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
  ```cpp
  // 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
  ```cpp
  // 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?

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

// 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;
}
Value Semantics

- Assignment operator
  - \[ y = x; \]
- Copy constructor
  - \[ \text{bag } y(x); // \text{ bag } y = x; \]

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

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

```c
bag x(4), y(5);
x.insert(18);
x.insert(19);
y=x;
x.insert(20);
```
Failure in auto assignment operator

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

<table>
<thead>
<tr>
<th>capacity used data</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>18</th>
<th>19</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
</tr>
</tbody>
</table>

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);

capacity used data

x
4 2 984

y
4 2 984

[0] [1] [2] [3]
18 19 ? ?
[0] [1] [2] [3] [4]

Question: What will happen after executing lines 2 – 5?
Failure in auto assignment operator

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

Consequence: Change to x’s array will also change y’s array.
If we want `y` to have its own dynamic array

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>984</td>
<td>964</td>
</tr>
</tbody>
</table>

```
```
Dynamic memory allocation is needed

Answer: overloading the assignment operator =

```c
bag x(4), y(5);
x.insert(18);
x.insert(19);
y = x;
x.insert(20);
```
**Dynamic memory allocation is needed**

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

<table>
<thead>
<tr>
<th>capacity used data</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>18 19 20 ?</td>
</tr>
<tr>
<td>y</td>
<td>18 19 ? ?</td>
</tr>
</tbody>
</table>

Answer: overloading the assignment operator =
Solution:

overloading assignment operator

Your own assignment operator

C++ Requires the overloaded assignment operator to be a member function

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

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

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

- y = x;
- y ↔ *this
- x ↔ source

```cpp
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 2\textsuperscript{nd} part of the value semantics

\textit{copy constructor}
Auto Copy Constructor: shallow copy

The only difference with auto assignment is:
y does not have its own data
Failure in auto copy constructor

```c++
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

- 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?

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

- 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?

```cpp
destructor
```
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

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

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

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

Question: What will happen after executing lines 1 – 8?
# Importance of the Law of Big-3

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

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

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

### capacity used data

<table>
<thead>
<tr>
<th>*x</th>
<th>4</th>
<th>0</th>
<th>984</th>
</tr>
</thead>
<tbody>
<tr>
<td>*y</td>
<td>5</td>
<td>0</td>
<td>964</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
</tr>
</thead>
</table>
Importance of the Law of Big-3

```
import bag

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

```c
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

```c
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

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

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

lost memory

dangling pointer

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)