CSC212

Data Structure

- Section FG

Lectures 4 & 5
Container Classes

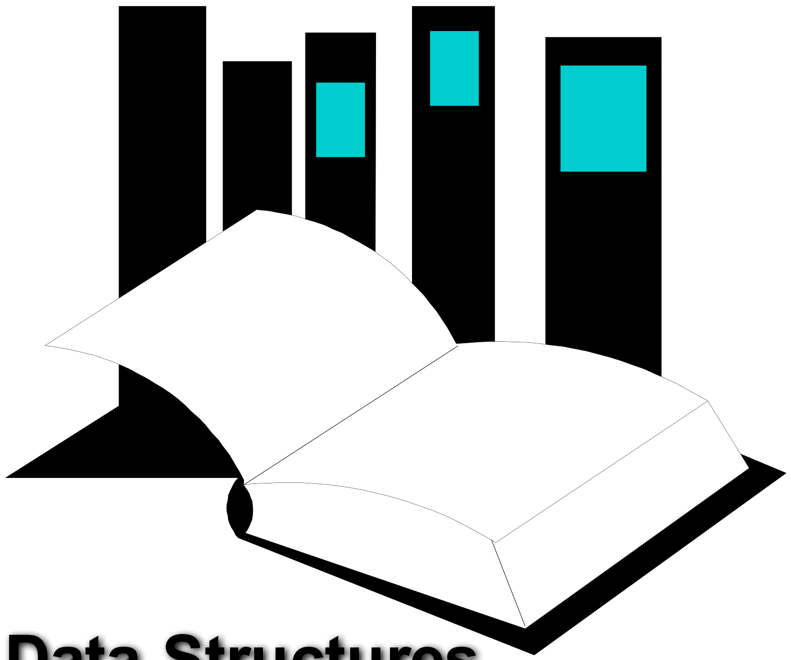
Instructor: Feng HU

Department of Computer Science

City College of New York

Container Classes

- A <u>**container class**</u> is a data type that is capable of holding a collection of items.
- In C++, container classes can be implemented as a class, along with member functions to add, remove, and examine items.



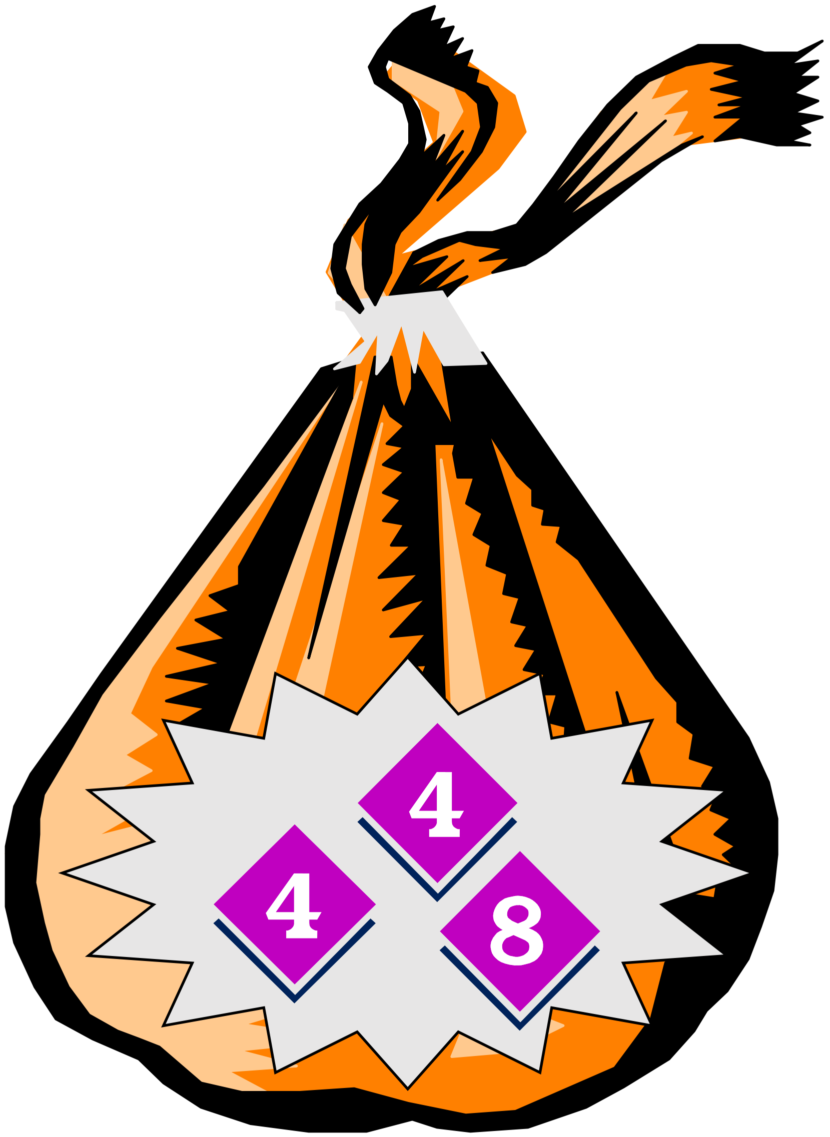
Data Structures and Other Objects Using C++

Bags

- For the first example, think about a bag.

Bags

- For the first example, think about a bag.
- Inside the bag are some numbers.



Initial State of a Bag

- When you first begin to use a bag, the bag will be empty.
- We count on this to be the **initial state** of any bag that we use.

Inserting Numbers into a Bag

- Numbers may be inserted into a bag.

Inserting Numbers into a Bag

- Numbers may be inserted into a bag.

Inserting Numbers into a Bag

- Numbers may be inserted into a bag.
- The bag can hold many numbers.

Inserting Numbers into a Bag

- Numbers may be inserted into a bag.
- The bag can hold many numbers.

Inserting Numbers into a Bag

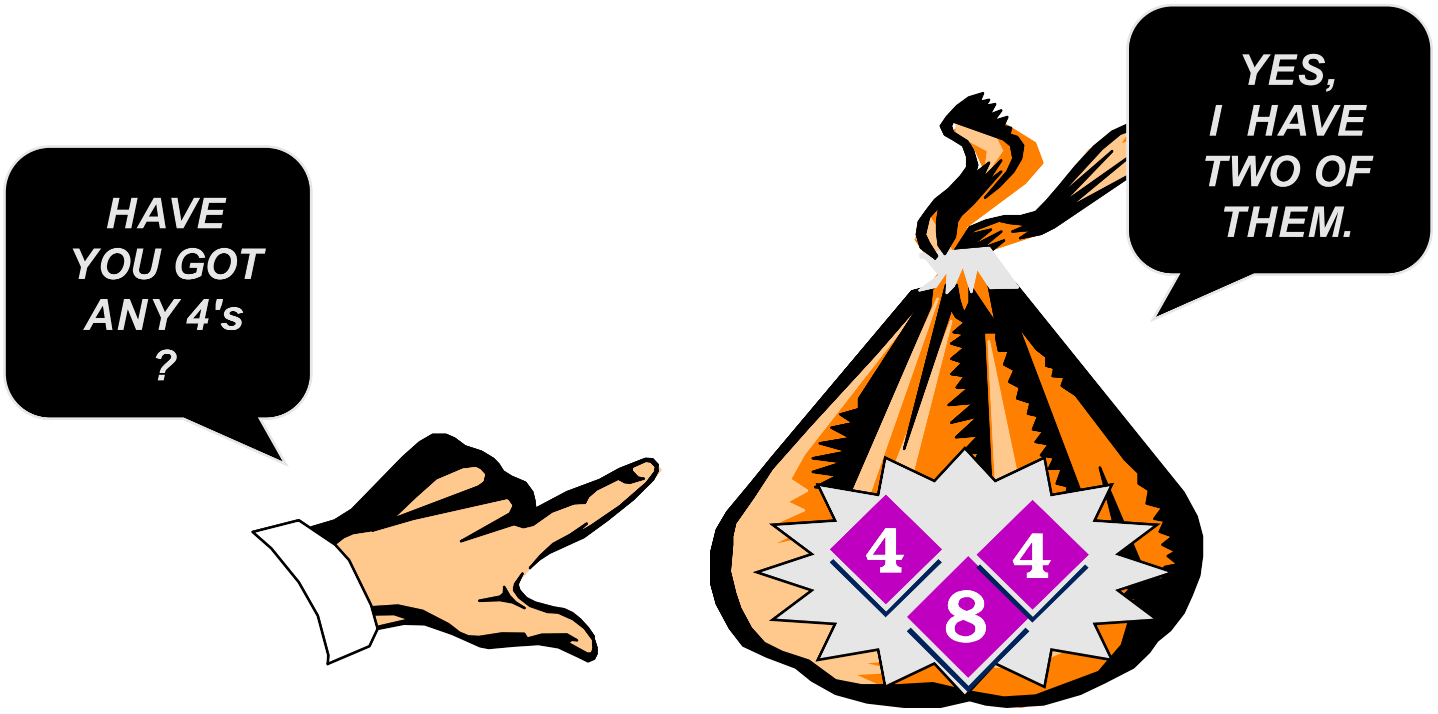
- Numbers may be inserted into a bag.
- The bag can hold many numbers.
- We can even insert the same number more than once.

Inserting Numbers into a Bag

- Numbers may be inserted into a bag.
- The bag can hold many numbers.
- We can even insert the same number more than once.

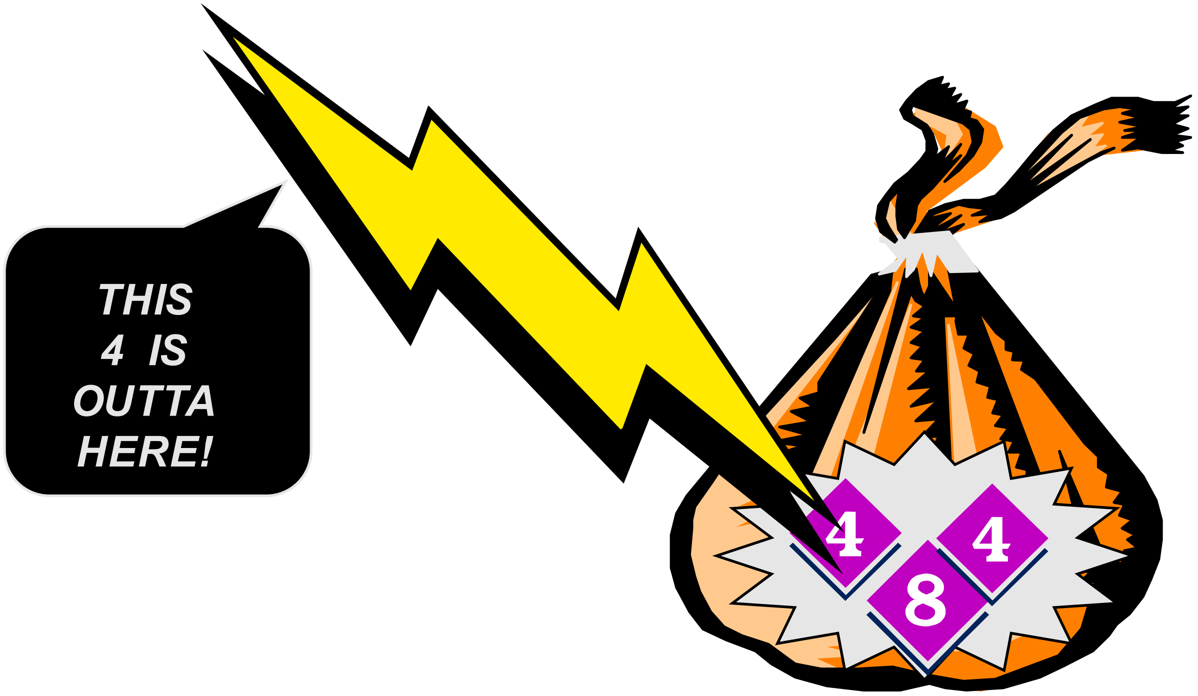
Examining a Bag

- We may ask about the contents of the bag.



Removing a Number from a Bag

- We may remove a number from a bag.



Removing a Number from a Bag

- We may remove a number from a bag.
- But we remove only one number at a time.

ONE 4 IS GONE, BUT THE OTHER 4 REMAINS.
4
8

How Many Numbers

- Another operation is to determine how many numbers are in a bag.

Summary of the Bag Operations

❶A bag can be put in its **initial state**, which is an empty bag.

❷Numbers can be **inserted** into the bag.

❸You may **count** how many occurrence of a certain number are in the bag.

❹Numbers can be **erased** from the bag.

❺You can check the **size** of the bag (i.e. how many numbers are in the bag).

The bag Class

- C++ classes (introduced in Chapter 2) can be used to implement a container class such as a bag.
- The class definition includes:

✓ **The heading of the definition**

```
class bag
```

The **bag** Class

- C++ classes (introduced in Chapter 2) can be used to implement a container class such as a bag.
- The class definition includes:

✓ **The heading of the definition**
✓ **A constructor prototype**

```
class bag
{
public:
        bag(  );
```

The bag Class

- C++ classes (introduced in Chapter 2) can be used to implement a container class such as a bag.
- The class definition includes:
  - ✓ The heading of the definition
  - ✓ A constructor prototype
  - ✓ Prototypes for public member functions

```
class bag
{
public:
        bag(  );
        void insert(...
        void erase(...
        ...and so on
```

The bag Class

- C++ classes (introduced in Chapter 2) can be used to implement a container class such as a bag.
- The class definition includes:
  - ✓ The heading of the definition
  - ✓ A constructor prototype
  - ✓ Prototypes for public member functions
  - ✓ Private member variables

```
class bag
{
public:
        bag( );
        void insert(...
        void erase(...
        ...and so on
private:
```

We'll look at private members later.

```
};
```

The **bag**’s Default Constructor

- Places a bag in the initial state (an empty bag)

```
bag::bag( )
//  Postcondition:  The bag has been initialized
//  and it is now empty.
{
 . . .

}
```

The **insert** Function

- Inserts a new number in the bag

```
void bag::insert(const int& new_entry)
//   Precondition:  The bag is not full.
//   Postcondition:  A new copy of new_entry has
//   been added to the bag.
{

    . . .
}
```

The **size** Function

- Checks how many integers are in the bag.

```
int bag::size(  ) const
//   Postcondition:  The return value is the number
//   of integers in the bag.
{

    . . .
}
```

The size Function

- Checks how many integers are in the bag.

```
size_t bag::size(  ) const
//   Postcondition:  The return value is the number
//   of integers in the bag.
{

    . . .
}
```

The count Function

- Counts how many copies of a number occur

```
size_t bag::count(const int& target) const
//   Postcondition:  The return value is the number
//   of copies of target in the bag.
{

    . . .
}
```

