COEN 79
Object-Oriented Programming and Advanced Data Structures
Department of Computer Engineering, Santa Clara University
Behnam Dezfouli, PhD

Container Classes

Slide Set: 3

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 `copy` function to copy arrays
- Writing **invariants** of data structures
- Writing interactive test programs to test a data structure

The Bag Class

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 check **how many occurrences of a certain number** are in the bag
- Numbers can be **removed** from the bag
- You can check **how many numbers** are in the bag

The Bag Class

- 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 remove(...
       ...and so on
private:

};
```

We'll look at private members later

The Bag Class Specification —The value semantics

- The bag class has 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();`

- A newly declared bag can be initialized as a copy of another bag, using the **copy constructor**
- We require that **bag objects can be copied** with an **assignment operator**

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

The Bag Class

The Bag Class Specification — A `typedef` for the `value_type`

- **Instead of forcing the bag to always include integers,** we use the name **`value_type`** for the data type of the items in a bag

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

- Bag functions can use the name **`value_type`** as a synonym for the data type **`int`**
- **Other functions, which are not bag member functions, can use the name `bag::value_type` as the type of the items in a bag**

The Bag Class

The Bag Class Specification — A `typedef` for the `size_type`

- In addition to the `value_type`, our bag defines **another data type that can be used 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 < Name of the class >
{
public:
    typedef <A data type such as int or double>  <A new name>;
    typedef <an integer type of some kind>  size_type;
    ...
```

The Bag Class

The Bag Class Specification — The `std:size_t` Data Type

- The data type **`size_t`** is an integer data type that can **hold only non-negative numbers**
- C++ implementation guarantees that the values of the `size_t` type are sufficient to hold the size of any variable that can be declared on your machine

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

To use `size_t` in a header file, we must include `cstdlib` and use the full name `std::size_t`

- The actual size of `size_t` is platform-dependent: On 32-bit and 64-bit systems `size_t` will take 32 and 64 bits, respectively

The Bag Class

The Bag Class Specification — The `size` member function

- The bag has a **constant member function** called `size`
- The return value of the `size` function tells how many items are currently in the bag

```
size_type size( ) const;
```

- There is a member function that places a new integer (called `entry`) into a bag

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

The Bag Class

The Bag Class Specification — The `count` member function

- This is a constant member function that determines how many copies of a particular number are in a bag

```
size_type count(const value_type& target) const;
```

- The activation of `count(n)` returns the number of occurrences of `n` in a bag

❑Example:

```
cout << first_bag.count(1) << endl;    Prints 0
cout << first_bag.count(4) << endl;    Prints 1
cout << first_bag.count(8) << endl;    Prints 2
```

The Bag Class Specification — `erase_one` and `erase` member functions

- These two member functions have the following prototypes:

```
bool erase_one(const value_type& target);
size_type erase(const value_type& target);
```

- Provided that the target is actually in the bag, the `erase_one` function **removes one copy of target** and returns `true`
- If target is not in the bag, attempting to erase one copy has no effect on the bag, and the function returns `false`
- The `erase` function **removes all copies of the target**; its return value tells how many copies were removed

The Bag Class Specification — 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

- To implement the union, we will overload the + operator as a **non-member function** with this prototype:

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

The Bag Class

The Bag Class Specification — Overloading the += operator

- The overloaded += will allow us to add the contents of one bag to the existing contents of another bag

- 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;
```

This adds the contents of `second_bag` to what's already in `first_bag`

- `first_bag` contains one 4 and two 8s

The Bag Class

The Bag Class Specification — The bag's CAPACITY

- We want to implement a **bounded bag** that can hold 30 items
- The best way to define `CAPACITY` is as a **static member constant**

❑Example:

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

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

The Bag Class

The Bag Class Specification — Clarifying the `const` Keyword

- A **static member constant** has the two keywords **`static`** and **`const`** before its declaration in a class

- The **`static`** keyword specifies that **one copy of the member is shared by all instances of the class**
- The keyword **`const`** indicates that **a program cannot change the value**

- Note: The constant must be **re-declared in the implementation file without the keyword `static`**

```
|| inclusion of header files

namespace scu_coen70_3
{
    const bag::size_type bag::CAPACITY;

    || implementation of functions
```

The Bag Class

The Bag Class Specification — The bag's CAPACITY (Cont'd)

```
class bag {

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

- Referring to the CAPACITY variable:

❑Example:

```
bag b;
cout << "The capacity of b is " <<  b.CAPACITY << endl;
cout << "Every bag has capacity " <<  bag::CAPACITY << endl;
```

The Bag Class Specification — Clarifying the `const` Keyword

For **integer types**, the initial value (such as 30), is given only in the header file, not the implementation file

The value of the non-integer types must be defined in the implementation file

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

The Bag Class

The Bag Class Specification — Clarifying the `static` Keyword

```
#include <iostream>

using namespace std;

class CMyClass {
public:
    static int m_i;
};

int CMyClass::m_i = 0;
CMyClass myObject1;
CMyClass myObject2;

int main() {
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;

    myObject1.m_i = 1;
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;

    myObject2.m_i = 2;
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;
}
```

```
Output:
0
0
1
1
2
2
```

The Bag Class

The Bag Class Documentation

- The documentation includes information about the two `typedef` statements (`value_type` and `size_type`) and the static member constant (`CAPACITY`)
- In particular, notice that we have been very specific about what sort of data type is required for the `value_type`

```
// …
// TYPEDEF and MEMBER CONSTANTS for the bag class:
// typedef ____ value_type
// bag::value_type is the data type of the items in the bag. It may be
// any of the C++ built-in types (int, char, etc.), or a class with a
// default constructor, an assignment operator, and operators to test
// for equality (x == y) and non-equality (x != y).
// …
```

The Bag Class

Documenting the Value Semantics

- One of the requirements for the `value_type` may seem peculiar
  - **Why do we require that `value_type` "must have an assignment operator"?**
  - Doesn't every data type permit assignments such as `x = y`?
  - **Won't there always be an automatic assignment operator? No!**
  - ❑For example, `x = y` is forbidden when x and y are arrays (see below)
- Later we will see other data types that require care in defining what the assignment operator actually means

```
int main()
{
    int a[3];
    int b[3] = {0, 1, 2};
    a = b;
}
```

```
Output:
In function 'int main()':
4:6: error: invalid array assignment
```

The Bag Class

Documentation for the Bag Header File (Cont'd)

```
// FILE: bag1.h
// CLASS PROVIDED: bag (part of the namespace scu_coen79_3)
//
// TYPEDEF and MEMBER CONSTANTS for the bag class:
// typedef ____ value_type
// bag::value_type is the data type of the items in the bag. It may be
// any of the C++ built-in types (int, char, etc.), or a class with a
// default constructor, an assignment operator, and operators to test
// for equality (x == y) and non-equality (x != y).
//
// typedef ____ size_type
// bag::size_type is the data type of any variable that keeps track of
// how many items are in a bag.
//
// static const size_type CAPACITY = _____
// bag::CAPACITY is the maximum number of items that a bag can hold.
```

The Bag Class

Documentation for the Bag Header File (Cont'd)

```
// CONSTRUCTOR for the bag class:
// bag( )
// Postcondition: The bag has been initialized as an empty bag.
//
// MODIFICATION MEMBER FUNCTIONS for the bag class:
// size_type erase(const value_type& target);
// Postcondition: All copies of target have been removed from the bag.
// The return value is the number of copies removed (which could be
// zero).
//
// void erase_one(const value_type& target)
// Postcondition: If target was in the bag, then one copy has been
// removed; otherwise the bag is unchanged. A true return value
// indicates that one copy was removed; false indicates that nothing
// was removed.
//
// void insert(const value_type& entry)
// Precondition:  size( ) < CAPACITY.
// Postcondition: A new copy of entry has been added to the bag.
```

