COEN 79

OBJECT-ORIENTED PROGRAMMING AND ADVANCED DATA STRUCTURES

DEPARTMENT OF COMPUTER ENGINEERING, SANTA CLARA UNIVERSITY

Container Classes

Implementations

Chapter 3

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

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);
```

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 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;
    }
```

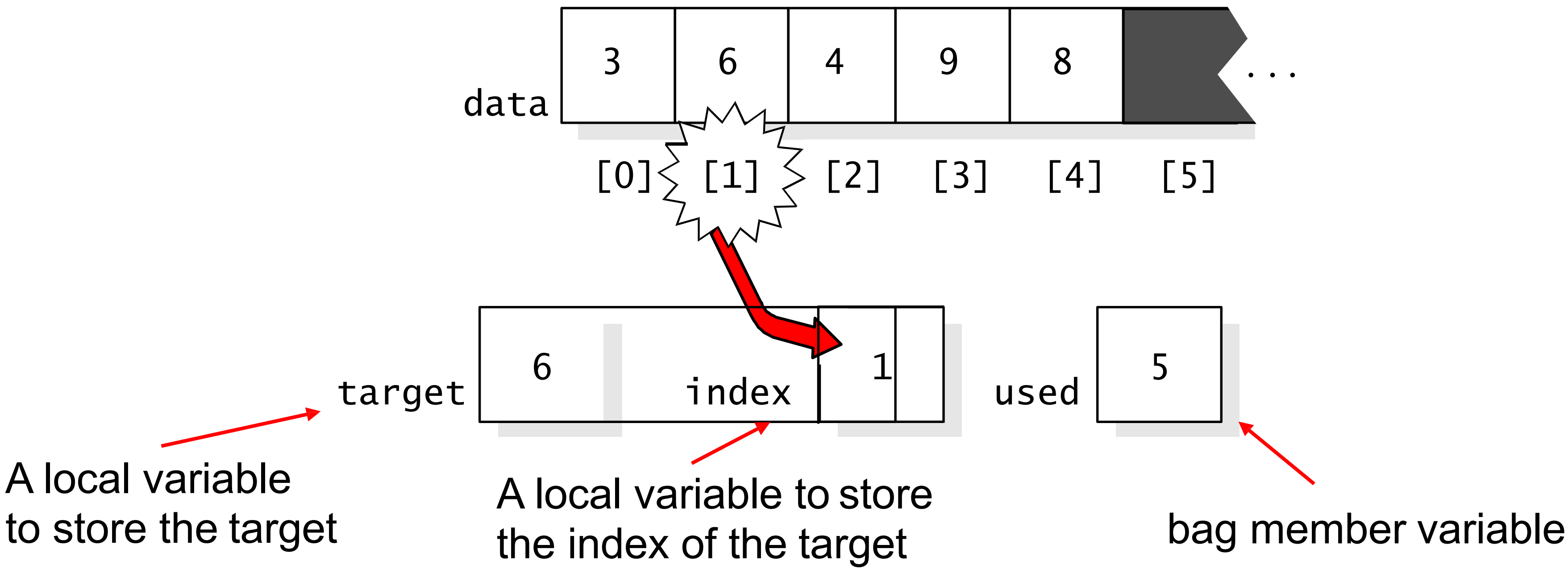
Can be replaced by:
`data[used++] = entry`

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

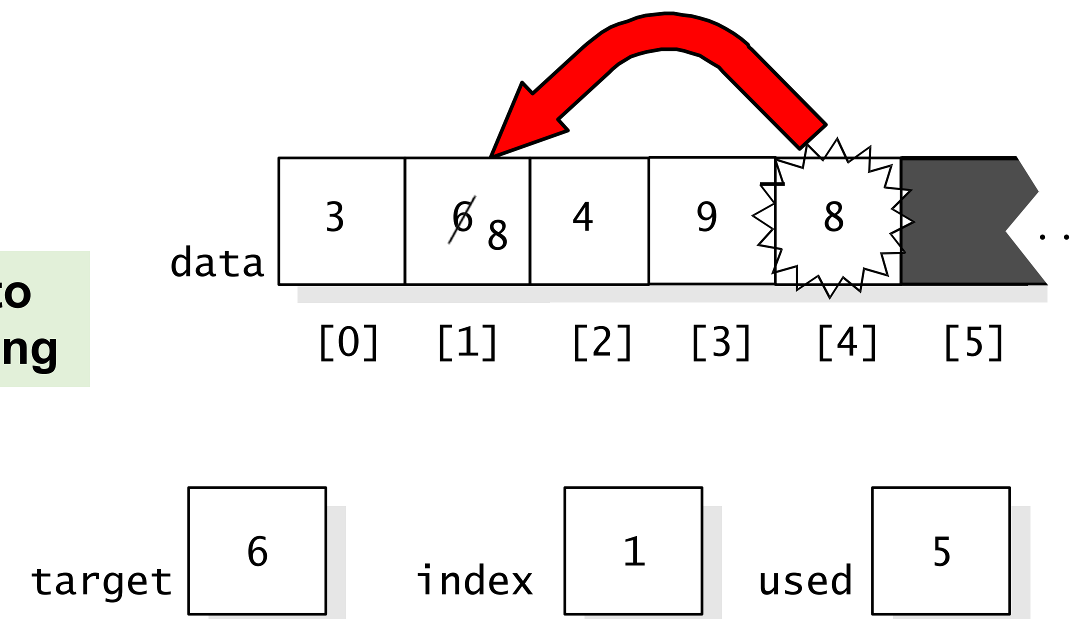
- How the `erase_one` function removes an item named `target` from a bag?

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]`



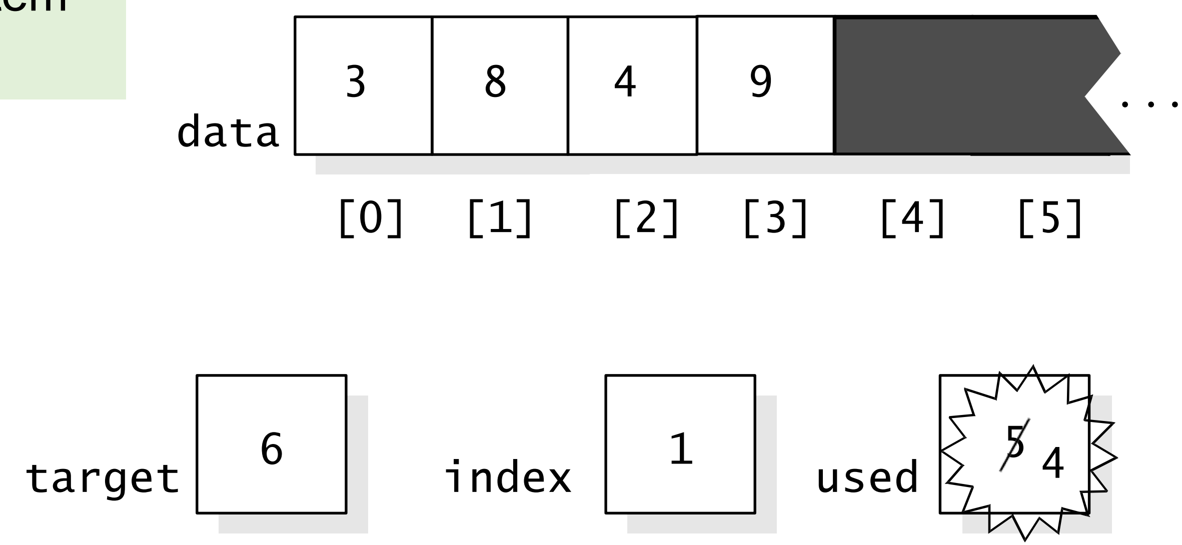
The final item is copied onto the item that we are removing

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**

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

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?**

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

`bag b`

3	4	6	2	4

Activate `b+=b`

3	4	6	2	4

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

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

3	4	6	2	4	3

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

3	4	6	2	4	3	4

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

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);

         Beginning location      Ending location              Destination
    copy(addend.data, addend.data + addend.used, data + used);

    used += addend.used;
}
```

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

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);
```

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 $n \times$ (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

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: