SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

Container Classes

Learning Objectives

- Design and implement data structures that use partially filled arrays to store a collection of elements
- Use *typedef* statements within the definition of a data structure to specify the data type of the container's elements
- Use static *const* members to define fixed integer information such as the size of an array
- Use the C++ Standard Library copy function to copy arrays
- Writing *invariants* of data structures
- Writing interactive test programs to test a data structure

Introduction

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

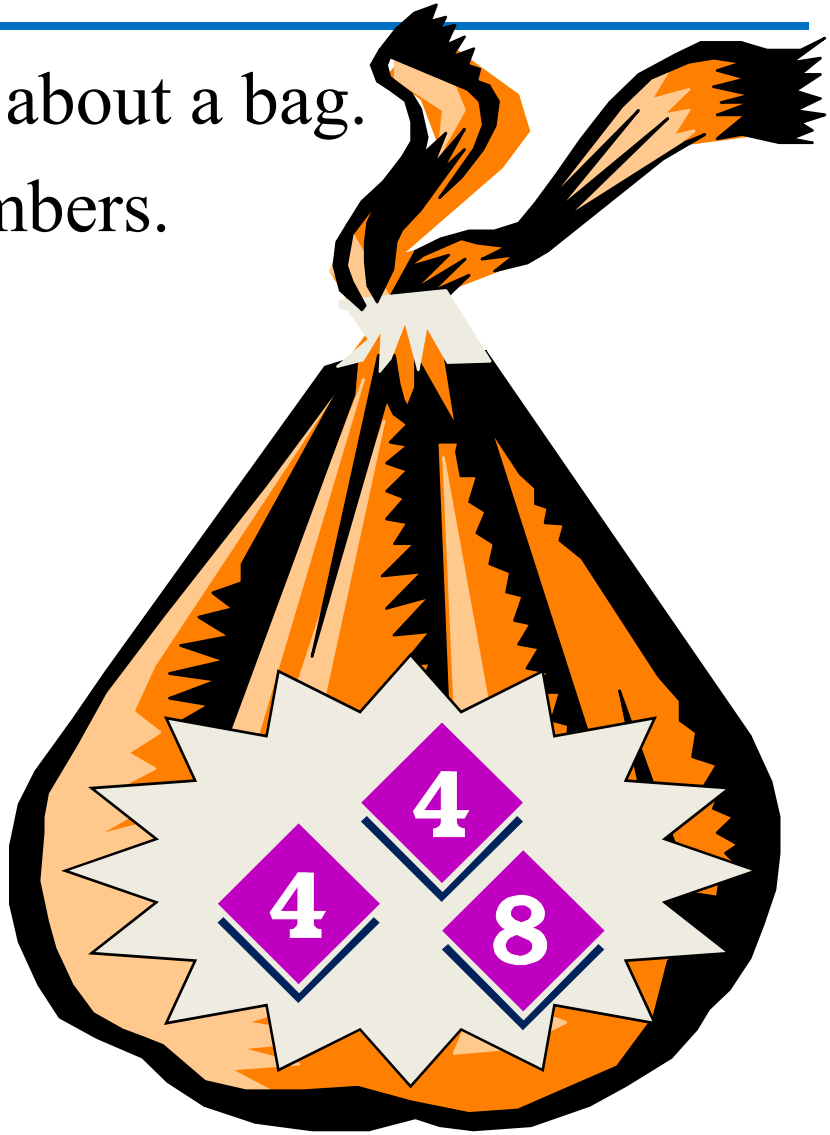
THE BAG CLASS

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

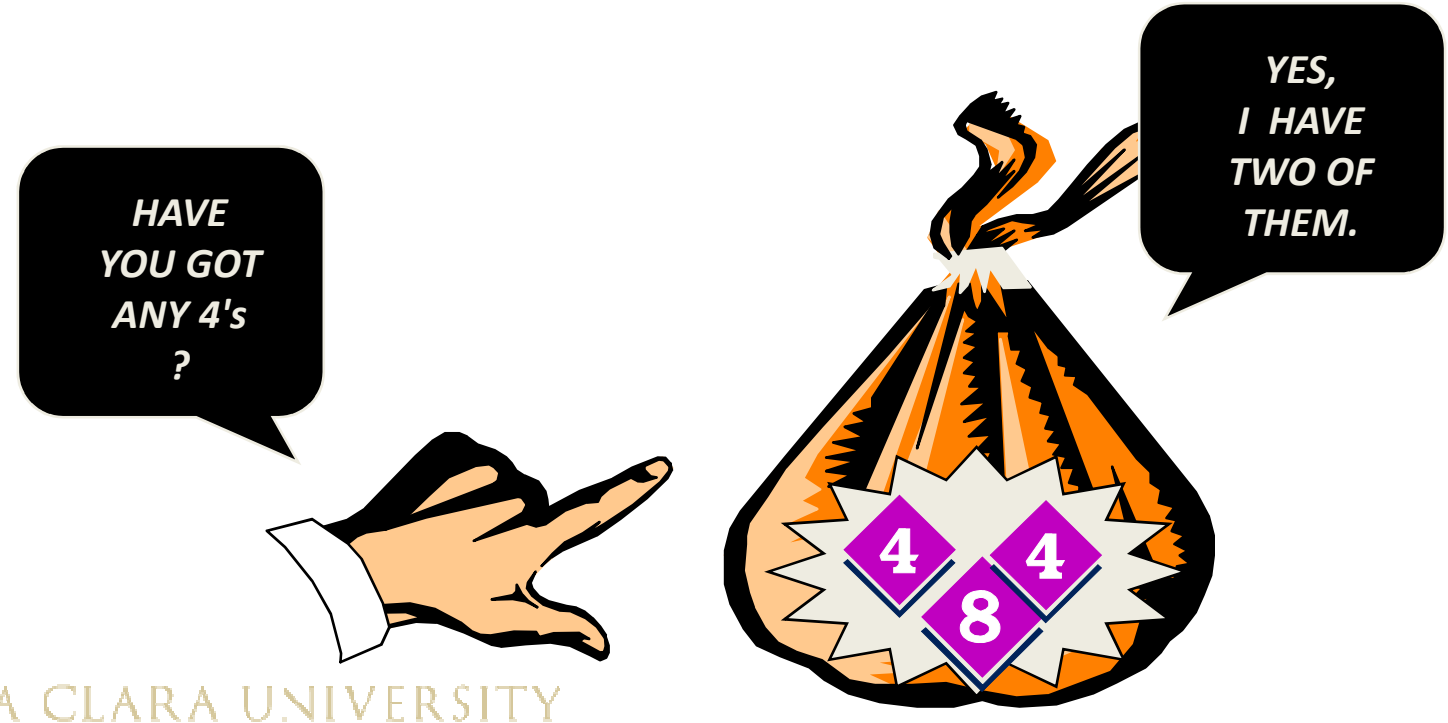
- Numbers may be inserted into a bag.
- The bag can hold many numbers.
- We can even insert the same number more than once.