The Bag Class

Documentation for the Bag Header File (Cont'd)

```
// void operator +=(const bag& addend)
// Precondition:  size( ) + addend.size( ) <= CAPACITY.
// Postcondition: Each item in addend has been added to this bag.
//
// CONSTANT MEMBER FUNCTIONS for the bag class:
// size_type size( ) const
// Postcondition: The return value is the total number of items in the
// bag.
//
// size_type count(const value_type& target) const
// Postcondition: The return value is number of times target is in the
// bag.
//
// NONMEMBER FUNCTIONS 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.
//
// VALUE SEMANTICS for the bag class:
// Assignments and the copy constructor may be used with bag objects.
```

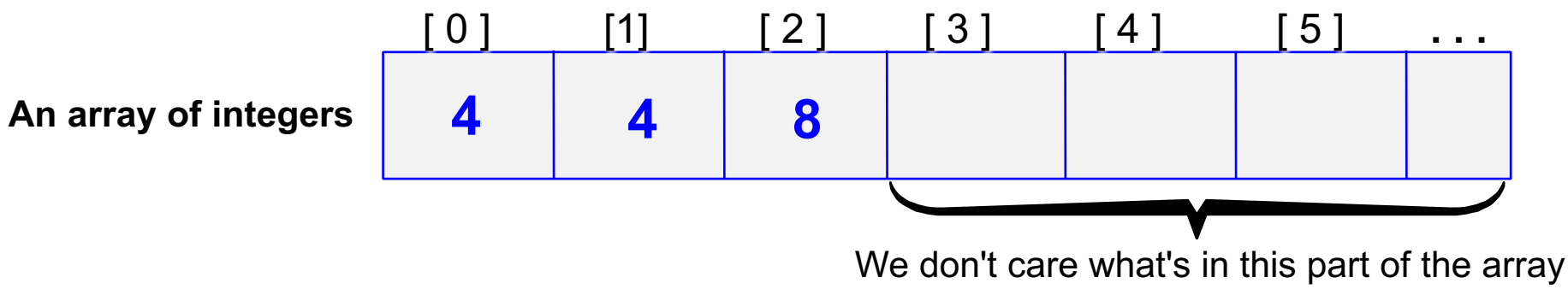
The Bag Class

Please refer to Appendix 1 to see a demonstration program for the Bag class

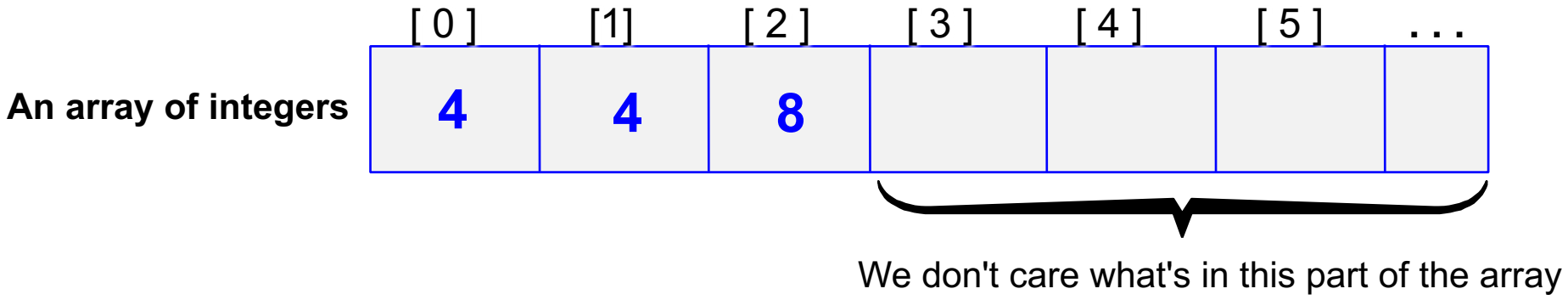
The Bag Class Design

- There are several ways to design the bag class
- For now: we design a somewhat inefficient data structure
- The primary structure for our design: An **array** that stores the items of a bag
- We use the **beginning part** of a large array to store entries of a bag
  - Such an array is called a **partially filled array**

❑Example:



The Bag Class Design (Cont'd)



- This **array will be one of the private member variables** of the bag class
- The **length of the array** will be determined by the constant `CAPACITY`
- Because part of the array can contain garbage, 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`**

An integer to keep track of the bag's size

The Bag Class Design (Cont'd)

```
class bag
{

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

    bag( ) { used = 0; }

    || 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
};
```

Two private member variables for the bag

The Bag Class

The `value_type` must have a default constructor

- The `value_type` is used as the component type of an array in the private member variable:

```
class bag {
    ...
    private:
    value_type  data[CAPACITY]; // An array to store items
    ...
```

- If the `value_type` is a class with constructors (rather than one of the C++ built-in types), then the compiler must initialize each component of the data array using the item's **default constructor**
- This is why our bag documentation includes the statement that "the `value_type` type must be "a class with a default constructor . . ."
- When an array has a component type that is a class, **the compiler uses the default constructor** to initialize the array components

The Invariant of a Class

- **We need to state how the member variables of the bag class are used to represent a bag of items**
- There are 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 do not care what is stored in any of `data`; for a non-empty bag, the items in the bag are stored in `data[0]` through `data[used-1]`, and we don't care what is stored in the rest of data
- **The rules that dictate how the member variables of a class represent a value** (such as a bag of items) are called the **invariant of the class**
- With the exception of the constructors, **each function depends on the invariant being valid when the function is called**

The Invariant of a Class (Cont'd)

- And each function, including the constructors, has a responsibility of ensuring that the invariant is valid when the function finishes
- 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 **is not usually written as an explicit part of the preconditions and postconditions** because the programmer who uses the class does not need to know about these conditions
- The invariant is a critical part of the implementation of a class, but it has no effect on the way the class is used

The Bag Class Implementation — The value semantics

- Our documentation indicates that **assignments and the copy constructor may be used with a bag**
- Our plan is to use the **automatic assignment operator** and the **automatic copy constructor**, each of which simply copies the member variables from one bag to another
- This is fine because **the copying process will copy both the data array and the member variable `used`**

❑Example: If a programmer has two bags `x` and `y`, then the statement `y = x` will invoke the automatic assignment operator to copy all of `x.data` to `y.data`, and to copy `x.used` to `y.used`

- Our only "work" for the value semantics is confirming that the automatic operations are correct

The Bag Class Implementation — The count member function

- To count **the number of occurrences of a particular item** in a bag, we step through the used portion of the partially filled array
- Remember that we are using locations `data[0]` through `data[used-1]`, so the correct loop is:

```
bag::size_type  bag::count(const value_type& target) const
{
    size_type  answer;
    size_type  i;

    answer = 0;
    for (i = 0; i < used; ++i)
        if (target == data[i])
            ++answer;
    return answer;
}
```

Iteration through the used portion of a partially filled array

The Bag Class Implementation — Needing to use the full type name

- When we implement the `count` function, we must take care to write the return type:

```
bag::size_type bag::count(const value_type& target) const
```

- We have used the completely specified type `bag::size_type` rather than just `size_type`
  - **Because many compilers do not recognize that you are implementing a bag member function until after seeing `bag::count`**
- In the implementation, after `bag::count`, we may use simpler names such as `size_type` and `value_type`
- However, before `bag::count`, we should use the full type name `bag::size_type`

The Bag Class Implementation — The `insert` member function

- The `insert` function checks that there is room to insert a new item
- The next available location is `data[used]`

❑Example: If `used=3`, then `data[0]`, `data[1]`, and `data[2]` are already occupied, and the next location is `data[3]`

```
void bag::insert(const value_type& entry)
   // Library facilities used: cassert
   {
       assert(size( ) < CAPACITY);

       data[used] = entry;
       ++used;
   }
```

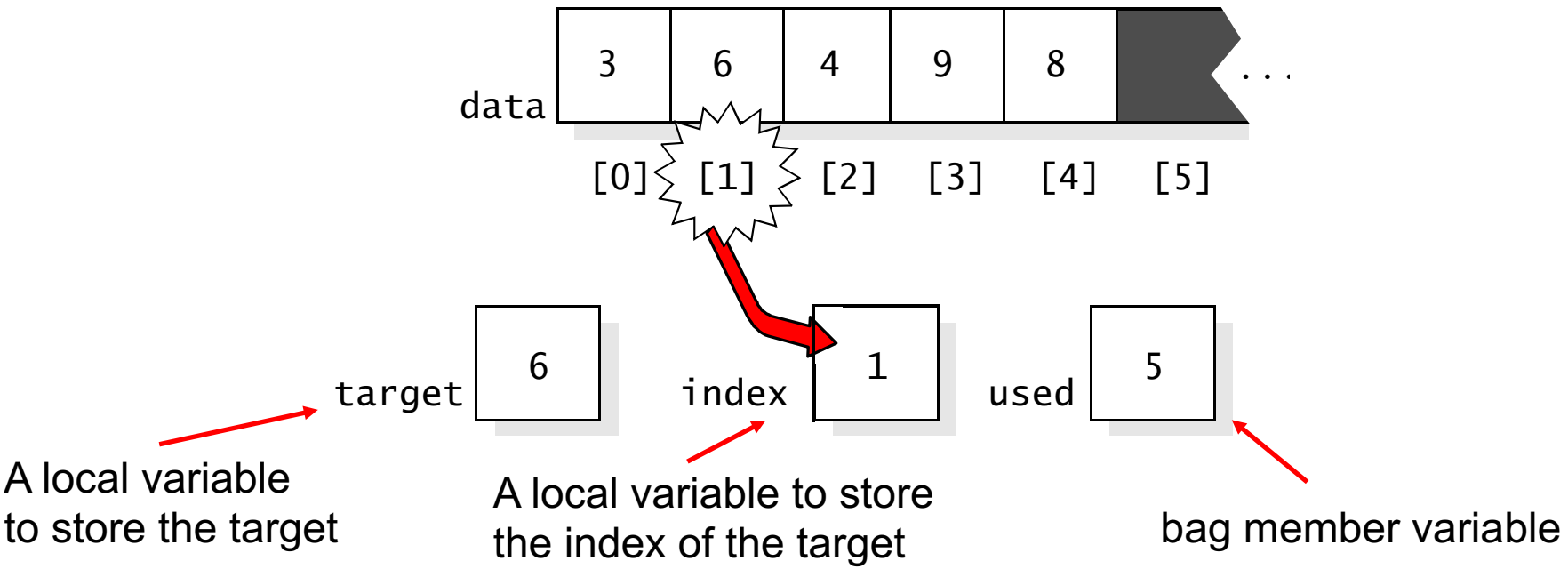
Can be replaced by:
`data[used++] = entry`

Note: Within a member function we can refer to the static member constant `CAPACITY` with no extra notation

The Bag Class Implementation — The `erase_one` member function

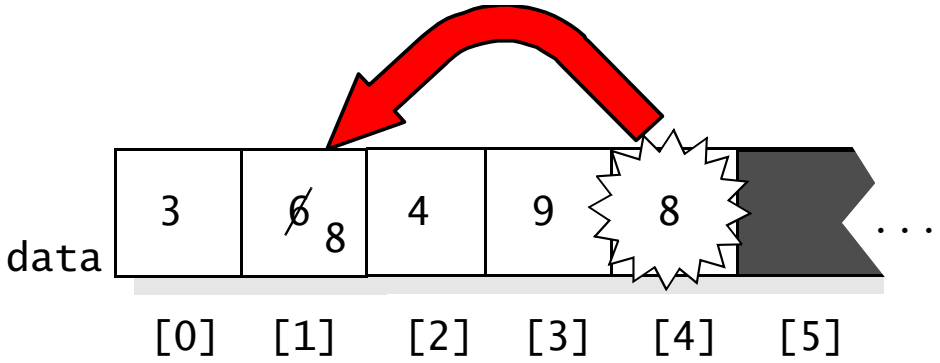
- How the `erase_one` function removes an item named `target` from a bag?

1. We find the index of `target` in the bag's array, and store this index in a local variable named `index`
   - ❑Example: Suppose that target is the number 6 in the five-item bag

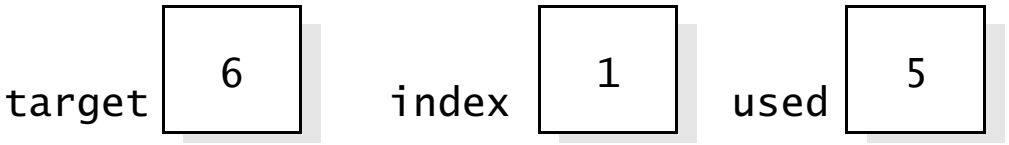


The Bag Class Implementation — The `erase_one` member function (Cont'd)

2. Take the final item in the bag and copy it to `data[index]`



The final item is copied onto the item that we are removing

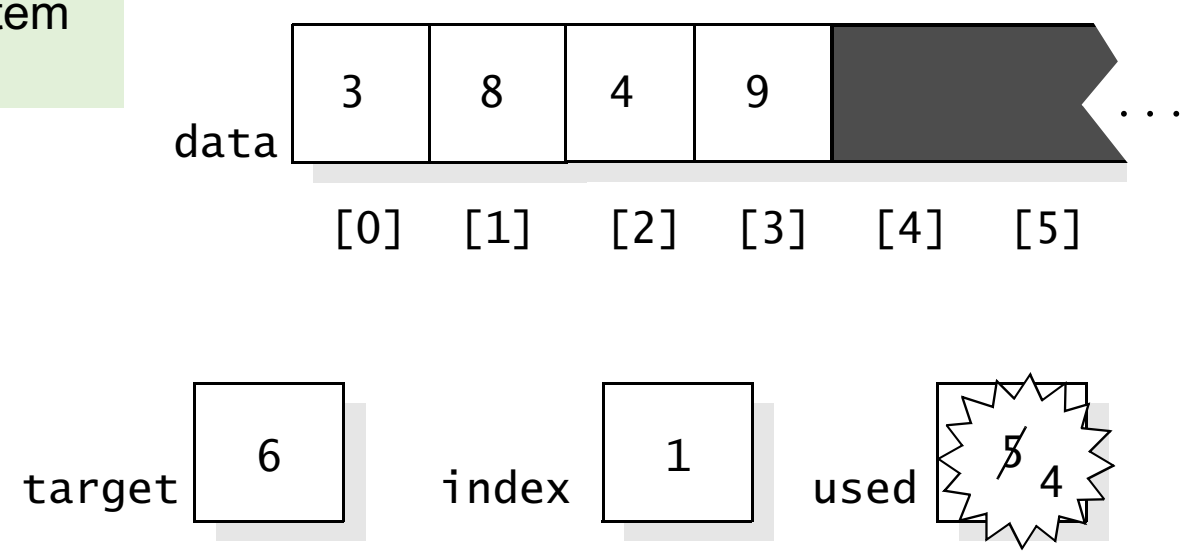


The reason for this copying is so that all the bag's items stay together at the front of the partially filled array, with no holes

The Bag Class Implementation — The `erase_one` member function (Cont'd)

3. Reduce the value of `used` by one — in effect reducing the used part of the array by one

The value of `used` is reduced by one to indicate that one item has been removed



The Bag Class

Implementation of the Member Function to Remove an Item

```
bool bag::erase_one(const value_type& target)
    {
        size_type index;

        index = 0;
        while ( (index < used) && (data[index] != target) )
            ++index;

        if (index == used)
            return false;

        --used;
        data[index] = data[used];
        return true;
    }
```

Set index to the location of target in the data array, which could be as small as 0 or as large as used−1
If target is not in the array, then index will be set equal to used

Target is not in the bag: No work to do

Target is in the bag

The Bag Class

Implementation of the Member Function to Remove an Item (Cont'd)

```
bool bag::erase_one(const value_type& target)
    {
        size_type index;
        index = 0;
        while ((index < used) && (data[index] != target))
            ++index;

        if (index == used)
            return false;
        --used;
        data[index] = data[used];
        return true;
    }
```

Test for (`index < used`) must appear before the other part of the test to ensure that only valid indexes are used

- C++ uses **short-circuit evaluation** to evaluate boolean expressions
- In short-circuit evaluation: A boolean expression is evaluated from left to right, **and the evaluation stops as soon as there is enough information to determine the value of the expression**

The Bag Class Implementation — The operator +=

- The implementation is as follows:

```
void bag::operator +=(const bag& addend)
{
    ...
    for (i = 0; i < number of items to copy; ++i)
    {
        data[used] = addend.data[i];
        ++used;
    }
}
```

- To avoid an explicit loop **we can use the `copy` function from the `<algorithm>` Standard Library**

The Bag Class

An Object Can Be An Argument To Its Own Member Function

☞**Pitfall**: The same variable is sometimes used on both sides of an assignment or other operator

- ❑Example:

```
bag b;
b.insert(5);
b.insert(2);
b += b;
```

b now contains a 5 and a 2

Now b contains two 5s and two 2s

Takes all the items in b (the 5 and the 2) and adds them to what's already in b, so b ends up with two copies of each number

- In the += statement, the bag b is activating the `+= operator`, but this same bag `b` is the actual argument to the operator
- This is a situation that must be carefully tested

The Bag Class

An Object Can Be An Argument To Its Own Member Function (Cont'd)

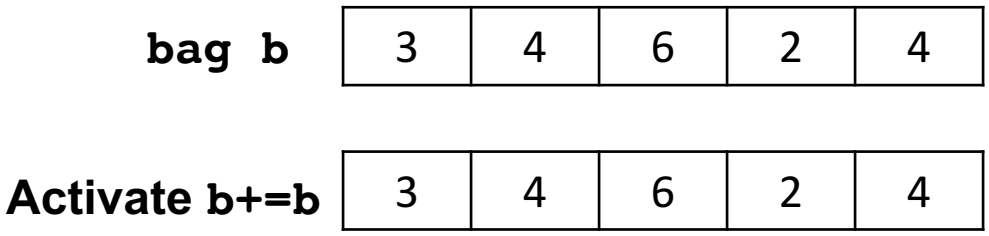
❑**Example of the danger**: Consider the **incorrect** implementation of +=

```
void bag::operator +=(const bag& addend)
{
    size_type i;     // An array index
    assert(size( ) + addend.size( ) <= CAPACITY);
    for (i = 0; i < addend.used; ++i)
    {
        data[used] = addend.data[i];
        ++used;
    }
}
```

- If we activate `b+=b` then the private member variable used is the same variable as `addend.used`
- **Each iteration of the loop adds 1 to `used`, and hence `addend.used` is also increasing, and the loop never ends**
- **What is the solution?**

The Bag Class

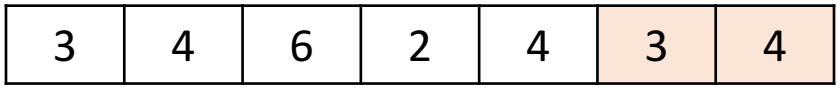
An Object Can Be An Argument To Its Own Member Function (Cont'd)




```
data[used] = addend.data[i];
++used;
```

```
data[5] = addend.data[0];
++5;
```


```
data[6] = addend.data[1];
++6;
```



The Copy Function From The C++ Standard Library

- The Standard Library contains a **copy function** for easy copying of items from one location to another
- The function is part of the `std` namespace in the `<algorithm>` facility:

```
copy(<beginning location>, <ending location>, <destination>);
```

- It continues beyond the beginning location, copying more and more items to the next spot of the destination, until we are about to copy the ending location - **The ending location is not copied**

❑Example: Suppose that `b` and `c` are arrays

To copy the items `b[0]...b[9]` into locations `c[40]...c[49]`, we could write:

```
copy(b, b + 10, c + 40);
```

- Note: `b[10]` is not copied

The Bag Class Implementation — The operator += (Cont'd)

- This implementation uses the `copy` function from the `<algorithm>` Standard Library

```
void bag::operator +=(const bag& addend)
{
    assert(size( ) + addend.size( ) <= CAPACITY);

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

    used += addend.used;
}
```

Beginning location

Ending location

Destination

The Bag Class

The Bag Class Implementation — The operator +

- The operator+ is **an ordinary function** rather than a member function
- The function must take two bags, add them together into a third bag, and return this **third bag**

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

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

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

Note: We need to use the scope resolution operator because operator+ is not a member function

- Does this function need to be a friend function of the bag class?

The Bag Class Implementation — The `erase` member function

- The `erase` function **removes all copies of target** from the bag and returns the number of copies removed

```
bag::size_type bag::erase(const value_type& target)
    {
        size_type index = 0;
        size_type many_removed = 0;

        while (index < used)
        {
            if (data[index] == target)
            {
                --used;
                data[index] = data[used];
                ++many_removed;
            }
            else
                ++index;
        }
        return many_removed;
    }
```

What if we want to erase 4, and there are three 4s in the bag?

Document Class Invariant in the Implementation File

- **The best place to document the class's invariant is at the top of the implementation file**
- In particular, do not write the invariant in the header file, **because a programmer who uses the class does not need to know about how the invariant dictates the use of private fields**
- But the programmer who implements the class does need to know about the invariant

The Bag Class

Header File for the Bag Class

```
#ifndef SCU_coen79_BAG1_H
#define SCU_coen79_BAG1_H
#include <cstdlib>  // Provides size_t

namespace scu_coen79_3
{
    class bag
    {
    public:
        // TYPEDEFS and MEMBER CONSTANTS
        typedef int value_type;
        typedef std::size_t size_type;
        static const size_type CAPACITY = 30;

        // CONSTRUCTOR
        bag( ) { used = 0; }

        // MODIFICATION MEMBER FUNCTIONS
        size_type erase(const value_type& target);
        bool erase_one(const value_type& target);
        void insert(const value_type& entry);
        void operator +=(const bag& addend);
```

The Bag Class

Header File for the Bag Class (Cont'd)

```
        // CONSTANT MEMBER FUNCTIONS
        size_type size( ) const { return used; }
        size_type count(const value_type& target) const;

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

    // NONMEMBER FUNCTIONS for the bag class
    bag operator +(const bag& b1, const bag& b2);
}

#endif
```

The Bag Class — Analysis

- We'll use the number of items in a bag as the input size for time analysis

- To count the operations, we'll count the number of statements executed by the function, although we won't need an exact count since our answer will use *big-O* notation
- All of the work in `count()` happens in this loop:

```
for (i = 0; i < used; ++i)
    if (target == data[i])
        ++answer;
```

