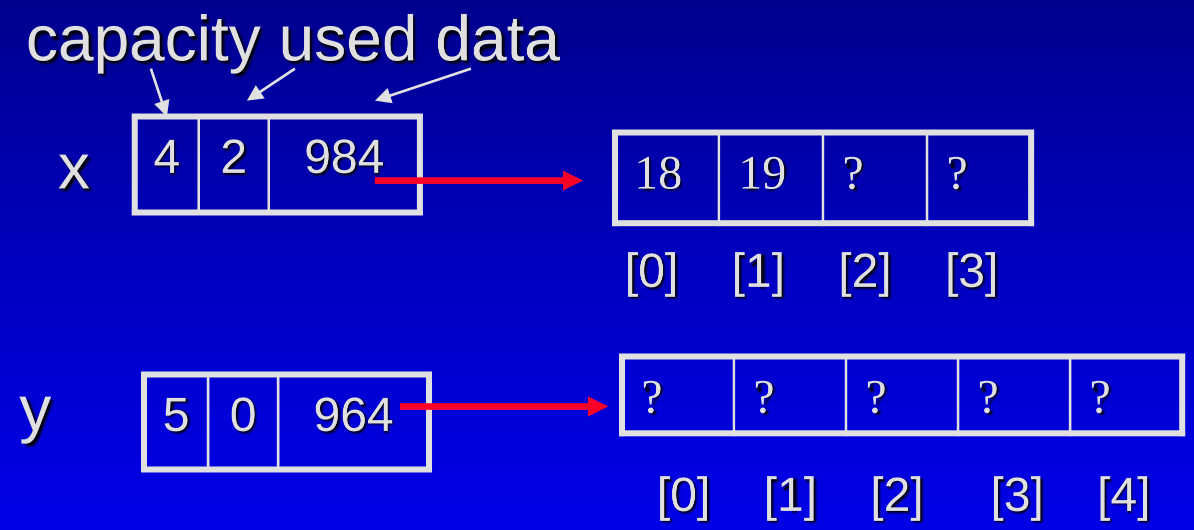
```
bag x(4), y(5);
x.insert(18);
x.insert(19);
y=x;
x.insert(20);
```



Consequence: Change to x’ array will also change y’s array

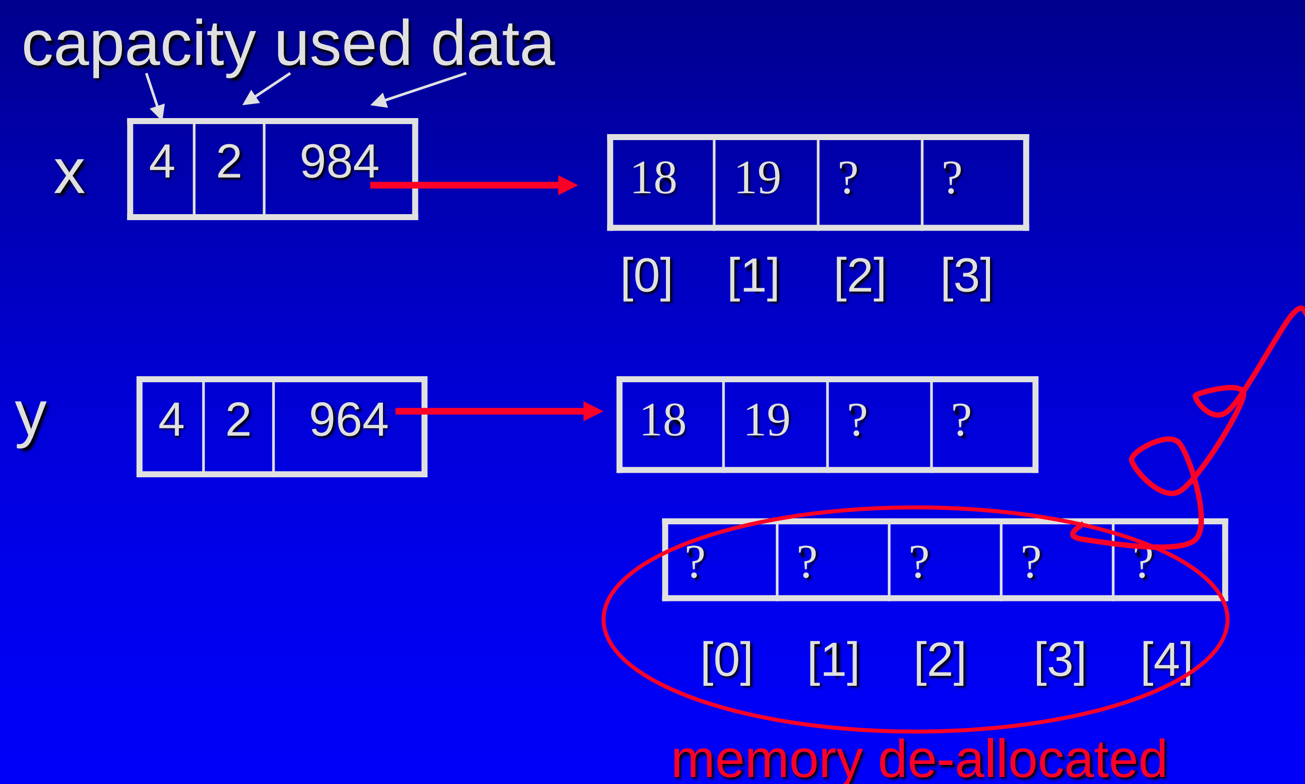
If we want y to have its own dynamic array

```
bag x(4), y(5);
x.insert(18);
x.insert(19);
y=x;
x.insert(20);
```



Dynamic memory allocation is needed

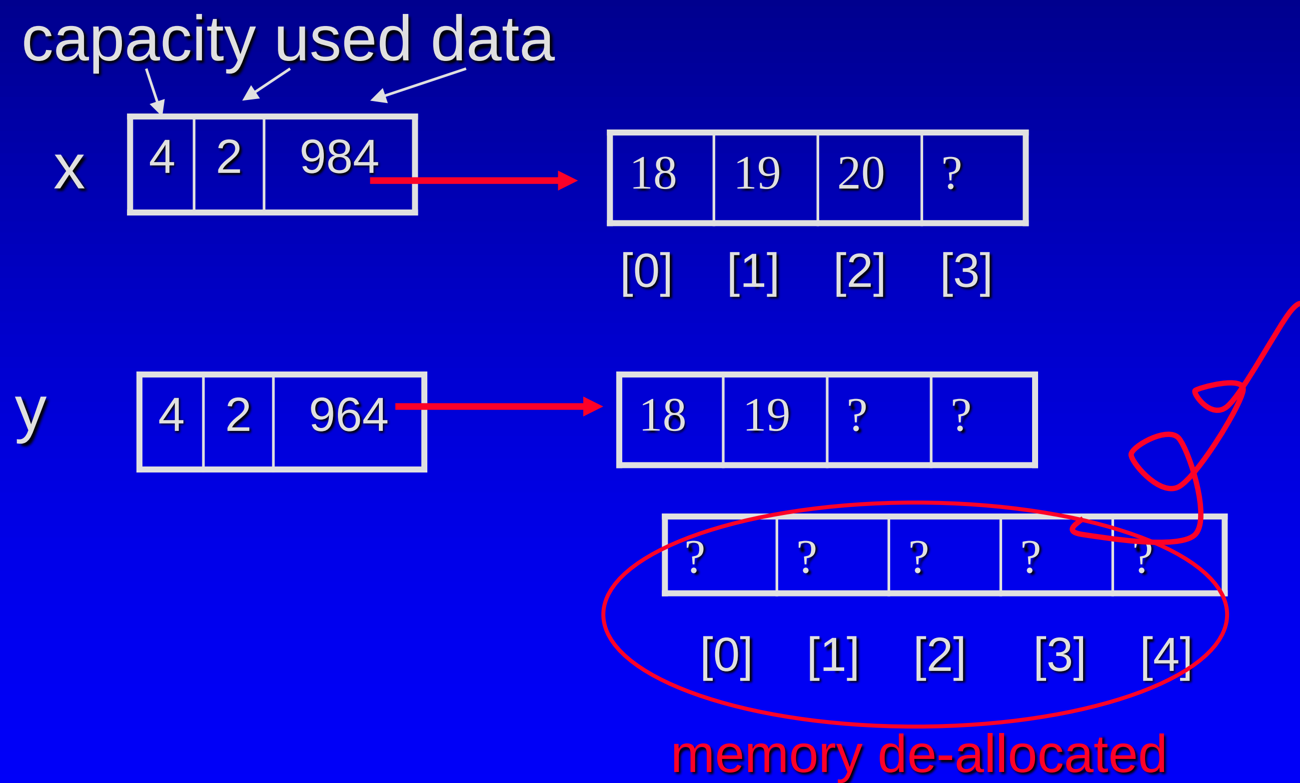
```
bag x(4), y(5);
x.insert(18);
x.insert(19);
y=x;
x.insert(20);
```



Answer: overloading the assignment operator =

Dynamic memory allocation is needed

```
bag x(4), y(5);
x.insert(18);
x.insert(19);
y=x;
x.insert(20);
```



Answer: overloading the assignment operator =

ignment operator

```
// From  bag2.h in Section 4.3
class bag
{
public:
     static const size_t DEFAULT_CAPACITY = 20;
     bag(size_type init_cap = DEFAULT_CAPACITY);

    ...
private:
    value_type  *data;
    size_type used;
    size_type  capacity;
};
```

```
// From implementation file bag2.cpp
bag::bag(size_type init_cap)
{
   data = new value_type[init_cap];
   capacity = init_cap;
   used = 0;
}
```

bag x, y; // OR bag x(4), y(5); // OR....

y=x; // y.operator=(x);

```
void bag::operator=(const bag& source)
// Postcondition: The bag that activated this function
    has the same items and capacity as source
```

A 5-minute Quiz: write your own implementation - turn in

Implementation of operator=

- y = x;
- y ⇔ *this
- x ⇔ source

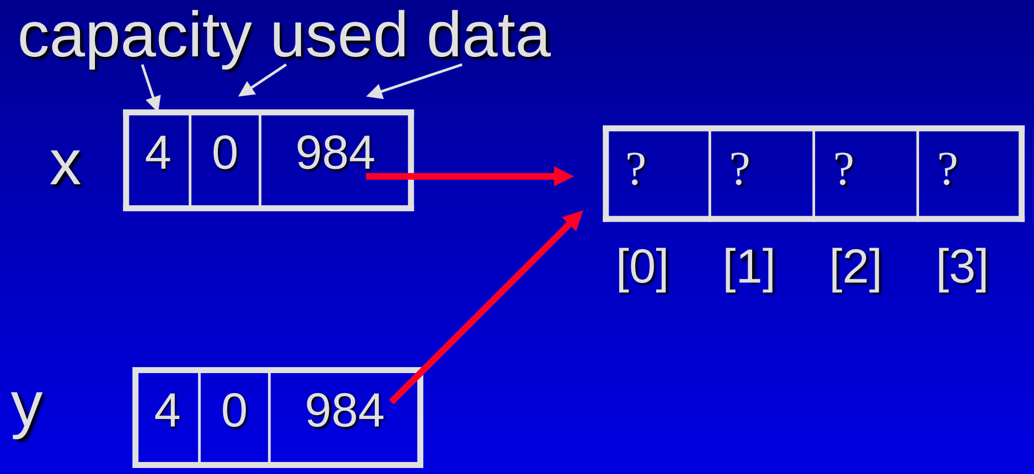
```
void bag::operator =(const bag& source)
  // Library facilities used: algorithm
  {
    value_type *new_data;

    // Check for possible self-assignment:
    if (this == &source)
          return;

    // If needed, allocate an array with a different size:
    if (capacity != source.capacity)
    {
        new_data = new value_type[source.capacity];
        delete [ ] data; // make sure all valid before delete!!!
        data = new_data;
        capacity = source.capacity;
    }

    // Copy the data from the source array:
    used = source.used;
    copy(source.data, source.data + used, data);
  }
```

The 2nd part of the value semantics

copy constructor

Auto Copy Constructor: shallow copy

```
bag x(4)
bag y(x);
x.insert(18);
x.insert(19);
```

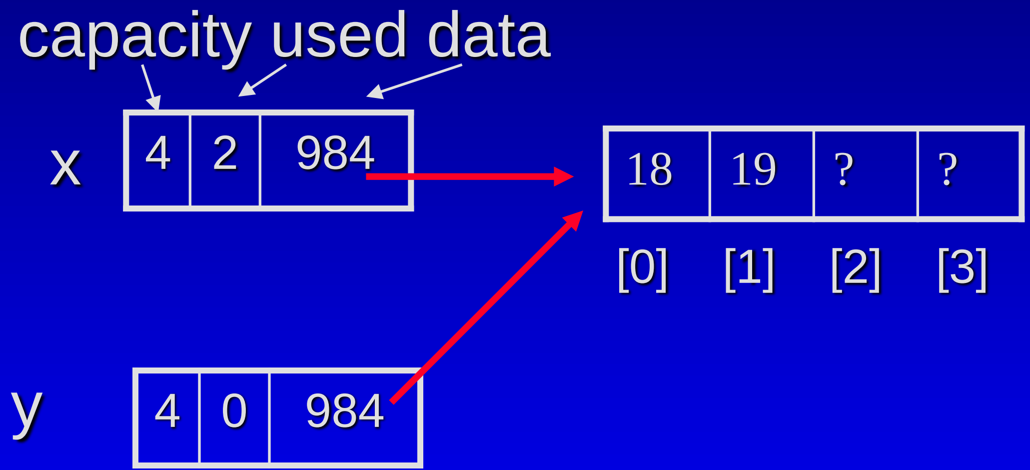


The only difference with auto assignment is:

y does not have its own data

Failure in auto copy constructor

```
bag x(4);
bag y(x);
x.insert(18);
x.insert(19);
```



change to x also changes y

Deep copy: providing your own copy constructor

```
bag::bag(const bag& source)
// Postcondition: The bag that has been constructed
    has the same items and capacity as source
```

- Questions on Implementation (homework!)
  - do you need to check self-copy
    - bag y(x); // never have bag y(y);
  - do you need to delete old bag?
- Questions on Usage
  - 4 different ways that copy constructor is used

Four common situations

- Declaration

  bag y(x);

- Declaration with Alternate Syntax

  bag y = x ;

- Returning an object from a function

  bag union(const bag& s1, const bag& s2);

- Value parameter is an object

  void temp_bag_copy(bag clone);

What’s missing?

allocate dynamic memory via new,

take care of the value semantics,

....?

De-allocation of dynamic memory

- Return an object's dynamic memory to the heap when the object is no longer in use
- Where and How? – Two ways
  - Take care of it yourself
    - delete dynamic data of an object after you're done with it
  - let the program do it automatically
    - destructor

Destructor

```
bag::~bag()
{
    delete [ ] data;
}
```

- The primary purpose is to return an object's dynamic memory to the heap, and to do other "cleanup"
- Three unique features of the destructor
  - The name of the destructor is always ~ followed by the class name;
  - No parameters, no return values;
  - Activated automatically whenever an object becomes inaccessible

Question: when this happens?

Destructor

```
bag::~bag()
{
    delete [ ] data;
}
```

- Some common situations causing automatic destructor activation
  - Upon function return, objects as local variables destroyed;
  - Upon function return, objects as value parameters destroyed;
  - when an object is explicitly deleted

Question: shall we put destructor in how-to-use-a-bag documentation?

The Law of the Big Three

- Using dynamic memory requires the following three things all together
  - a destructor
  - a copy constructor (and of course an ordinary one)
  - an overloaded assignment operator
- In other words, the three functions come in a set – either you need to write all three yourself, or you can rely on the compiler-supplied automatic versions of all the three.

What will happen if not?

If we only have a constructor and a destructor, but do not provide a copy constructor and an overloaded assignment operator

Importance of the Law of Big-3

```
bag *x, *y;
x = new bag(4);
y = new bag(5);
x->insert(18);
x->insert(19);
*y = *x;
delete x;
y->insert(20);
```

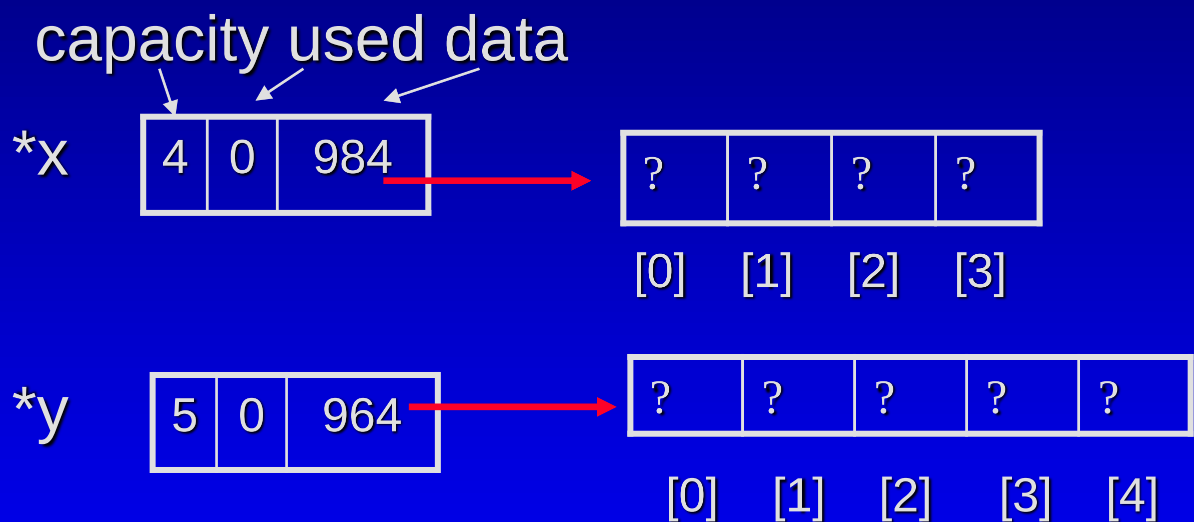
```
// constructor
bag::bag(size_type init_cap)
{
  data = new value_type[init_cap];
  capacity = init_cap;
  used = 0;
}
```

```
// destructor
bag::~bag()
{
    delete [ ] data;
}
```

Question: What will happen after executing lines 1 – 8?

Importance of the Law of Big-3

```
bag *x, *y;
x = new bag(4);
y = new bag(5);
x->insert(18);
x->insert(19);
*y = *x;
delete x;
y->insert(20);
```

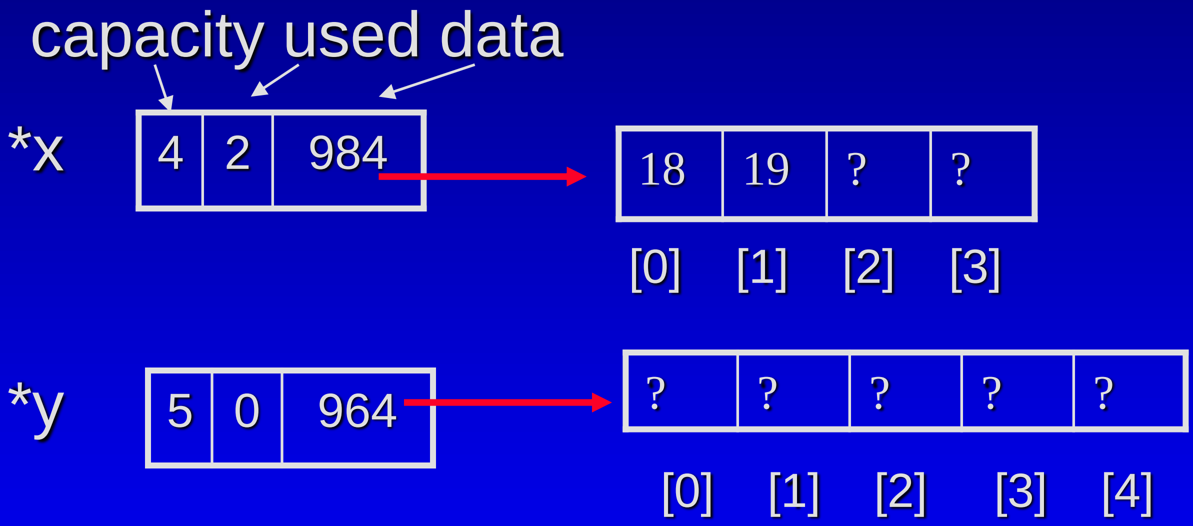


allocate memory for objects (*x, *y) and their dynamic arrays

```
// From implementation file bag2.cpp
bag::bag(size_type init_cap)
{
  data = new value_type[init_cap];
  capacity = init_cap;
  used = 0;
}
```

Importance of the Law of Big-3

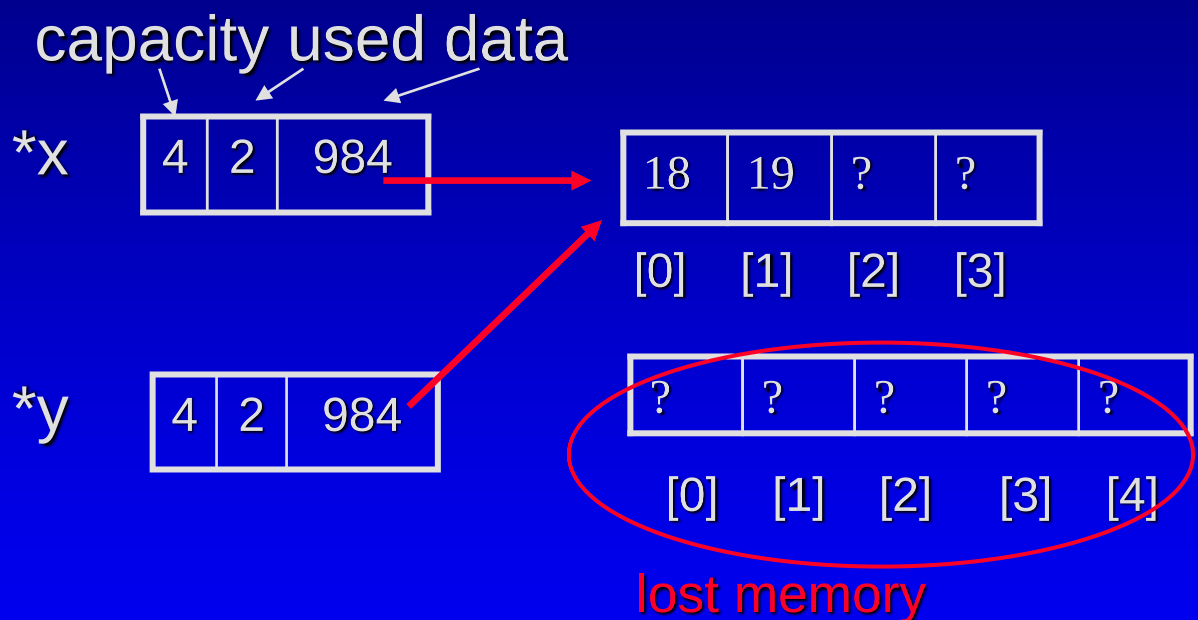
```
bag *x, *y;
x = new bag(4);
y = new bag(5);
x->insert(18);
x->insert(19);
*y = *x;
delete x;
y->insert(20);
```



Insert two items in the dynamic array of object *x

Importance of the Law of Big-3

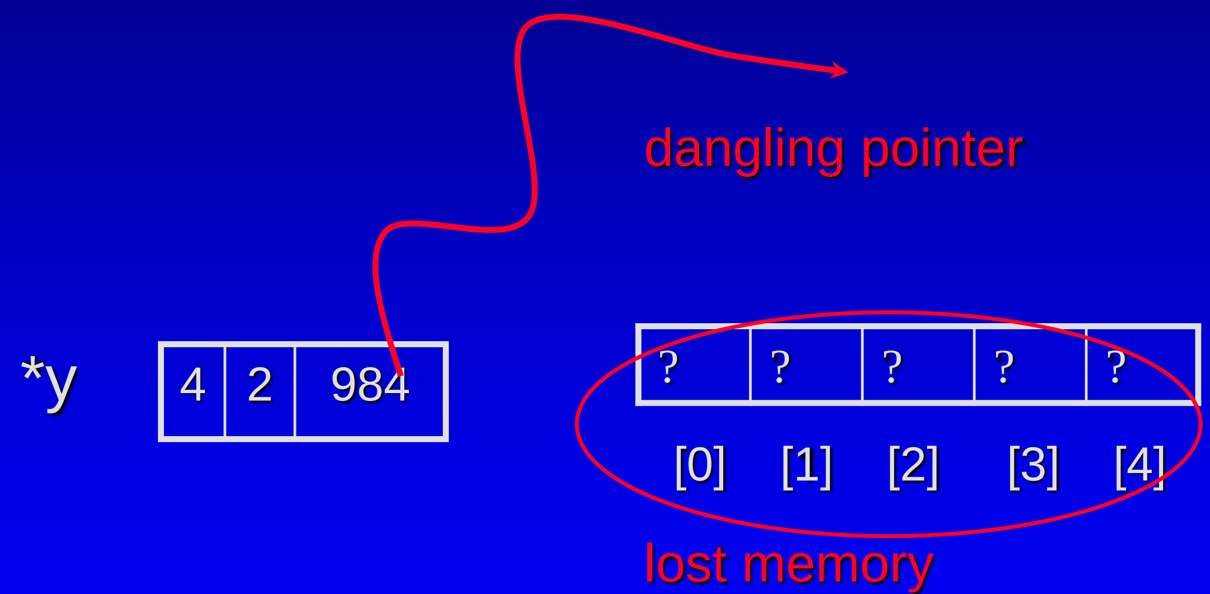
```
bag *x, *y;
x = new bag(4);
y = new bag(5);
x->insert(18);
x->insert(19);
*y = *x;
delete x;
y->insert(20);
```



automatic assignment only copies three variables (capacity, used and data) from *x to *y

Importance of the Law of Big-3

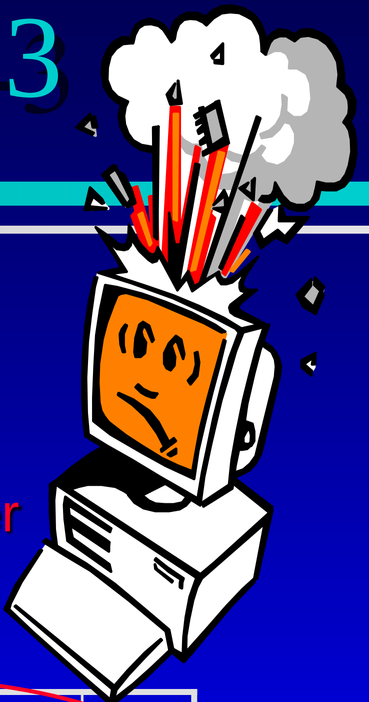
```
bag *x, *y;
x = new bag(4);
y = new bag(5);
x->insert(18);
x->insert(19);
*y = *x;
delete x;
y->insert(20);
```



Deleting x will also delete the dynamic array of *x by calling the destructor

```
bag::~bag()
{
    delete [ ] data;
}
```

Importance of the Law of Big-3



```
bag *x, *y;
x = new bag(4);
y = new bag(5);
x->insert(18);
x->insert(19);
*y = *x;
delete x;
y->insert(20);
```

dangling pointer

*y

4	2	984

[0]	[1]	[2]	[3]	[4]
?	?	?	?	?

lost memory

Your program crashes: *y needs its own copy of data !!!

Reading and Programming Assignments

- Putting pieces together
  - bag2.h, bag2.cpp both in textbook and online
- **Self-test exercises**
  - **16 - 23**
- After-class reading (string)
  - Section 4.5, Self-Test 26- 32 (within exam scope)