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.

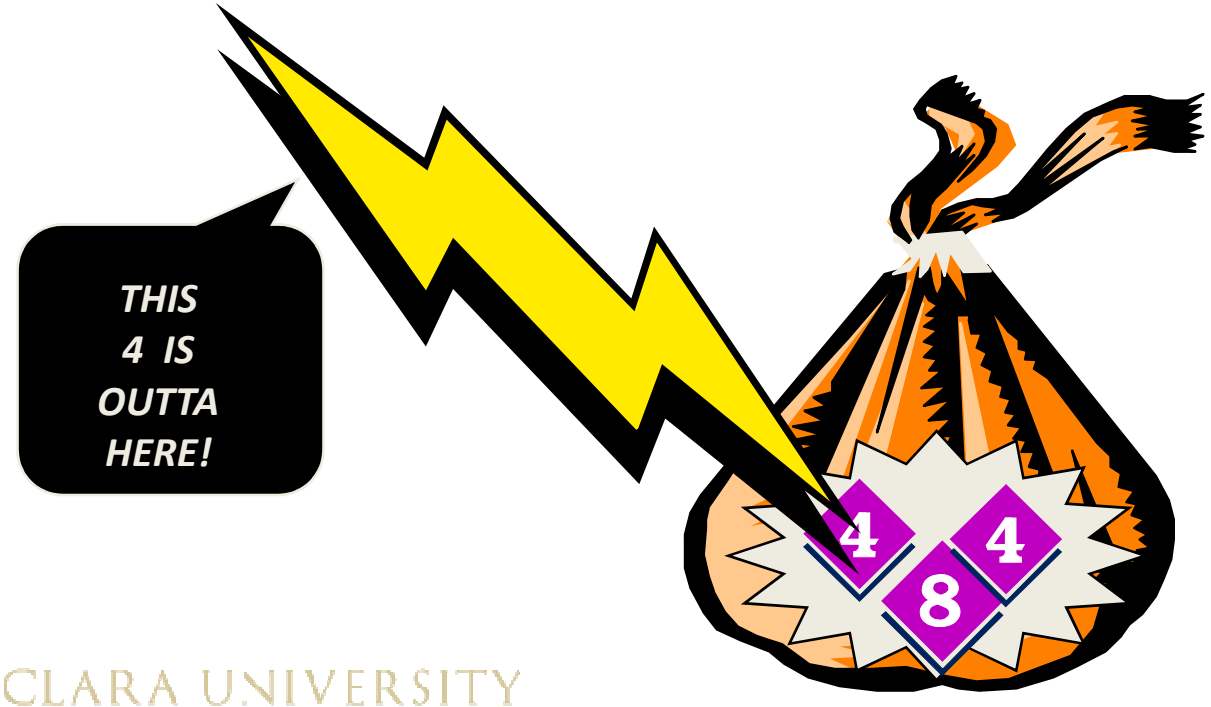
Examining a Bag

- 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.

How Many Numbers

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

Summary of the Bag Operations

- A bag can be put in its **initial state**, which is an empty bag.
- Numbers can be **inserted** into the bag.
- You may **count** how many occurrences of a certain number are in the bag.
- Numbers can be **erased** from the bag.
- You can check the **size** of the bag (i.e. how many numbers are in the bag).

The Bag Class

- C++ classes 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 Class - Specification

- Constructor
  - A default constructor to initialize a bag to be empty
  - The name of the constructor must be the same as the name of the class itself, so the prototype is: bag();
- Value semantics
  - A new bag can be initialized as a copy of another bag, using the **copy constructor**
  - Bag objects can be copied with an **assignment statement**

```
bag b;
b.insert(42);
bag c(b);
bag d = c;
```

The Bag Class – Specification (cont.)

- typedef for the value_type
  - Use name *value_type* for the data type of the items in a bag

```
class bag
{
public:
     typedef int value_type;
     ...
```

The Bag Class – Specification (cont.)

- typedef for the size_type
  - Define another data type for variables that keep track of how many items are in a bag
  - This type will be called *size_type*, with its definition near the top of the bag class definition:

```
class bag
{
public:
    typedef int value_type;
    typedef std::size_t size_type;
```

The `size` member function

- Check how many integers are in the bag

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

The `insert` member function

- Insert a new number in the bag

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

The `count` Function

- Count how many copies of a number occur

```
size_type count(const value_type& target) const;
//   Postcondition:  The return value is the number
//   of copies of target in the bag.
```

The `erase_one` member function

- Removes (erase) one copy of a number

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

- `erase` member function?

Union operator

- The union of two bags is a new larger bag that contains all the numbers in the first bag plus all the numbers in the second bag
- Overload the + operator as a nonmember function

```
bag operator+(const bag& b1, const bag& b2);
```

Overloading the += operator

- The overloaded += will allow us to add the contents of one bag to the existing contents of another bag
  - Similar to the same way that += works for integers or real numbers
- The prototype of the member function:

```
void operator +=(const bag& addend);
```

- Example:

```
bag first_bag, second_bag;
first_bag.insert(8);
second_bag.insert(4);
second_bag.insert(8);
first_bag += second_bag;
```

=> first_bag contains one 4 and two 8s

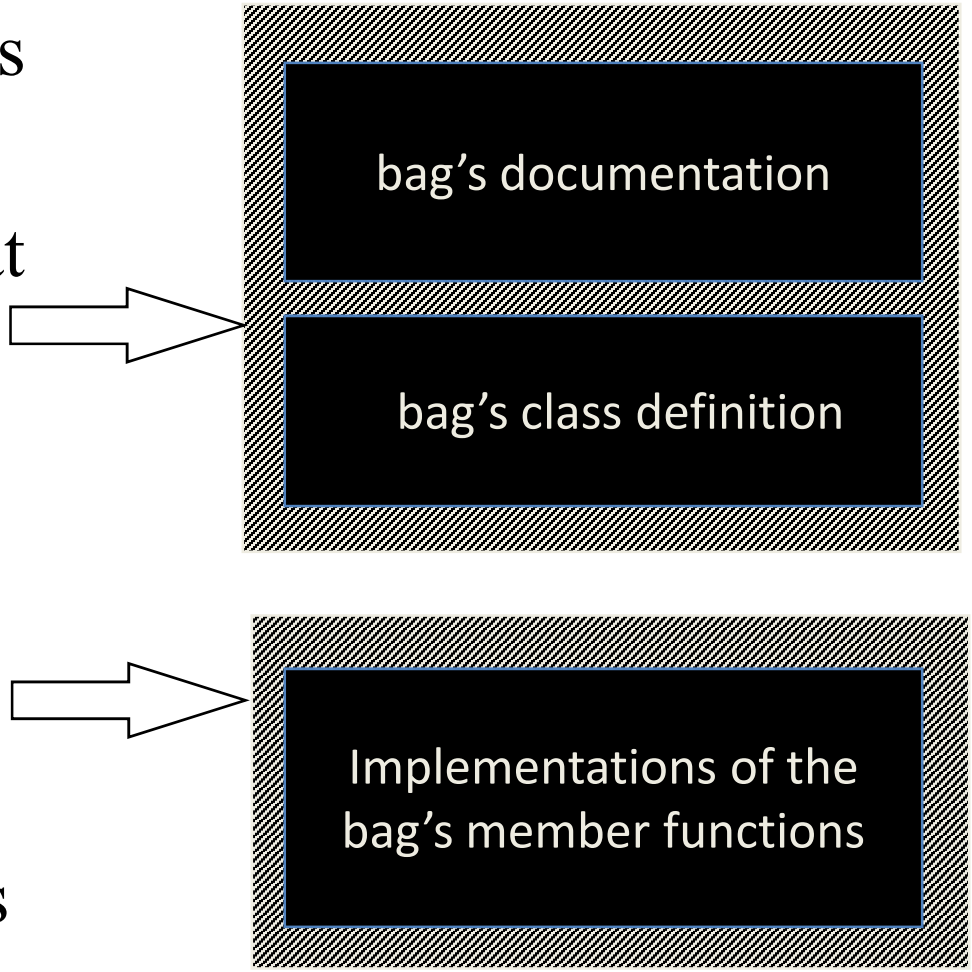
The bag's CAPACITY

- We want to implement a bounded bag
- Define CAPACITY as a **static member constant**
- All of the class's objects use the same value
- Value of CAPACITY is defined once and cannot be changed while the program is running

```
class bag
{
public:
      typedef int value_type;
      typedef std::size_t size_type;
      static const size_type CAPACITY = 30;
      ...
```