- The body of the loop will be executed exactly *n* times
- The time expression is always *O(n)*

Time Analysis for the Bag Functions

Operation	Time Analysis	
Default constructor	$O(1)$	Constant time
count	$O(n)$	n is the size of the bag
erase_one	$O(n)$	Linear time
erase	$O(n)$	Linear time

- `erase_one` sometimes requires fewer than n × (number of statements in the loop); however, this does not change the fact that the function is $O(n)$
- In the worst case, the loop does execute a full n iterations, therefore the correct time analysis is no better than O(n)

Time Analysis for the Bag Functions (Cont'd)

Operation	Time Analysis	
+= another bag	$O(n)$	n is the size of the other bag
b1 + b2	$O(n_1 + n_2)$	n_1 and n_2 are the sizes of the bags
insert	$O(1)$	Constant time
size	$O(1)$	Constant time

- Several of the other bag functions do not contain any loops at all, and do not call any functions with loops
  - Example, when an item is added to a bag, the new item is always placed at the end of the array

The Sequence Class
(Lab Project 3)

Programming Project: The Sequence Class

Introduction

- 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

Programming Project: The Sequence Class

The Sequence Class — Specification

- Our sequence is a class that depends on an underlying `value_type`, and the class also provides a `size_type`
- At the moment, a sequence will be limited to no more than 30 items

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

- **The capacity and item type can easily be changed and recompiled if we need other kinds of sequences**

Programming Project: The Sequence Class

Member functions to examine a sequence

- With the bag class, all that we can do is inquire how many copies of a particular item are in the bag
- A sequence is more flexible, allowing us to examine the items one after another

- Three member functions work together to enforce the **in-order retrieval rule**

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

- After activating `start`, the `current` function returns the first item
- Each time we call `advance`, the `current` function changes so that it returns the next item in the sequence

Programming Project: The Sequence Class

Member functions to examine a sequence (Cont'd)

- Example

If a sequence named `numbers` contains the four numbers 37, 10, 83, and 42, then we can write:

```
numbers.start( );
cout << numbers.current( ) << endl;          //Prints 37

numbers.advance( );
cout << numbers.current( ) << endl;          //Prints 10

numbers.advance( );
cout << numbers.current( ) << endl;          //Prints 83
```

Programming Project: The Sequence Class

Member functions to examine a sequence (Cont'd)

- Member function `is_item` cooperates with `current`
- `is_item` returns a boolean value to indicate **whether there actually is another item for `current` to provide**, or **whether `current` has advanced right off the end**

```
bool is_item( ) const;
// Postcondition: A true return value indicates that there is a valid
// "current" item that can be obtained from the current member
// function.
// A false return value indicates that there is no valid current item.
```

- Using all four of the member functions in a for-loop, we can print an entire sequence:

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

Programming Project: The Sequence Class

The `insert` and `attach` member functions

- There are two member functions **to add new items to a sequence**
  - `Insert`: Places a new item **before** the current item
  - `Attach`: Adds a new item to a sequence **after** the current item

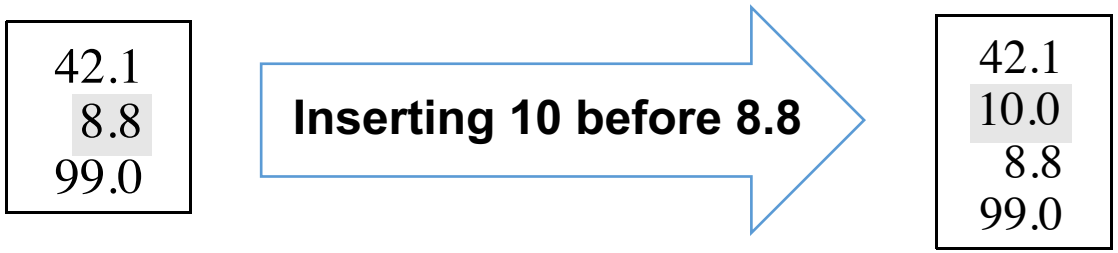
Programming Project: The Sequence Class

The `insert` and `attach` member functions (Cont'd)

- **`Insert`**

```
void insert(const value_type& entry);
// Precondition: size( ) < CAPACITY.
// Postcondition: A new copy of entry has been inserted in the sequence
// before the current item. If there was no current item, then the new entry
// has been inserted at the front. In either case, the new item is now the
// current item of the sequence.
```

❑Example:



- 8.8 and 99.0 are **moved to make room** for the new item

Programming Project: The Sequence Class

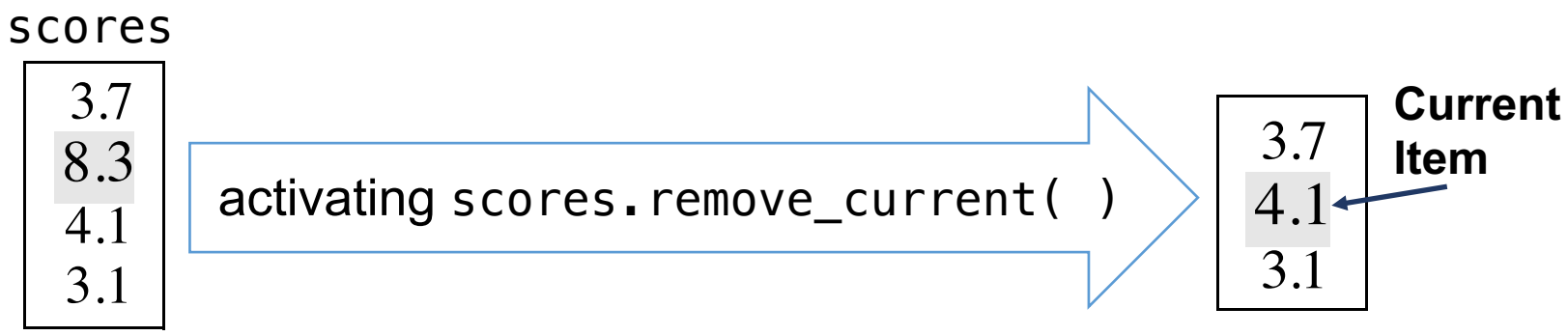
The `remove_current` member function (Cont'd)

- The current item can be removed from a sequence
- The member function for a removal has no parameters

```
void remove_current( );
// Precondition: is_item returns true.
// Postcondition: The current item has been removed from the sequence,
// and the item after this (if there is one) is now the new current item.
```

- The function's precondition requires that there is a current item

❑Example

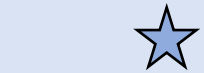


The Sequence Class — Design

The sequence class has three private member variables:

- **`data`**: Is an **array** (partially filled) that stores the items of the sequence
- **`used`**: Keeps track of how much of the data array is currently being used, extends from `data[0]` to `data[used-1]`
- **`current_index`**: Gives the index of the "current" item in the array (if there is one); If there is no valid current item in the sequence, then `current_index` will be the same number as `used` (since this is larger than any valid index)

The Sequence Class — Design



Complete **invariant of our class**, stated as three rules:

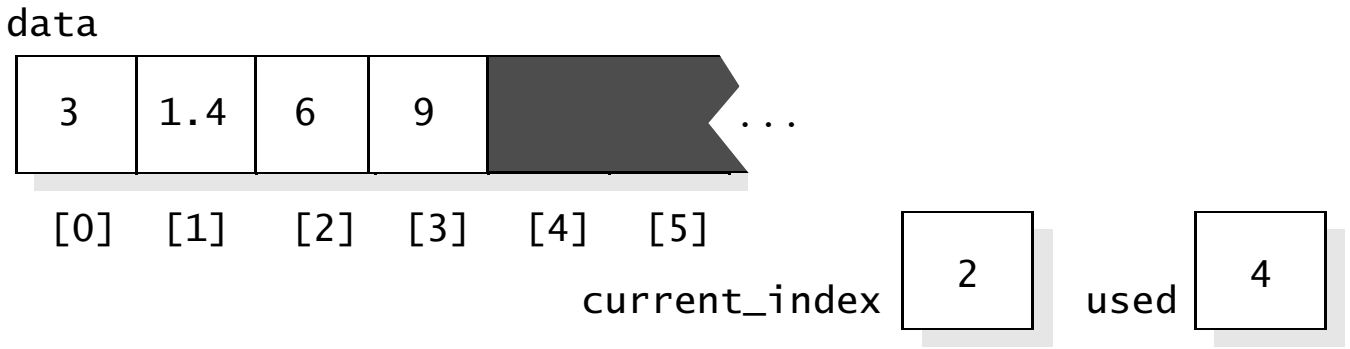
- The number of items in the sequence is stored in the member variable **`used`**
- For an empty sequence, we do not care what is stored in any of `data`; for a non-empty sequence, the items are stored in their sequence order from `data[0]` to `data[used-1]`, and we don't care what is stored in the rest of `data`
- If there is a current item, then it lies in `data[current_index]`; if there is no current item, then `current_index` equals `used`

The Sequence Class — Design (Cont'd)

❑Example

Suppose that a sequence contains four numbers, with the current item at `data[2]`

- The member variables of the object might appear as



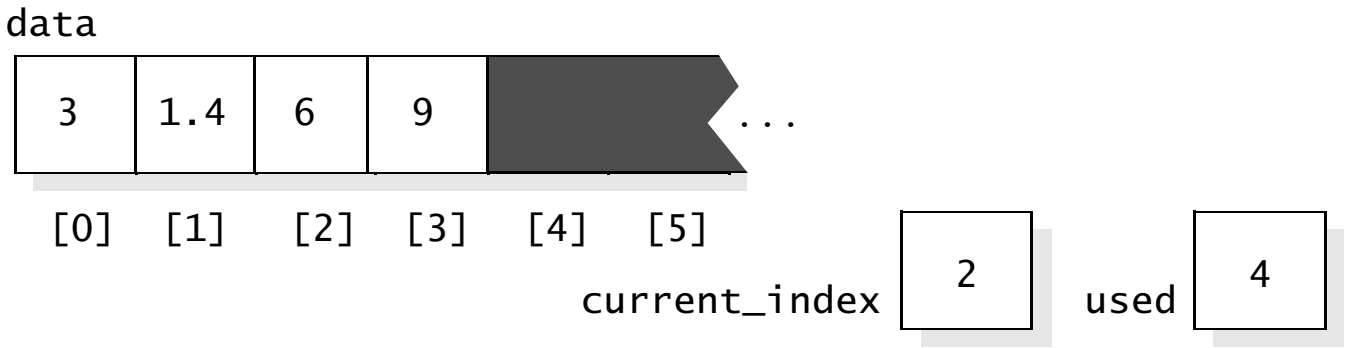
- The current item is at `data[2]`, so the `current()` function would return the number 6

The Sequence Class — Design (Cont'd)

❑Example (Cont'd)

If we called `advance()`, then `current_index` would increase to 3, and `current()` would then return 9

- Normally, a sequence has a "current" item, and the member variable `current_index` contains the location of that current item
- **If there is no current item, then `current_index` contains the same value as `used`**



Programming Project: The Sequence Class

Header File for the Sequence Class

```
#ifndef SCU_coen79_SEQUENCE_H
#define SCU_coen79_SEQUENCE_H
#include <cstdlib>  // Provides size_t

namespace scu_coen79_3 {
    class sequence
    {
    public:
        // TYPEDEFS and MEMBER CONSTANTS
        typedef double value_type;
        typedef std::size_t size_type;
        static const size_type CAPACITY = 30;

        // CONSTRUCTOR
        sequence( );

        // MODIFICATION MEMBER FUNCTIONS
        void start( );
        void advance( );
        void insert(const value_type& entry);
        void attach(const value_type& entry);
        void remove_current( );
```

Programming Project: The Sequence Class

Header File for the Sequence Class (Cont'd)

```
        // CONSTANT MEMBER FUNCTIONS
        size_type size( ) const;
        bool is_item( ) const;
        value_type current( ) const;

    private:
        value_type data[CAPACITY];
        size_type used;
        size_type current_index;

    };
}

#endif
```

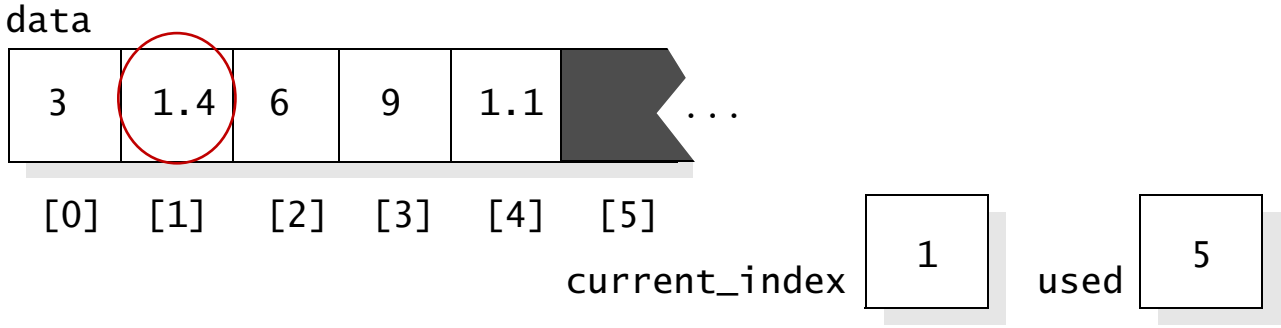
Programming Project: The Sequence Class

Pseudocode for the Implementation — The `remove_current` function

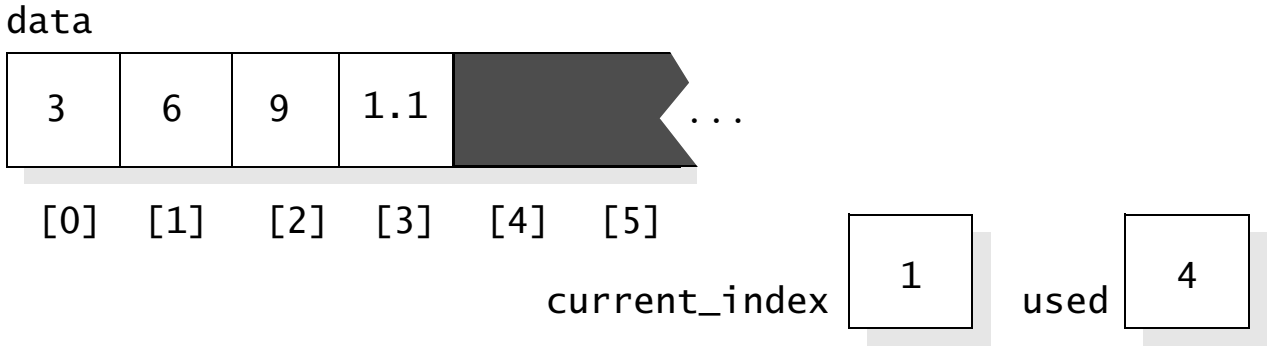
The `remove_current` function removes the current item from the sequence:

1. Checks that the precondition is valid (use `is_item()` in an assertion)
2. **Removes the current item by shifting each of the subsequent items leftward one position**

❑Example:



1. After removing the current item (1.4):



Programming Project: The Sequence Class

Pseudocode for the Implementation — `remove_current` function (Cont'd)

- Pseudocode for moving items leftward one position:

```
for (i = the index after the current item;  i < used;  ++i)
     move an item from data[i] back to data[i-1];

used = used-1;
```

- **You should not use the `copy` function from `<algorithm>`** since that function forbids the overlap of the source with the destination
- You should check that the function works correctly for **boundary values**—removing the **first item** and removing the **final item**
- Both these cases do work fine:
  - When the final item is removed, `current_index` will end up with the same value as `used`, indicating that there is no longer a current item

Programming Project: The Sequence Class

Pseudocode for the Implementation — The `insert` function

- If there is a current item, then the insert function must take care to insert the new item **just before** the current position
- Items that are already at or after the current position must be shifted rightward to make room for the new item
- You should start by checking the precondition:

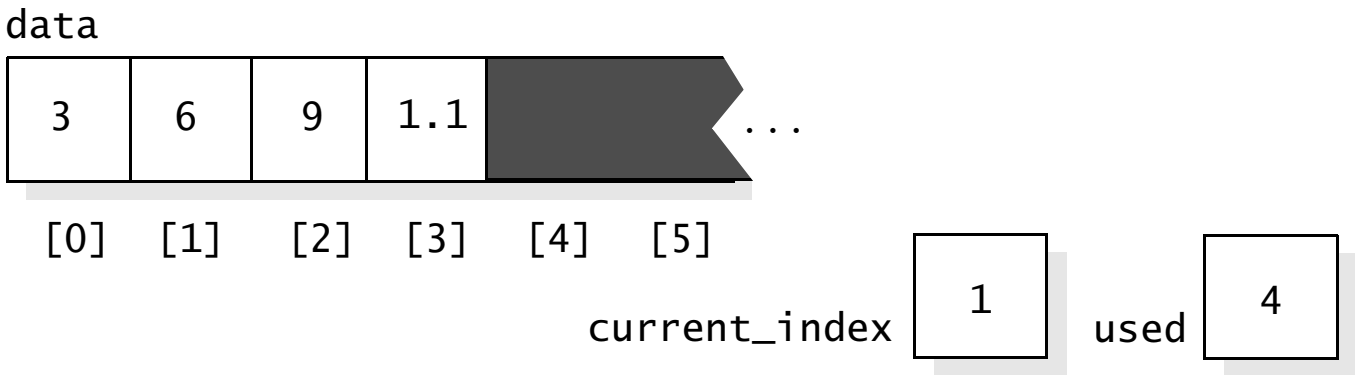
```
size( ) < CAPACITY
```

- Then shift items at the end of the array rightward one position each until you reach the position for the new item

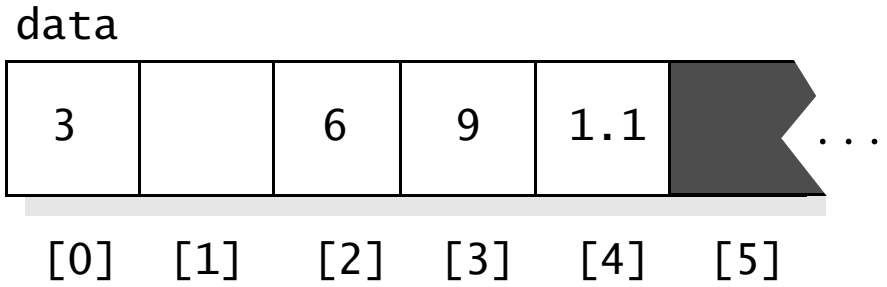
Programming Project: The Sequence Class

Pseudocode for the Implementation — The `insert` function (Cont'd)

- Example: Suppose you are inserting 1.4 at the location `data[1]`:



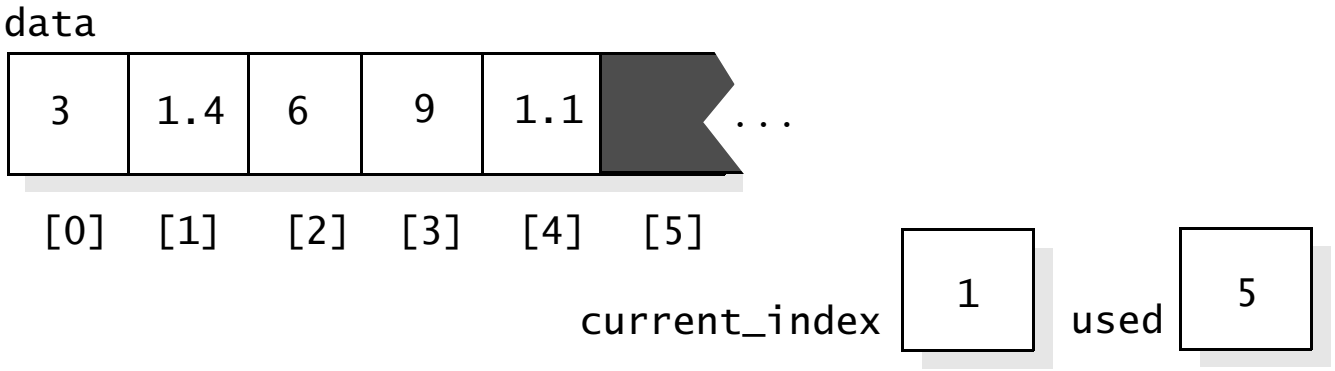
- Shift `data[3]` to `data[4]`; shift `data[2]` to `data[3]`; shift `data[1]` to `data[2]`. The array looks like:



Programming Project: The Sequence Class

Pseudocode for the Implementation—The insert function (Cont'd)

- We can place the 1.4 in `data[1]` and add one to `used`



- Pseudocode for shifting the items rightward:

```
for (i = used; data[i] is the wrong spot for entry; --i)
    data[i] = data[i-1];
```

- The key to the loop is the test `data[i]` is the wrong spot for entry; a position is wrong if `(i > current_index)`

Interactive Test Program for the Sequence Class

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

Interactive Test Programs

Interactive Test Program for the Sequence Class (Cont'd)

```
// FILE: sequence_test.cxx
// An interactive test program for the new sequence class
#include <cctype>       // Provides toupper
#include <iostream>     // Provides cout and cin
#include <cstdlib>      // Provides EXIT_SUCCESS
#include "sequence1.h"  // With value_type defined as double
using namespace std;
using namespace scu_coen79_3;

// PROTOTYPES for functions used by this test program:
void print_menu( );
// Postcondition: A menu of choices for this program has been written
// to cout.

char get_user_command( );
// Postcondition: The user has been prompted to enter a one character
// command.
// The next character has been read (skipping blanks and newline
// characters), and this character has been returned.
```

Interactive Test Program for the Sequence Class (Cont'd)

```
void show_sequence(sequence display);
// Postcondition: The items on display have been printed to cout (one
// per line).

double get_number( );
// Postcondition: The user has been prompted to enter a real number.
// The number has been read, echoed to the screen, and returned by the
// function.


int main( )
{
    sequence test;   // A sequence that we will perform tests on
    char choice;     // A command character entered by the user

    cout << "I have initialized an empty sequence of real numbers." <<
                                                            endl;
```

Interactive Test Programs

Interactive Test Program for the Sequence Class (Cont'd)

```
Do
{
    print_menu( )

    choice = toupper(get_user_command( ));

    switch (choice)
    {
        case '!': test.start( );
                  break;
         case '+': test.advance( );
                  break;
```