The **erase_one** Function

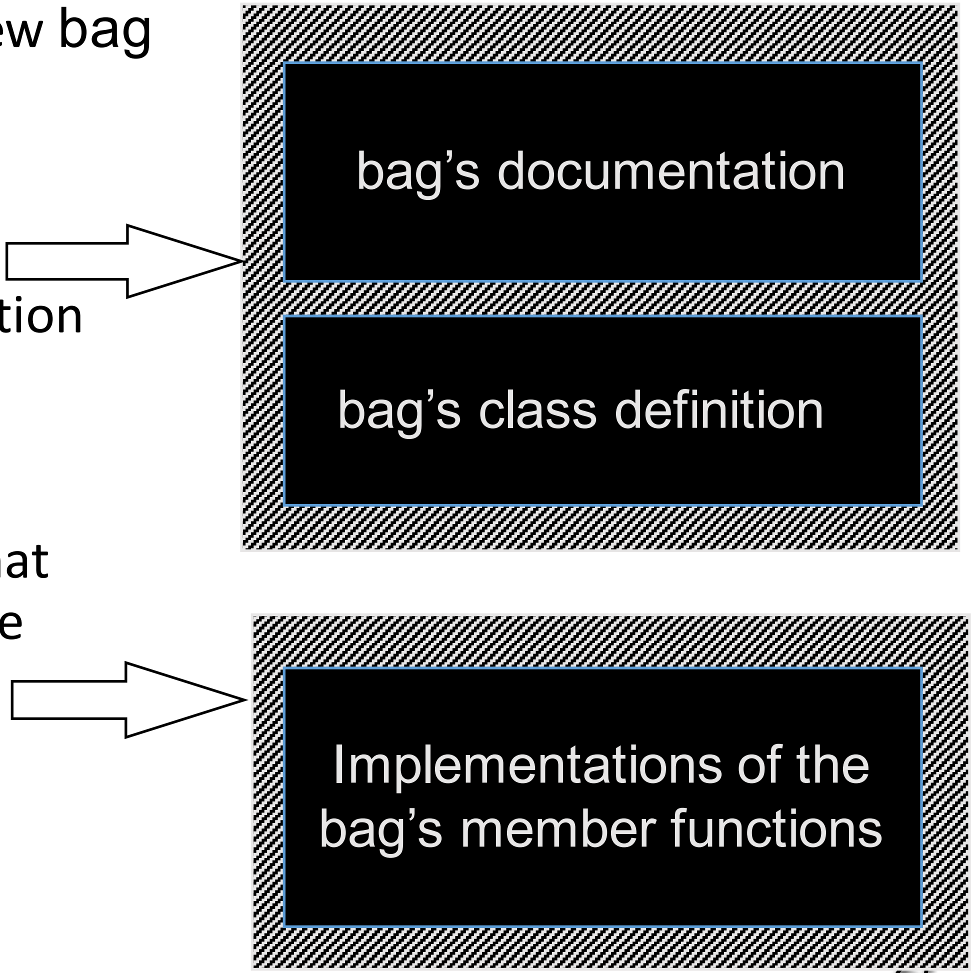
- Removes (erase) one copy of a number

```
void bag::erase_one(const int& target)
//   Postcondition:  If target was in the bag, then
//   one copy of target has been removed from the
//   bag; otherwise the bag is unchanged.
{

    . . .
}
```

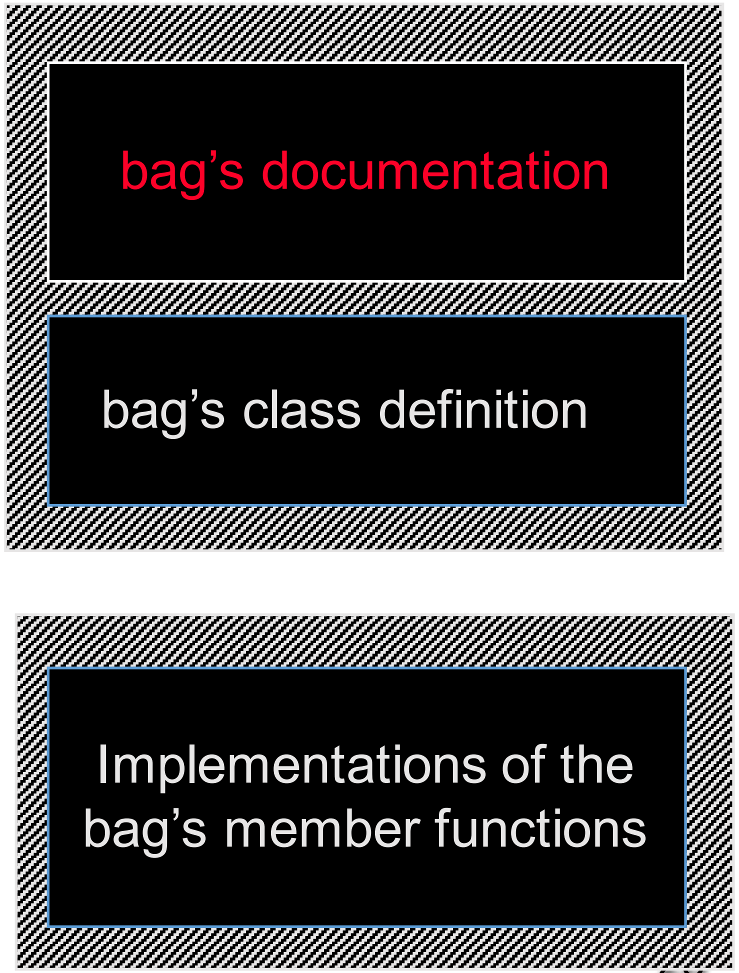
The Header File and Implementation File

- The programmer who writes the new bag class must write two files:
- **bag1.h**, a header file that contains documentation and the class definition
- **bag1.cxx**, an implementation file that contains the implementations of the bag's member functions



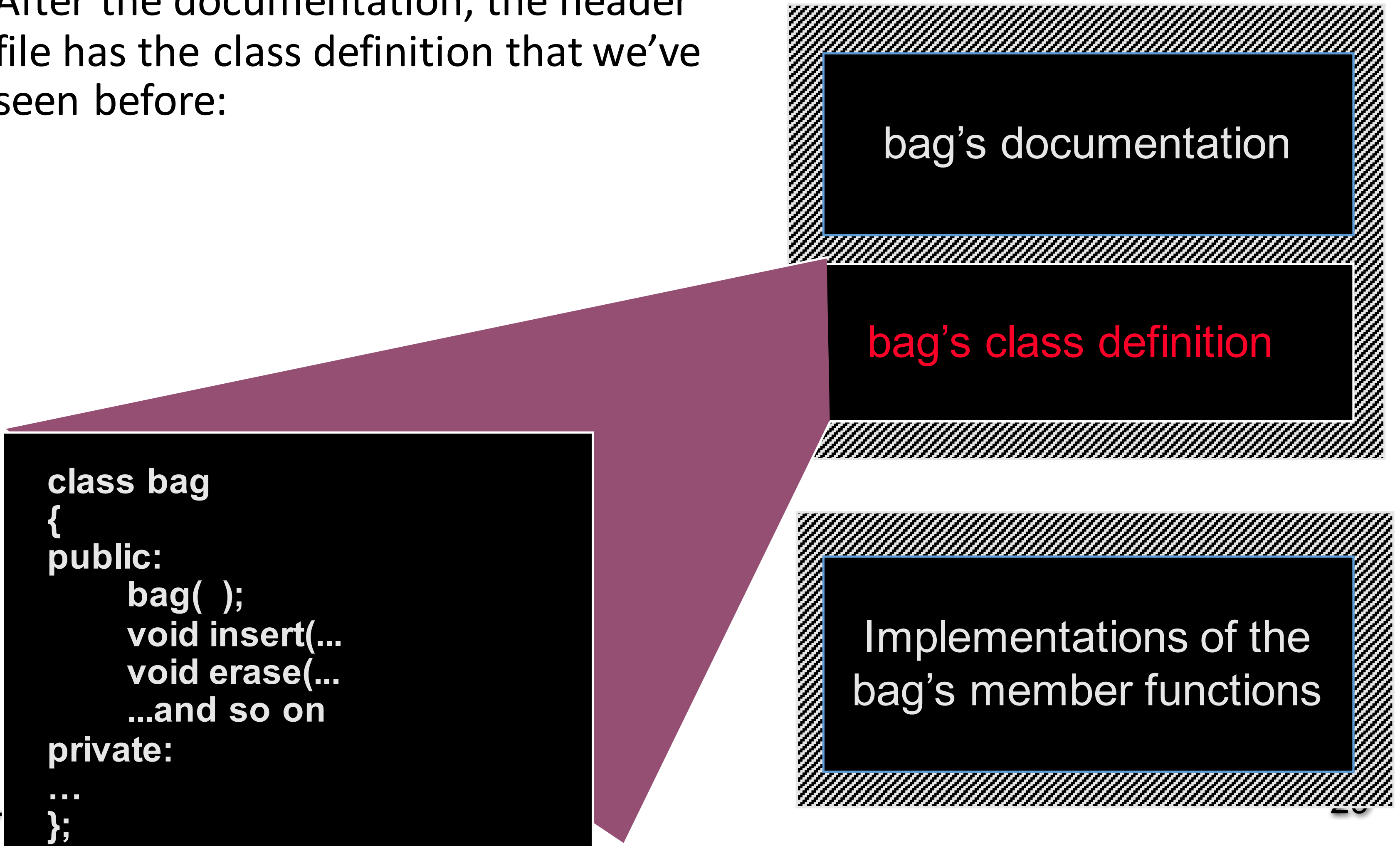
Documentation for the bag Class

- The documentation gives **prototypes and specifications** for the bag member functions.
- Specifications are written as **precondition/postcondition** contracts.
- Everything needed to **use** the bag class is included in this comment.



The **bag** ’s Class Definition

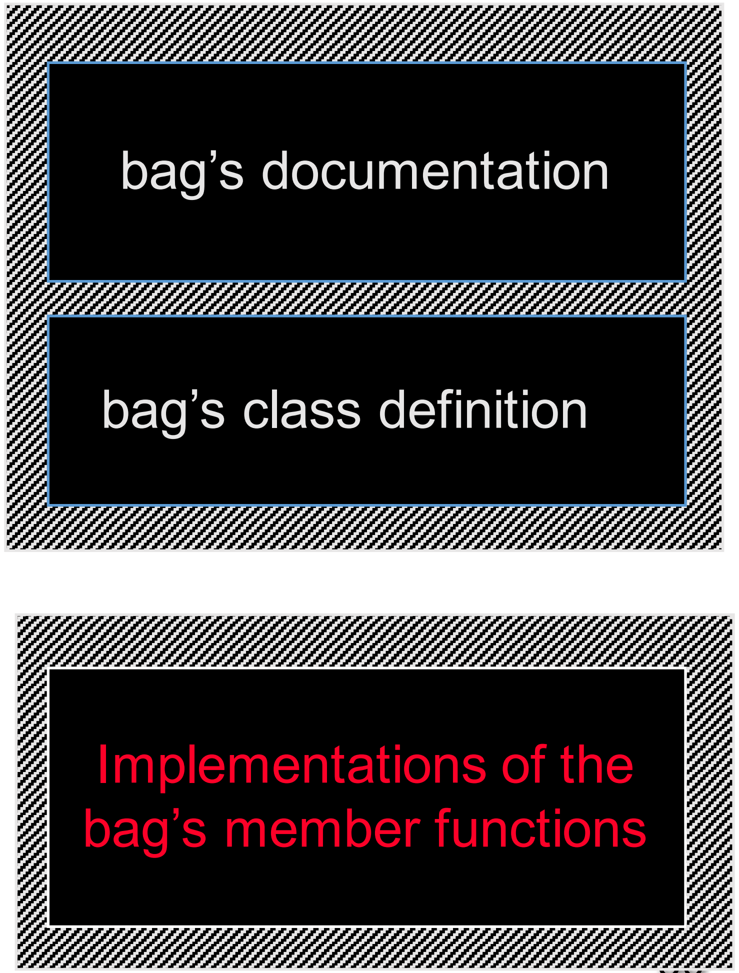
- After the documentation, the header file has the class definition that we’ve seen before:



```
class bag
{
public:
        bag(  );
        void insert(...
        void erase(...
        ...and so on
private:
    ...
};
```

The Implementation File

- As with any class, the actual definitions of the member functions are placed in a separate implementation file.
- The **implementations** of the bag's member functions are in bag1.cxx.



A Quiz

Suppose that a Mysterious Benefactor provides you with the bag class, but you are only permitted to read the documentation in the header file. You cannot read the class definition or implementation file. Can you write a program that uses the bag *data type ?*

① Yes I can.

② No. Not unless I see the class definition for the bag .

③ No. I need to see the class definition for the bag , and also see the implementation file.

A Quiz

Suppose that a Mysterious Benefactor provides you with the Bag class, but you are only permitted to read the documentation in the header file. You cannot read the class definition or implementation file. Can you write a program that uses the bag *data type ?*

① Yes I can.

You know the name of the new data type, which is enough for you to declare bag variables. You also know the headings and specifications of each of the operations.

Using the **bag** in a Program

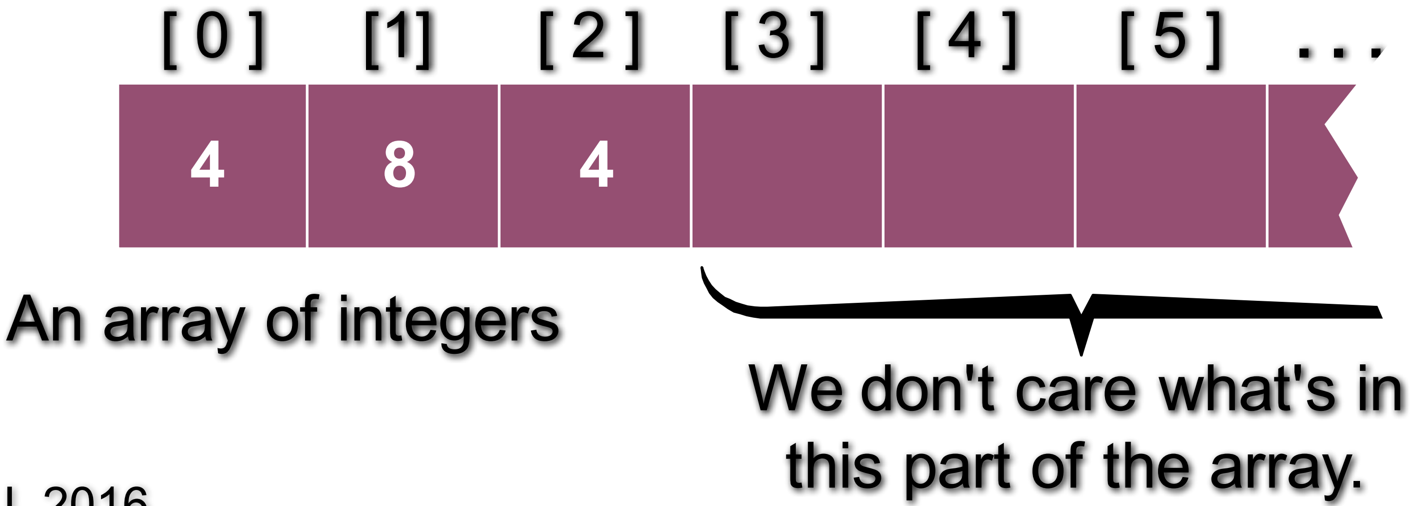
- Here is typical code from a program that uses the new bag class:

```
bag  ages;

// Record the ages of three children:
ages.insert(4);
ages.insert(8);
ages.insert(4);
```

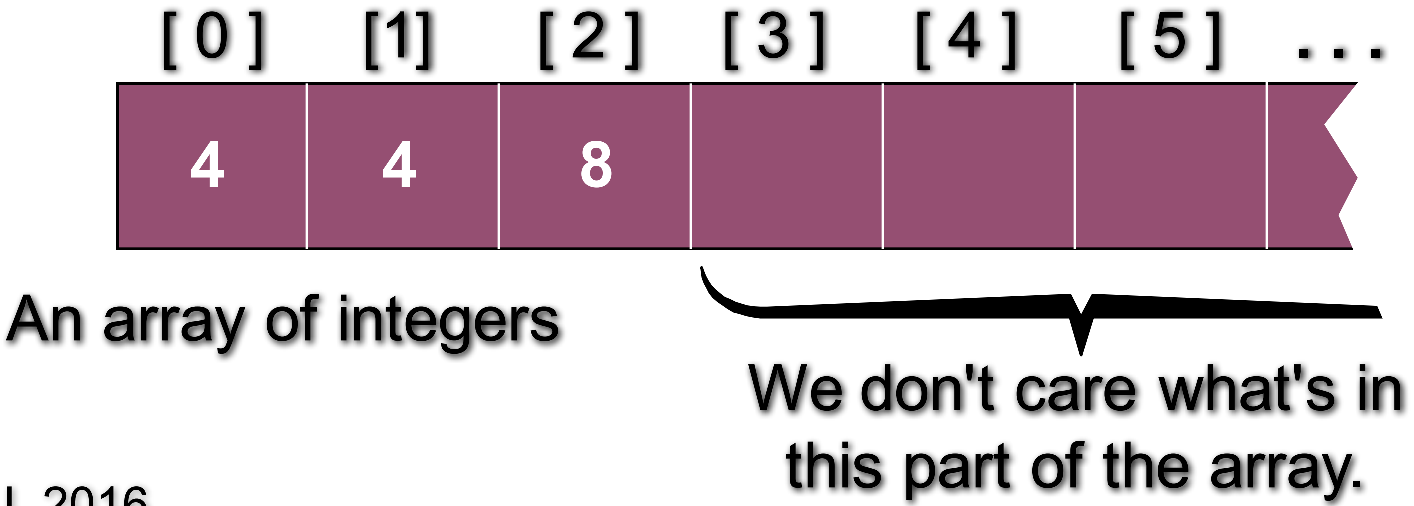
Implementation Details

- The entries of a bag will be stored in the front part of an array, as shown in this example.



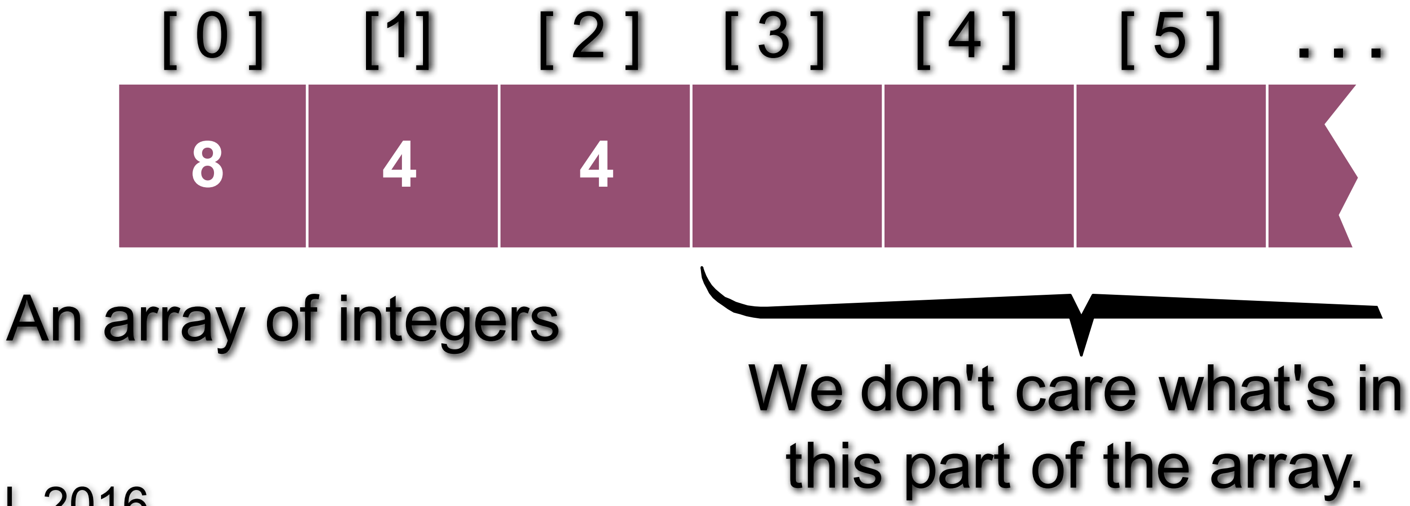
Implementation Details

- The entries may appear in any order. This represents the same bag as the previous one. . .



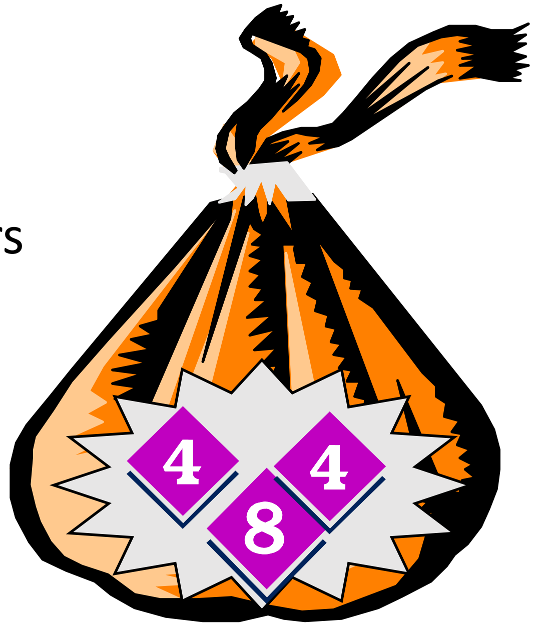
Implementation Details

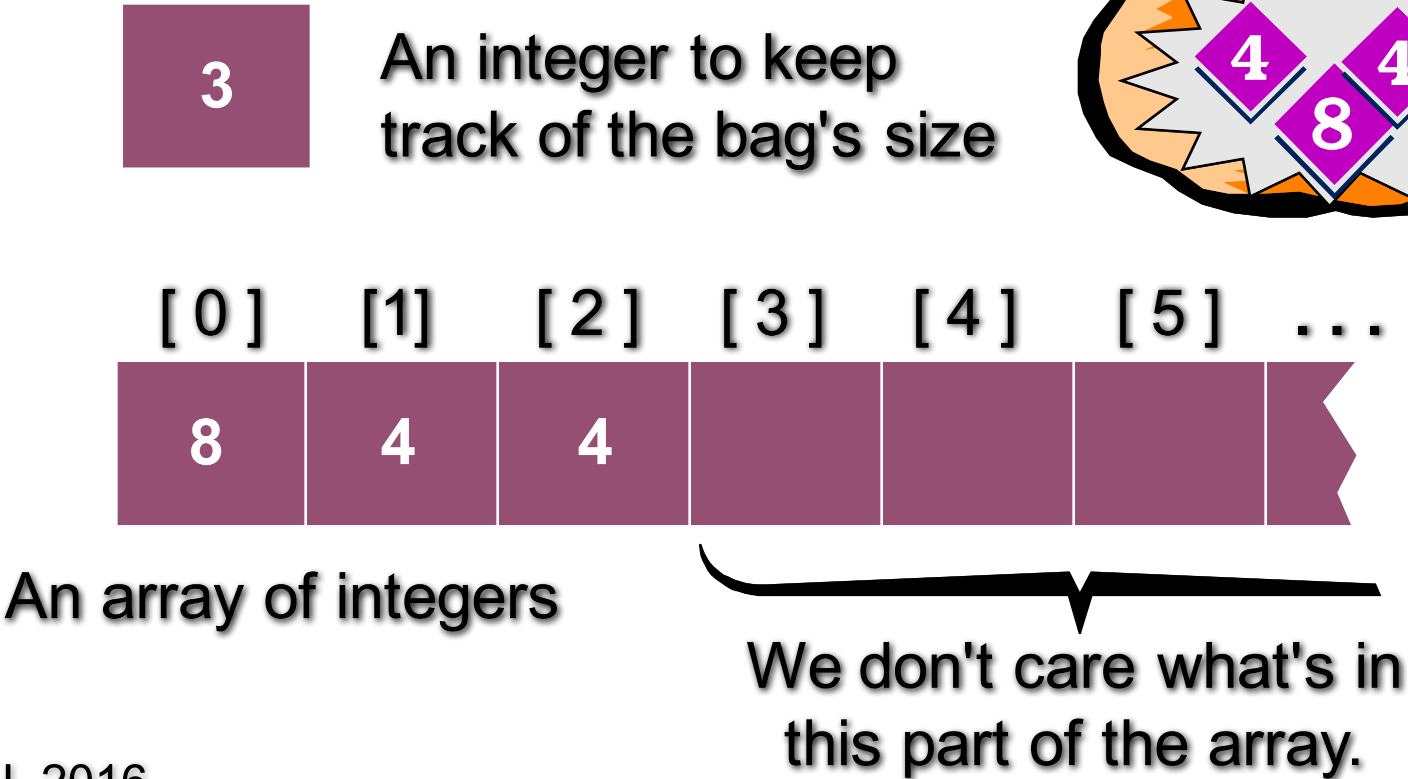
- . . . and this also represents the same bag.



Implementation Details

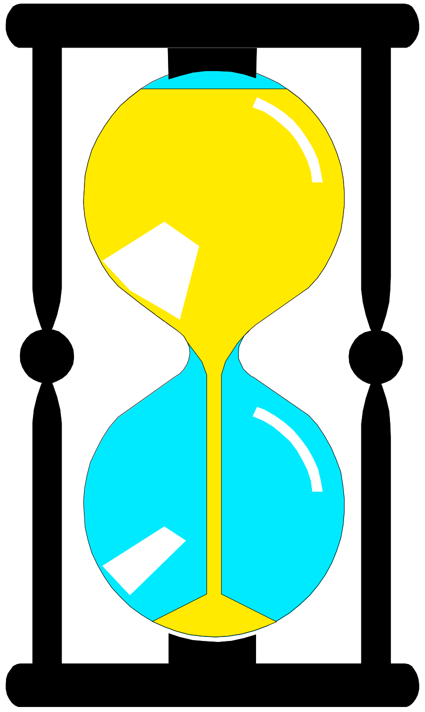
- We also need to keep track of how many numbers are in the bag.




3
An integer to keep track of the bag's size
[0]
[1]
[2]
[3]
[4]
[5]
. . .
8
4
4
An array of integers
We don't care what's in this part of the array.

An Exercise

Use these ideas to write a list of private member variables could implement the bag *class. You should have two member variables. Make the bag capable of holding up to 20 integers.*

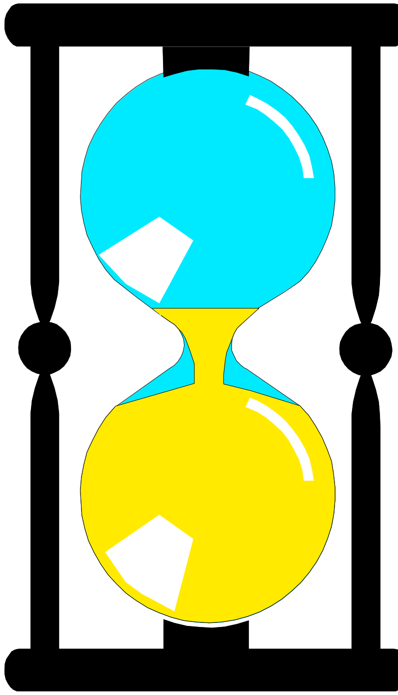


You have 60 seconds to write the declaration.

An Exercise

One solution:

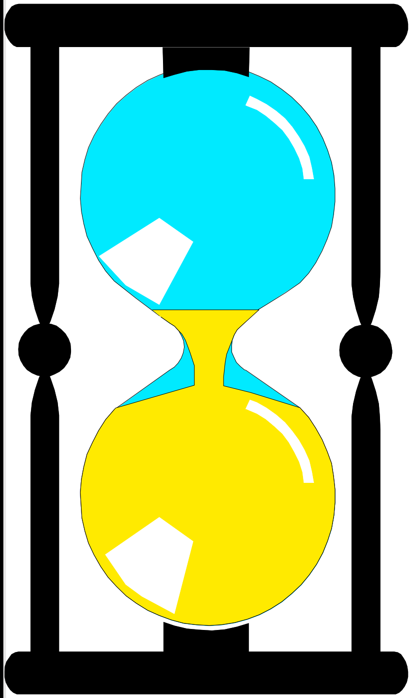
```
class bag
{
public:
     ...
private:
     int  data[20];
     size_t used;
};
```



An Exercise

A more flexible solution:

```
class bag
{
public:
    static const size_t CAPACITY = 20;
    ...
private:
    int  data[CAPACITY];
    size_t used;
};
```



The Invariant of a Class

- Two rules for our bag implementation
  - The number of items in the bag is stored in the member variable used;
  - For an empty bag, we don't care what is stored in any of data; for a non-empty bag, the items are stored in data[0] through data[used-1], and we don't care what are stored in the rest of data.
- The rules that dictate how the member variables of a (bag) class are used to represent a value (such as a bag of items) are called the invariant of the class

The Invariant of a Class

- The invariant of the class is essential to the correct implementation of the class's functions
- In some sense,
  - the invariant of a class is a condition that is an *implicit* part of every function's postcondition
  - And (except for the constructors) it is also an *implicit* part of every function's precondition.

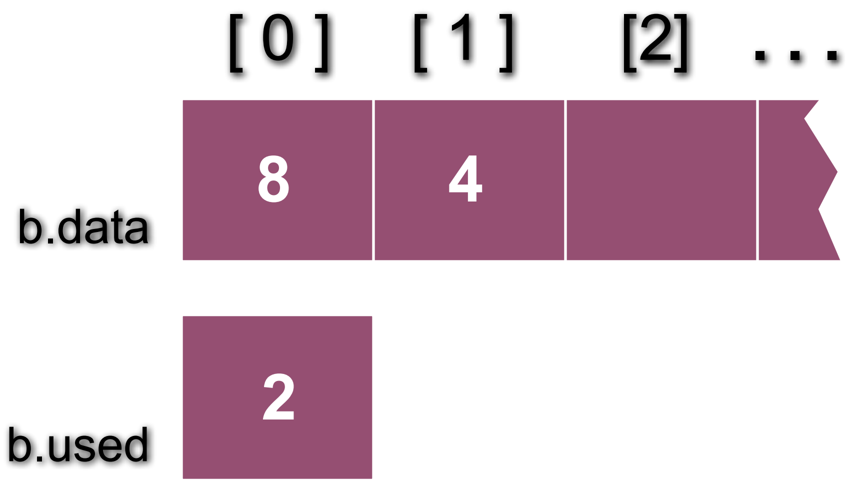
The Invariant of a Class

- Precondition and Postcondition
  - contract for each function, for use of the function
  - document pre- and post- in the header file
- The invariant of the class
  - *implicit* part of pre- and post- so is not usually written as an *explicit* part of pre- and post-
  - about the private member variables, thus for implementation, but not for how to use them
  - documented in the implementation file
- Value Semantics
  - both for implementation and for use
  - documented in the header file

An Example of Calling **insert**

```
void bag::insert(const int& new_entry)
```

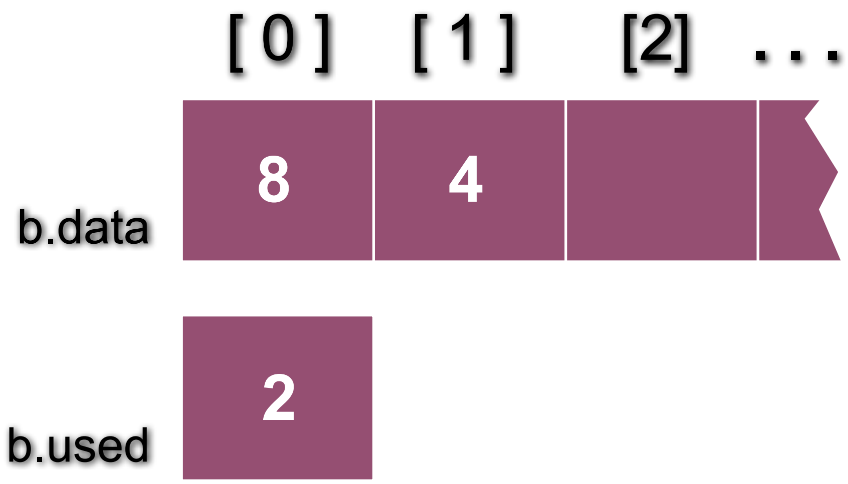
Before calling insert, we might have this bag b:



An Example of Calling **insert**

```
void bag::insert(const int& new_entry)
```

We make a function call
b.insert(17)

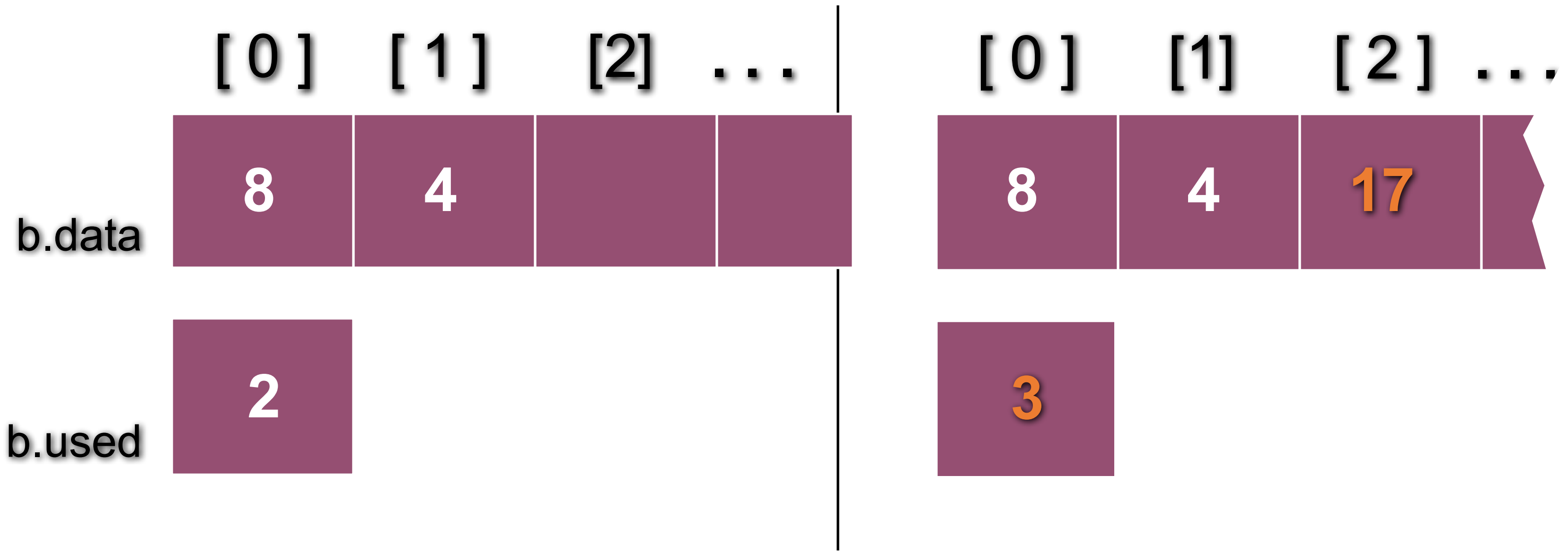


What values will be in b.data and b.count after the member function finishes ?

An Example of Calling **insert**

```
void bag::insert(const int& new_entry)
```

After calling b.insert(17), we will have this bag b:



Pseudocode for bag::insert

```
❶ assert(size( ) < CAPACITY);
❷ Place new_entry in the appropriate location of the data
   array.
❸ Add one to the member variable count.
```

What is the "appropriate location" of the data array ?

Pseudocode for bag::insert

❶ assert(size( ) < CAPACITY);

❷ Place new_entry in the appropriate location of the data array.

❸ Add one to the member variable count.

```
data[used]  = new_entry;
used++;
```

Pseudocode for bag::insert

❶ assert(size( ) < CAPACITY);

❷ Place new_entry in the appropriate location of the data array.

❸ Add one to the member variable count.

```
data[ used++]  = new_entry;
```

Summary

- A container class is a class that can hold a collection of items.
- Container classes can be implemented with a C++ class.
- The class is implemented with
  - a header file (containing documentation and the class definition) bag1.h and
  - an implementation file (containing the implementations of the member functions) bag1.cxx.
- Other details are given in Section 3.1, which you should read, especially the real bag code

Outline for Lecture 5

- Bag class definition/implementation details
  - Inline functions
    - constructor, size
  - Other basic functions
    - insert, erase_one, erase, count
  - More advanced functions
    - operators +, +=, -
- Time Analysis
  - Big-O
- Introduction to sequence

The Other bag Operations

- Read Section 3.1 for the implementations of the other bag member functions
  - such as operators append (+=) and union (+)
- Remember: If you are just **using** the bag class
  - then you don't need to know how the operations are implemented.
- Later we will **reimplement** the bag using more efficient techniques.
- We'll also have a few other operations to manipulate bags.

Append Operator +=

```
void bag::operator+=(const bag& addend)
//     Precondition:  size( ) + addend.size( ) <= CAPACITY.
//     Postcondition: Each item in addend has been added to this bag.

{
  size_t i;
  assert(size( ) + addend.size( ) <= CAPACITY);
  for (i = 0; i< addend.used; ++i)
  {
        data[used] = addend.data[i];
        ++used;
  }
}

// calling program: a  += b;  (OKAY)
// Question : What will happen if you call: b += b;
```

Append Operator +=

```
void bag::operator+=(const bag& addend)
//     Precondition:  size( ) + addend.size( ) <= CAPACITY.
//     Postcondition: Each item in addend has been added to this bag.
//    Library facilities used: algorithm, cassert
{
  assert(size( ) + addend.size( ) <= CAPACITY);

  copy(addend.data, addend.data + addend.used, data + used);
  used += addend.used;
}

// copy (<beginning location>, ending location>, <destination>);
// Question : Can you fix the bug in the previous slide without using copy ?
```

@Feng HU, 2016

Union Operator +

```
// NONMEMBER FUNCTION for the bag class:
bag operator+(const bag& b1, const bag& b2)
//     Precondition:  b1.size( ) + b2.size( ) <= bag::CAPACITY.
//     Postcondition: The bag returned is the union of b1 and b2.
//     Library facilities used: cassert
{
       bag answer;

       assert(b1.size( ) + b2.size( ) <= bag::CAPACITY);

       answer += b1;
       answer += b2;
       return answer;
}

// calling program:  c =a+b;
// Question : what happens if you call a =a+b ?
```

@

Subtract Operator -

```
// Prototype: NONMEMBER friend FUNCTION for the bag class:
// bag operator-(const bag& b1, const bag& b2);
//     Postcondition: For two bags b1 and b2, the bag x-y contains all the
   items of x, with any items from y removed
// Write your implementation
// HINTS:
// 1. A friend function can access private member variables of a bag
// 2. You cannot change constant reference parameters
// 3. You may use any member functions of the bag class such as
//        b1.count(target); // how many target is in bag b1?
//        b1.erase_one(target); // target is an integer item
//        b2.size(); // size of the bag b2;
//           bag b3(b2); // automatic copy constructor
//
```

@

Subtract Operator -

```
// NONMEMBER friend FUNCTION for the bag class:
bag operator-(const bag& b1, const bag& b2)
//     Postcondition: For two bags b1 and b2, the bag x-y contains all the
   items of x, with any items from y removed
{
   size_t index;
   bag answer(b1);  // copy constructor
   size_t size2 = b2.size(); // use member function size
   for (index = 0; index < size2; ++index)
   {
         int target = b2.data[index];  // use private member variable
         if (answer.count(target) ) // use function count
                answer.erase_one(target); // use function erase_one
   }
   return answer;
}
```

Other Kinds of Bags

- In this example, we have implemented a bag containing **integers**.
- But we could have had a bag of **float numbers**, a bag of **characters**, a bag of **strings** . . .

Suppose you wanted one of these other bags. How much would you need to change in the implementation ?
Section 3.1 gives a simple solution using the C++ typedef statement.

Time Analysis of the Bag Class

- count – the number of occurrence
- erase_one – remove one from the bag
- erase – remove all
- += - append
- b1+b2 - union
- insert – add one item
- size – number of items in the bag

What's the most important, then?

- Concept of Container Classes
  - the bag class is not particularly important
- Other kinds of container classes
  - sequence – similar to a bag, both contain a bunch of items. But unlike a bag, the items in a sequence is arranged in order.
  - will be the topic of our second assignment– paying attention to the differences
    - index – have current, next, last, etc
    - member functions and their implementation (e.g. insert, attach)
    - time analysis (insert)

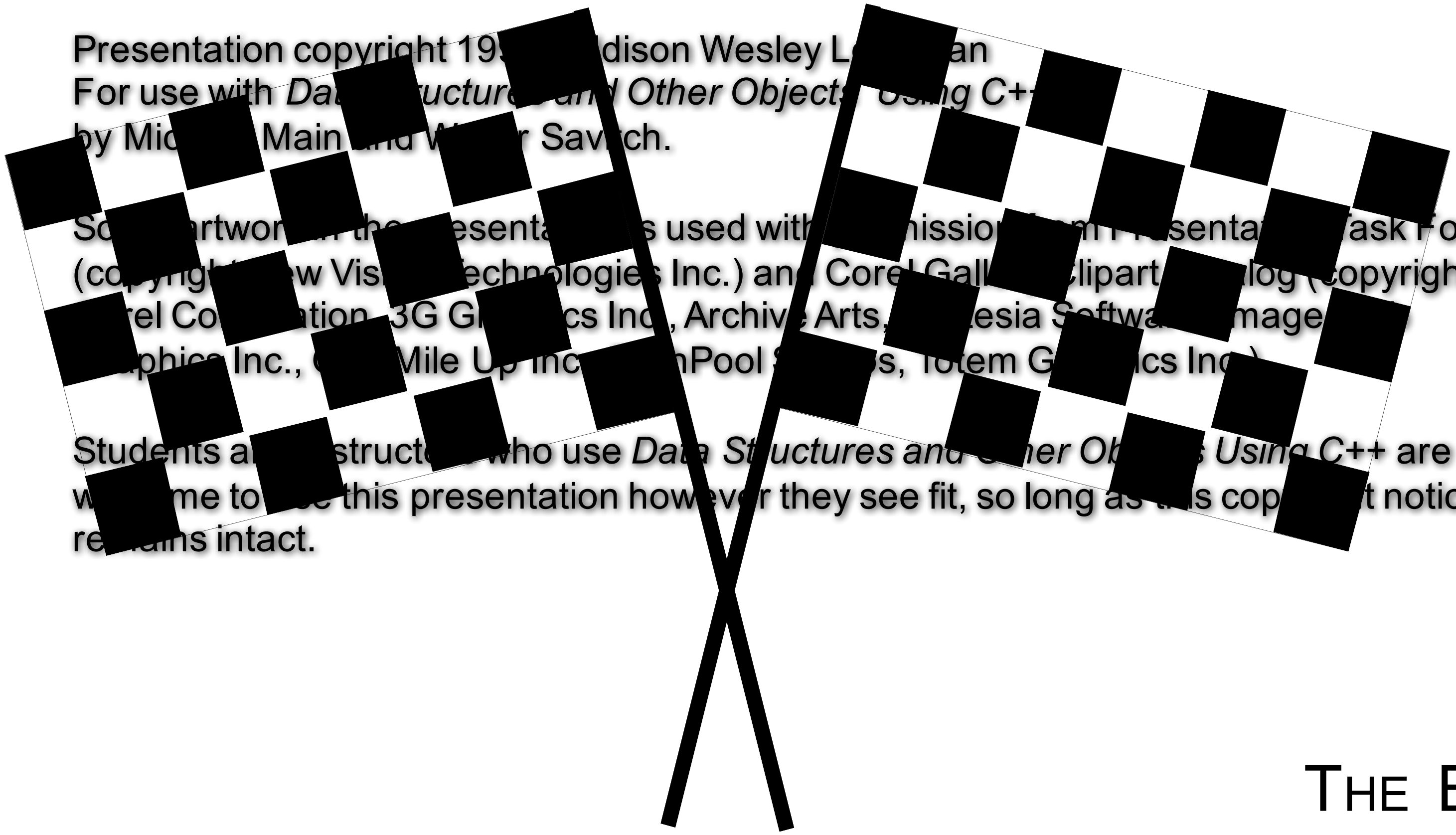
After Class…

- **Assignment 2**
  - **Due Wednesday, Sept 28**
  - Reading: Chapter 3, Section 3.2-3.3
  - especially the sequence code
- Self-Test Exercises
  - 1,3, 5,10,11,14,18-24
- Reading for next lecture
  - Chapter 4, Section 4.1-4.2

Summary

- A container class is a class that can hold a collection of items.
- Container classes can be implemented with a C++ class.
- The class is implemented with
  - a header file (containing documentation and the class definition) bag1.h and
  - an implementation file (containing the implementations of the member functions) bag1.cxx.
- Other details are given in Section 3.1, which you should read, especially the real bag code

This lecture was modified from the authors' presentation, with new conventions provided in the second edition (2001) of the textbook and other minor changes -- Feng HU, 2016, CCNY



THE END