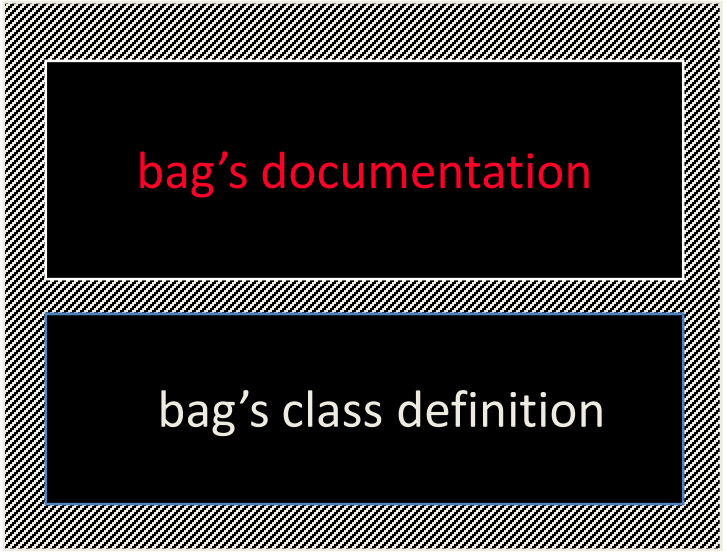
The Bag Class - Documentation

- 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

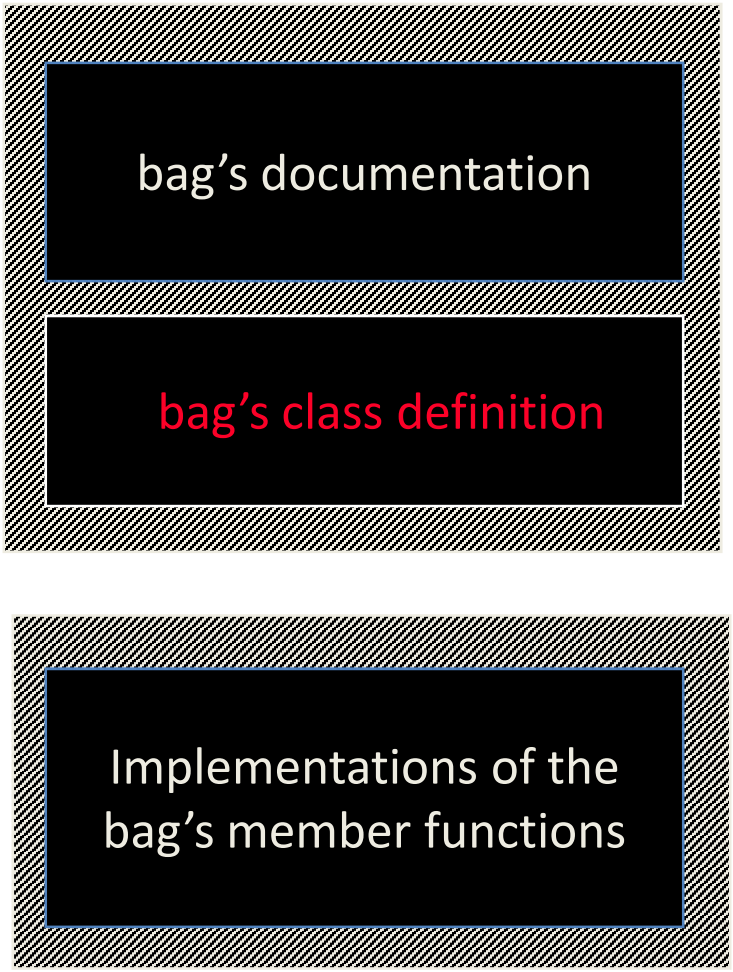
- The documentation gives **prototypes and specifications** for the bag member functions
- Specifications are written as **precondition/postcondition** contracts
- Everything needed to **use** the `bag` class is included in this comment



The bag's Class Definition

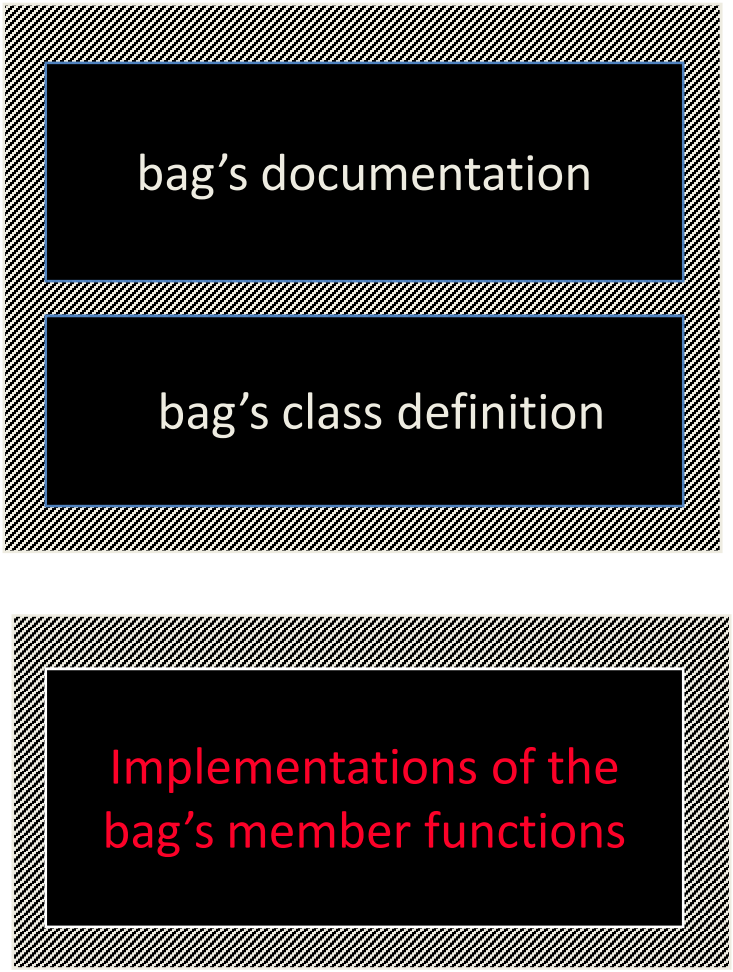
- After the documentation, the header file has the class definition:

```
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 ?

- ✶ Yes, I can.
- ✸ No. Not unless I see the class definition for the `bag` .
- ✹ 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 ?

★ 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 perations.

Using the bag in a Program

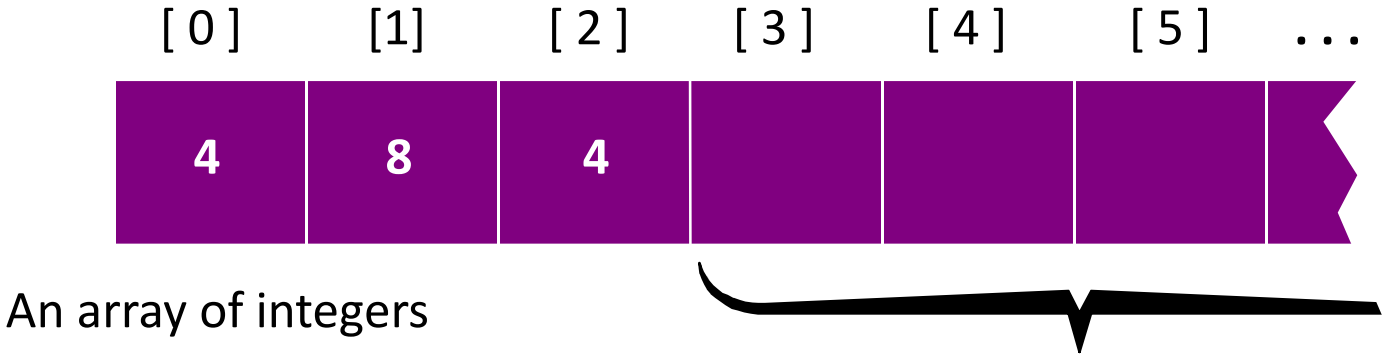
- Here is typical code from a program that uses the new `bag` class:

```
bag ages;

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

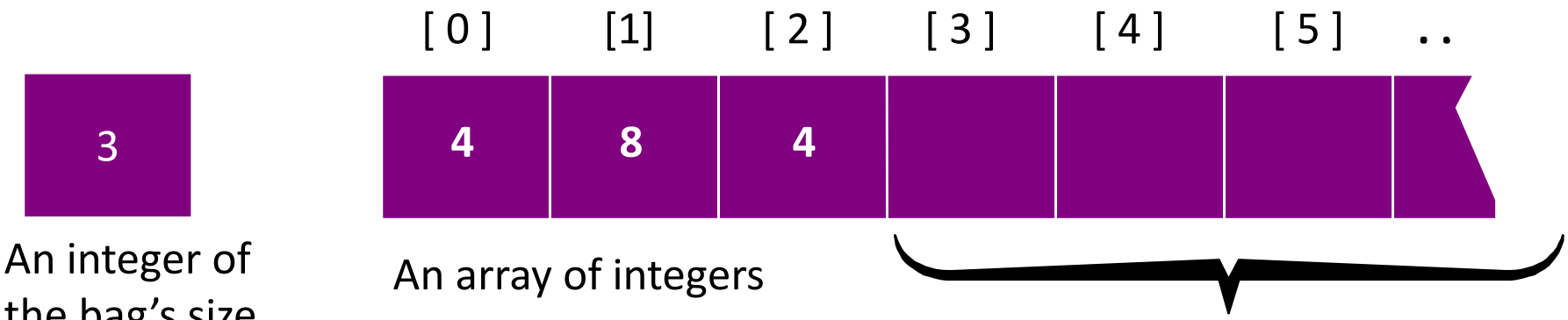
The Bag Class - Design

- There are several ways to design the bag class
- For now, we design a somewhat inefficient data structure using an array
- We use the *beginning part* of a large array to store entries of a bag
  - Such an array is called a **partially filled array**



The Bag Class – Design (cont.)

- This array will be one of the private member variables
- The length of the array will be determined by the constant CAPACITY
- The bag class must keep track of one other item: How much of the array is currently being used?
- We will keep track of the amount in a private member variable called `used`



The Bag Class – Design (cont.)

```
class bag
{
public:
    // TYPEDEFS and MEMBER CONSTANTS
    typedef int value_type;
    typedef std::size_t size_type;
    static const size_type CAPACITY = 30;

    The rest of the public members will be listed later

private:
    value_type data[CAPACITY];// The array to store items
    size_type used;           // How much of array is used
};
```

The Invariant of a Class

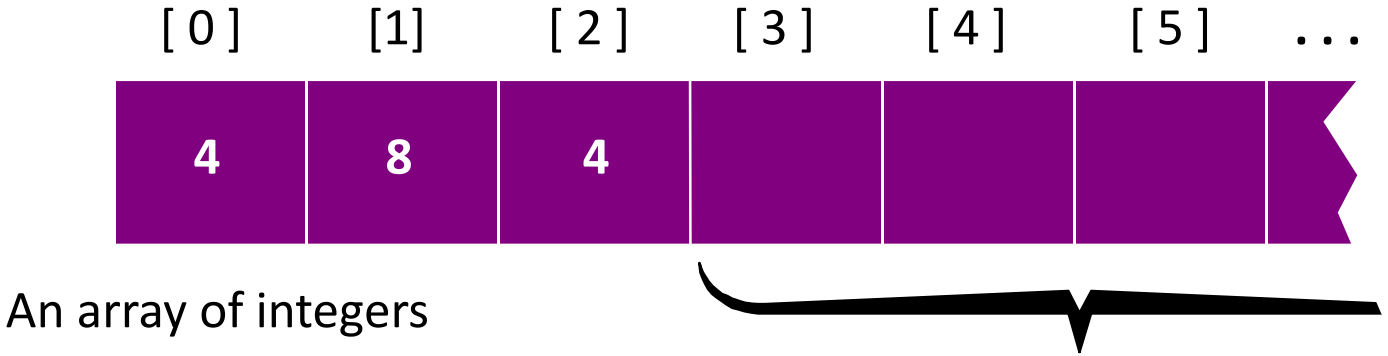
- Two rules for our bag implementation
  - The number of items in the bag is stored in the member variable `used`;
  - For an empty bag, we don't care what is stored in any of `data`; for a non-empty bag, the items are stored in `data[0]` through `data[used-1]`, and we don't care what are stored in the rest of `data`.

The Invariant of a Class (cont.)

- *Invariant of a class*: The rules that dictate how the member variables of a (bag) class represent a value
  - Essential to the correct implementation of the class's functions
  - A condition that is an *implicit* part of every function's postcondition
  - Except for the constructors, also an *implicit* part of every function's precondition

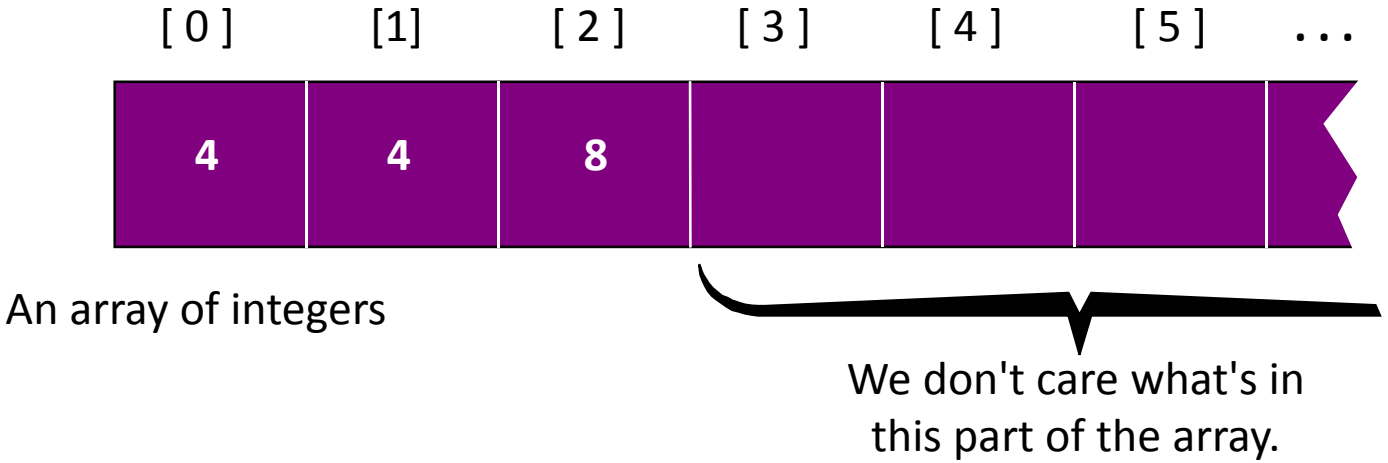
Implementation Details

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



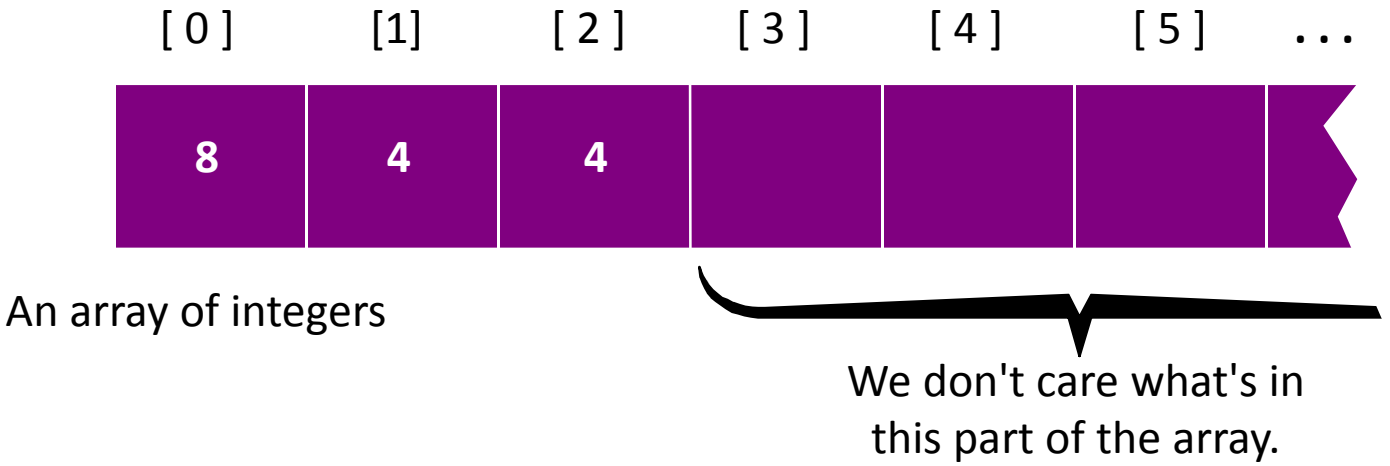
Implementation Details

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



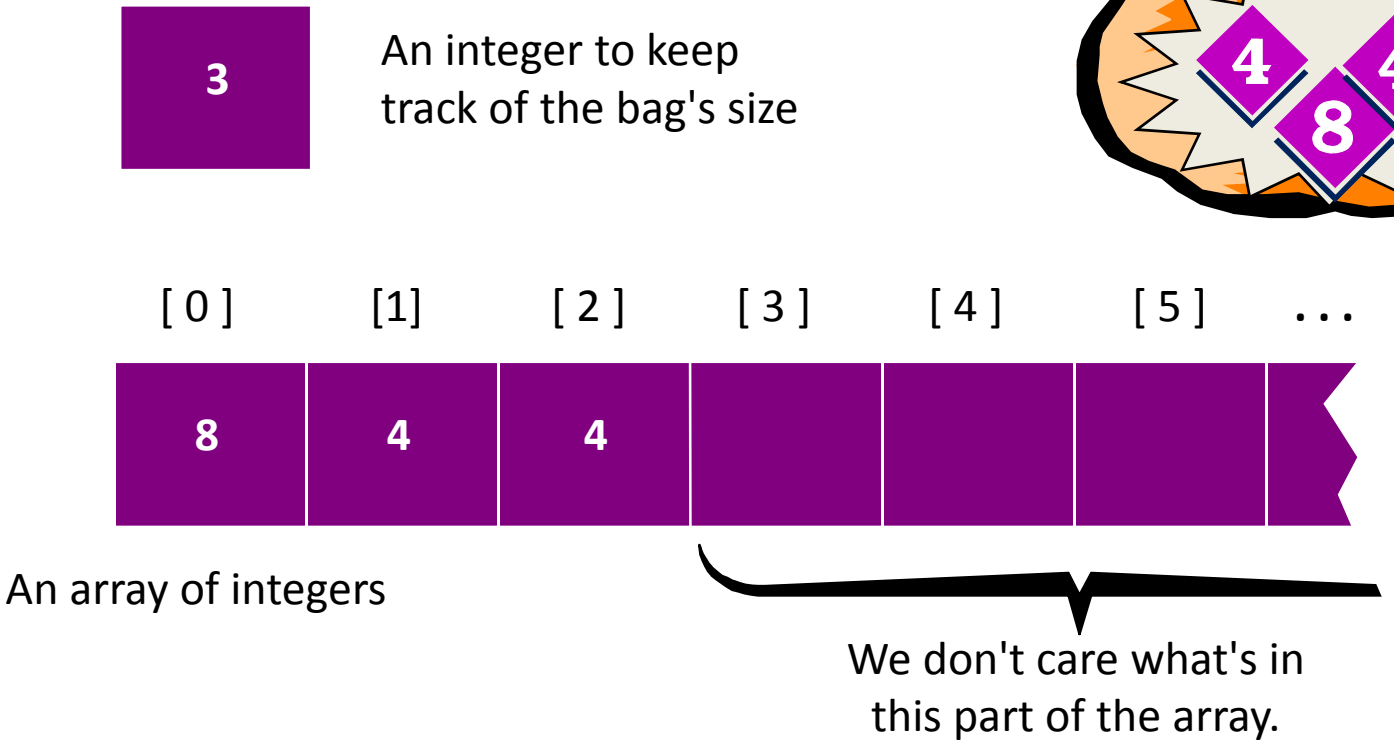
Implementation Details

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



Implementation Details

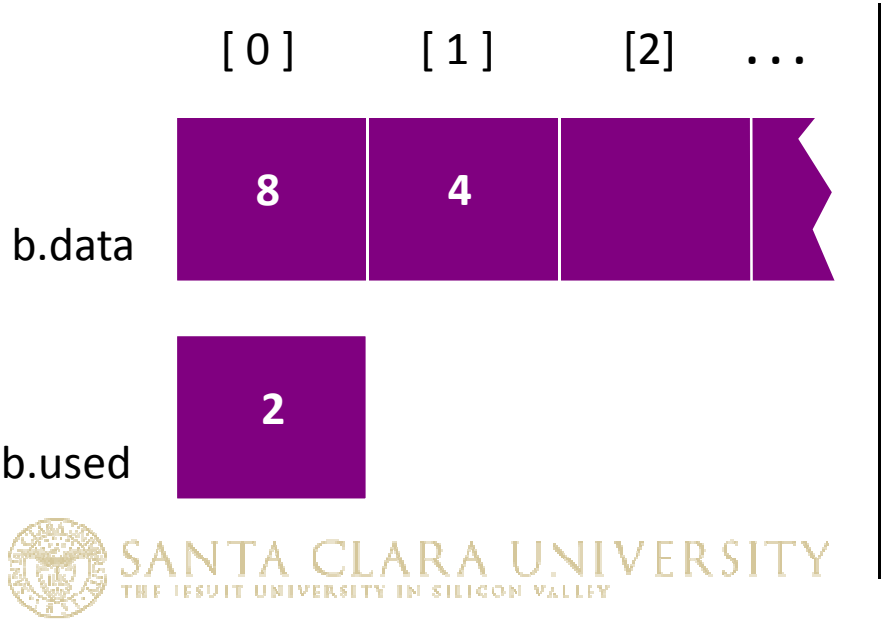
- We also need to keep track of how many numbers are in the bag.



Example of Calling insert

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

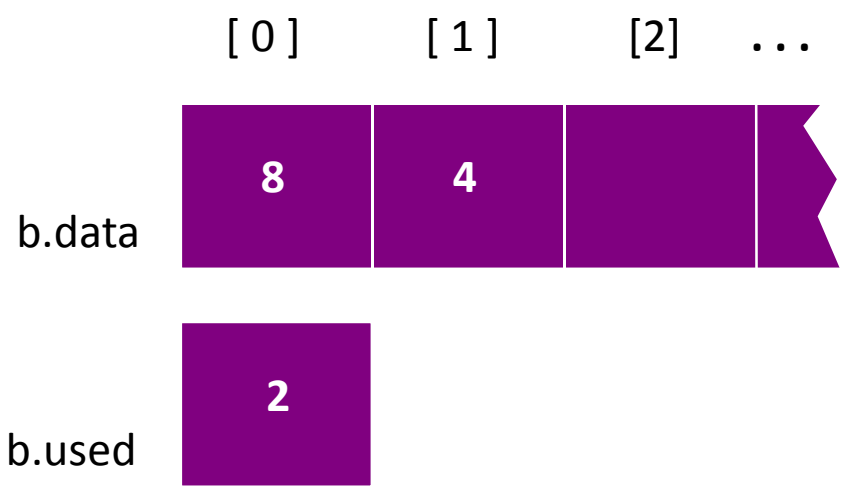
Before calling insert, we might have this bag b:



Example of Calling insert (cont.)

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

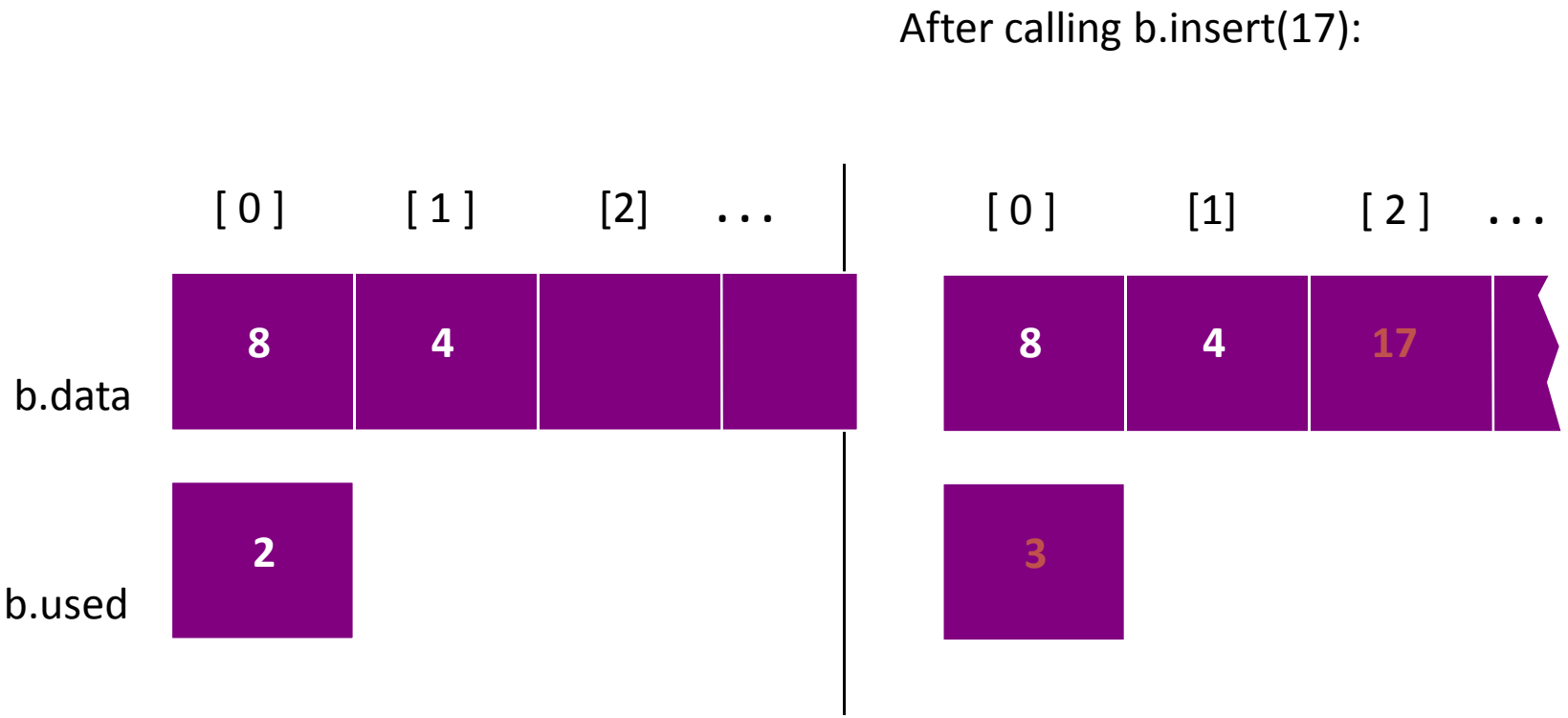
A function call
b.insert(17)



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

Example of Calling insert (cont.)

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



Pseudocode for bag::insert

- `assert(size( ) < CAPACITY);`
- Place `new_entry` in the appropriate location of the `data` array.
- Add one to the member variable `used`.

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

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

The Other bag Operations

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 ?
```

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 ?
```

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 ?

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

PROGRAMMING PROJECT: THE SEQUENCE CLASS

The Sequence Class

- A sequence class is similar to a bag—both contain a bunch of items, but unlike a bag, **the items in a sequence are arranged in an order**
- In contrast to the bag class, the member functions of a sequence will allow a program to **step through the sequence one item at a time**
- Member functions also permit a program to **control precisely where items are inserted and removed within the sequence**
- The technique of using member functions to access items is called an **internal iterator**, which differs from external iterators of the Standard Library containers

The Sequence Class - Specification

```
class sequence
{
public:
        // TYPEDEF and MEMBER CONSTANTS
        typedef double value_type;
        typedef std::size_t size_type;
        static const size_type CAPACITY = 30;
        ...
```

- Default constructor
- The size member function

The Sequence Class – Specification (cont.)

- Member functions to examine a sequence

```
void start( );
value_type current( ) const;
void advance( );
```

A sequence named `numbers` contains 37, 10, 83, and 42. What is output?

```
numbers.start( );
cout << numbers.current( ) << endl;
numbers.advance( );
cout << numbers.current( ) << endl;
numbers.advance( );
cout << numbers.current( ) << endl;
```

The Sequence Class – Specification (cont.)

- Member functions to examine a sequence

```
bool is_item( ) const;
```

```
for (numbers.start( ); numbers.is_item( ); numbers.advance( ))
cout << numbers.current( ) << endl;
```

The Sequence Class – Specification (cont.)

- The insert and attach member functions

```
void insert(const value_type& entry);

void attach(const value_type& entry);
```

- The remove_current member function

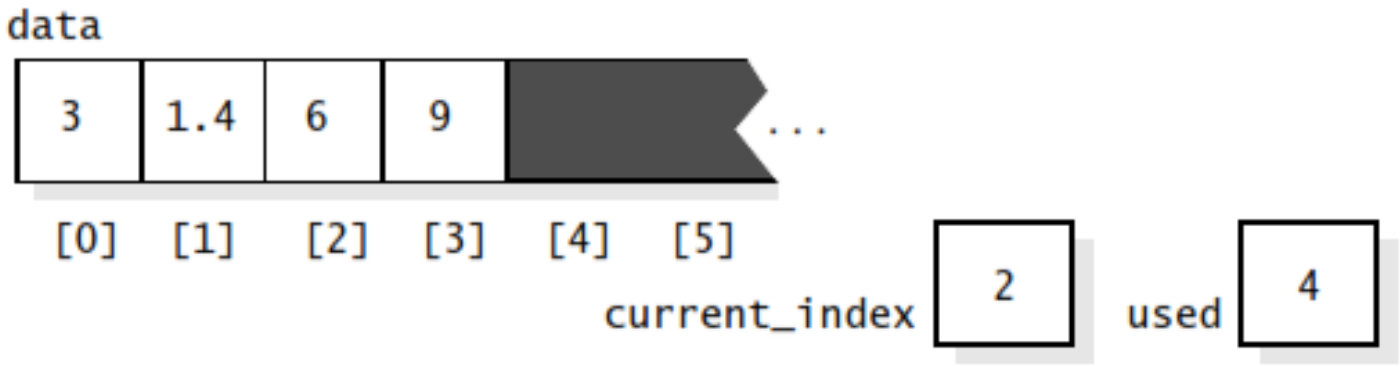
```
void remove_current( );
```

The Sequence Class - Design

- Member variables
  - `data`
  - `used`
  - `current_index`

- Class invariant
  - The number of items in the sequence 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 is stored in the rest of `data`.
  - If there is a current item, it lies in `data[current_index]`; if there is no current item, then `current_index` equals used.

The Sequence Class - Design



The Sequence Class - Pseudocode

The Sequence Class - Implementation

INTERACTIVE TEST PROGRAM

Interactive Test Program for the Sequence Class

- A small test program that provides user a menu of choices with letters or other meaningful characters to allow the user to select a choice like:
  - ! Activate the start() function
  - + Activate the advance() function
  - S Print the result from the size() function
  - I Insert a new number with the insert() function
  - …
- Based on the user choice the program will take some actions on a sequence object like:
  - Output: Size is 1
- The test program uses two techniques:
  - Converting input to uppercase letters
  - Acting on the input via a switch statement

Copyright Notice

Part of this lecture was adapted from the slides of Dr. Behnam Dezfouli