An interactive loop that continues as long as the user wants

A small menu of choices is written for the user Each choice is printed along with a letter or other meaningful character to allow the user to select the choice

Converting input to uppercase letters

The user's selection from the menu is read

Based on the user's selection, some action is taken on the sequence object

Interactive Test Programs

Interactive Test Program for the Sequence Class (Cont'd)

```
case '?': if (test.is_item( ))
            cout << "There is an item." << endl;
          else
            cout << "There is no current item." << endl;
          break;
case 'C': if (test.is_item( ))
            cout << "Current item is: " << test.current( )
                                                << endl;
          else
            cout << "There is no current item." << endl;
          break;
case 'P': show_sequence(test);
          break;
case 'S': cout << "Size is " << test.size( ) << '.' << endl;
          break;
case 'I': test.insert(get_number( ));
          break;
case 'A': test.attach(get_number( ));
          break;
```

Interactive Test Program for the Sequence Class (Cont'd)

```
            case 'R': test.remove_current( );
                      cout << "The current item has been removed." <<
                      endl;
                      break;

            case 'Q': cout << "Ridicule is the best test of truth." <<
                      endl;
                      break;

            default:  cout << choice << " is invalid." << endl;
        }
    }

    while ((choice != 'Q')); //end of the do-while loop

    return EXIT_SUCCESS;
}
```

Interactive Test Programs

Interactive Test Program for the Sequence Class (Cont'd)

A small menu of choices is written for the user

```
void print_menu( )
{
    cout << endl; // Print blank line before the menu
    cout << "The following choices are available: " << endl;
    cout << " !   Activate the start( ) function" << endl;
    cout << " +   Activate the advance( ) function" << endl;
    cout << " ?   Print the result from the is_item( ) function" <<
                                                          endl;
    cout << " C   Print the result from the current( ) function" <<
                                                          endl;
    cout << " P   Print a copy of the entire sequence" << endl;
    cout << " S   Print the result from the size( ) function" << endl;
    cout << " I   Insert a new number with the insert(...) function" <<
                                                          endl;
    cout << " A   Attach a new number with the attach(...) function" <<
                                                          endl;
    cout << " R   Activate the remove_current( ) function" << endl;
    cout << " Q   Quit this test program" << endl;
}
```

Interactive Test Program for the Sequence Class (Cont'd)

```
char get_user_command( )
{
    char command;

    cout << "Enter choice: ";
    cin >> command;   // Input of characters skips blanks and newline
                      // character

    return command;
}
```

```
void show_sequence(sequence display)
{
    for (display.start( ); display.is_item( ); display.advance( ))
        cout << display.current( ) << endl;
}
```

Interactive Test Program for the Sequence Class (Cont'd)

```
double get_number( )
{
    double result;

    cout << "Please enter a real number for the sequence: ";
    cin  >> result;
    cout << result << " has been read." << endl;

    return result;
}
```

Interactive Test Programs

A Sample Dialogue from the Program

```
I have initialized an empty sequence of real numbers.

The following choices are available:
 !  Activate the start( ) function
 +  Activate the advance( ) function
 ?  Print the result from the is_item( ) function
 C  Print the result from the current( ) function
 P  Print a copy of the entire sequence
 S  Print the result from the size( ) function
 I  Insert a new number with the insert(...) function
 A  Attach a new number with the attach(...) function
 R  Activate the remove_current( ) function
 Q  Quit this test program
Enter choice: A
Please enter a real number for the sequence: 3.14
3.14 has been read.
```

Interactive Test Programs

A Sample Dialogue from the Program (Cont'd)

```
The following choices are available:
 !  Activate the start( ) function
 +  Activate the advance( ) function
 ?  Print the result from the is_item( ) function
 C  Print the result from the current( ) function
 P  Print a copy of the entire sequence
 S  Print the result from the size( ) function
 I  Insert a new number with the insert(...) function
 A  Attach a new number with the attach(...) function
 R  Activate the remove_current( ) function
 Q  Quit this test program
Enter choice: S
Size is 1.
```

The dialogue continues until the user types **Q** to stop the program.

The Polynomial Class and a Project

Programming Project: The Polynomial

- A one-variable **polynomial** is an arithmetic expression of the form:

$$a_k x^k + \ldots + a_2 x^2 + a_1 x^1 + a_0 x^0$$

- The highest exponent, *k*, is called the **degree** of the polynomial, and the constants a_0, a_1, ... are the **coefficients**

❑ Example: Here is a polynomial with degree 3:

$$0.3x^3 + 0.5x^2 + (-0.9)x^1 + 1.0x^0$$

- Each individual **term** of a polynomial consists of a real number as a coefficient (such as 0.3), the variable *x*, and a non-negative integer as an **exponent**
- The x^1 term is usually written with just an *x* rather than x^1; the x^0 term is usually written with just the coefficient (since x^0 is always defined to be 1); and a negative coefficient may also be written with a subtraction sign:

$$0.3x^3 + 0.5x^2 - 0.9x + 1.0$$

Programming Project: The Polynomial

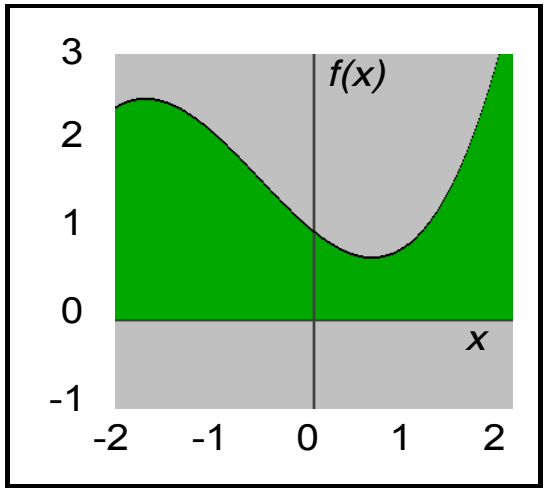
- For any specific value of *x*, a polynomial can be evaluated by plugging the value of *x* into the expression

❑ Example: The value of the sample polynomial at *x* = 2 is:

$$0.3(2)^3 + 0.5(2)^2 - 0.9(2) + 1.0 \;=\; 3.6$$

- A typical algebra exercise is to plot the graph of a polynomial for each value of *x* in a given range

❑Example



The graph of the function f(x) defined by the polynomial

$$0.3x^3 + 0.5x^2 - 0.9x + 1.0$$

Plots the value of a polynomial for each *x* in the range of –2 to +2

Programming Project: The Polynomial

- **For this project, you should specify, design, and implement a class for polynomials**
- The coefficients are **double** numbers, and the exponents are **non-negative integers**
- The coefficients should be stored in an array of double numbers, with the exponent for the x^k term stored in location [k] of the array
- The maximum index of the array needs to be at least as big as the degree of the polynomial, so that the largest nonzero coefficient can be stored
- The class may contain a static member constant, `MAXDEGREE`, which indicates the maximum degree of any polynomial
- Think about operations that make sense on polynomials

❑Example:

- An operation that adds two polynomials
- An operation should evaluate the polynomial for a given value of *x*

Summary

Summary

- A **container class** is a class where each object contains a collection of items
  - ❑ Examples: Bags and sequences classes
- `typedef` statement makes it easy to alter the data type of the underlying items
- The simplest implementations of container classes use a **partially filled array**, which requires each object to have at least two member variables:
  - The array
  - A variable to keep track of how much of the array is being used
- At the top of the implementation file: When you design a class, always make an explicit statement of the rules **(invariant of the class)** that dictate how the member variables are used

Copyright Notice

The following copyright notices apply to some of the materials in this presentation: