Container Classes

- A **container class** is a data type that is capable of holding a collection of 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.

Summary of the Bag Operations

1. A bag can be put in its **initial state**, which is an empty bag.
2. Numbers can be **inserted** into the bag.
3. You may check how many **occurrences** of a certain number are in the bag.
4. Numbers can be **removed** from the bag.
5. You can check **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**
  - **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.

```
};
```

Using the Bag in a Program

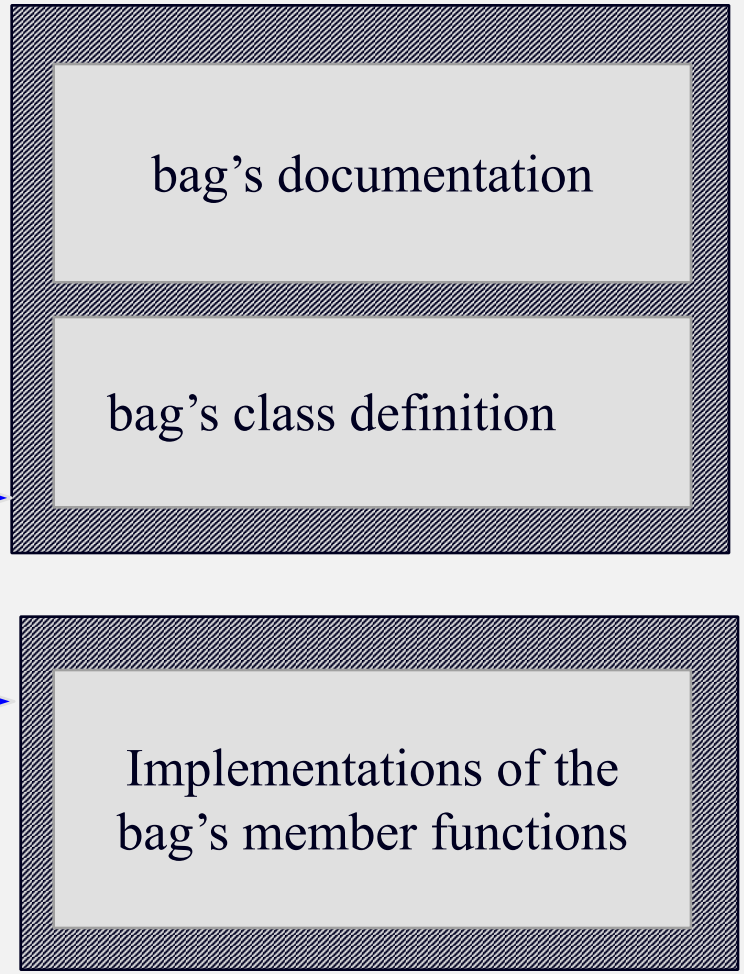
- Here is typical code from a program that uses the new bag class:

```
bag  ages;

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

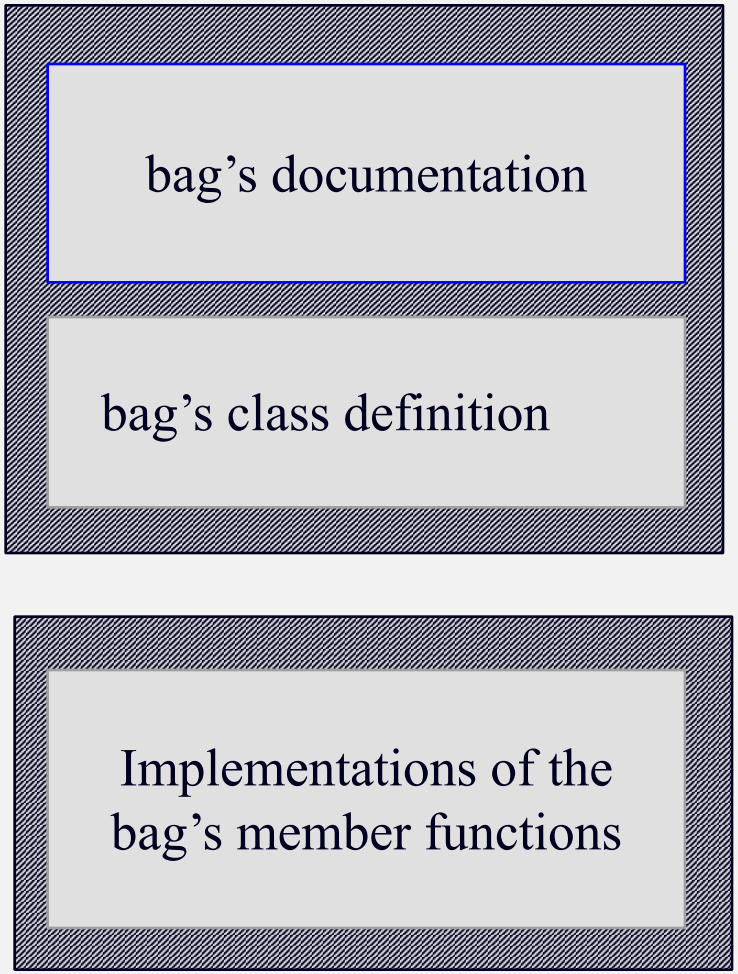
The Header File and Implementation File

- The programmer who writes the new bag class must write two files:
- **bag1.hxx**, 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

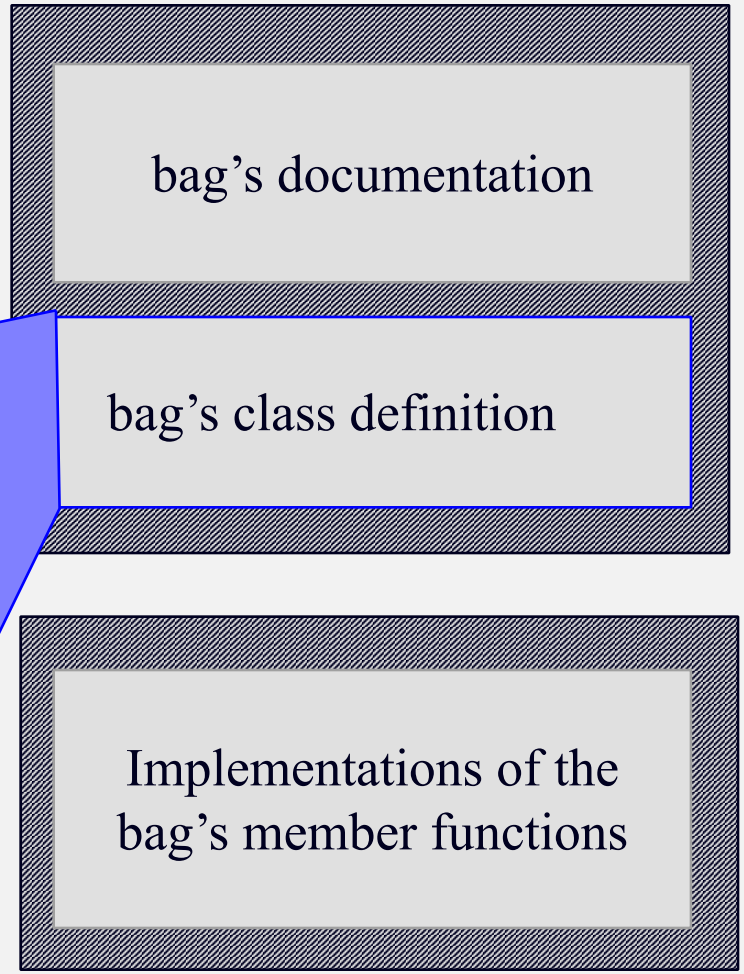
- The documentation gives **prototypes and specifications** for the bag member functions.
- Specifications are written as **precondition/postcondition** contracts.
- Everything needed to **use** the bag class is included in this comment.



The Bag's Class Definition

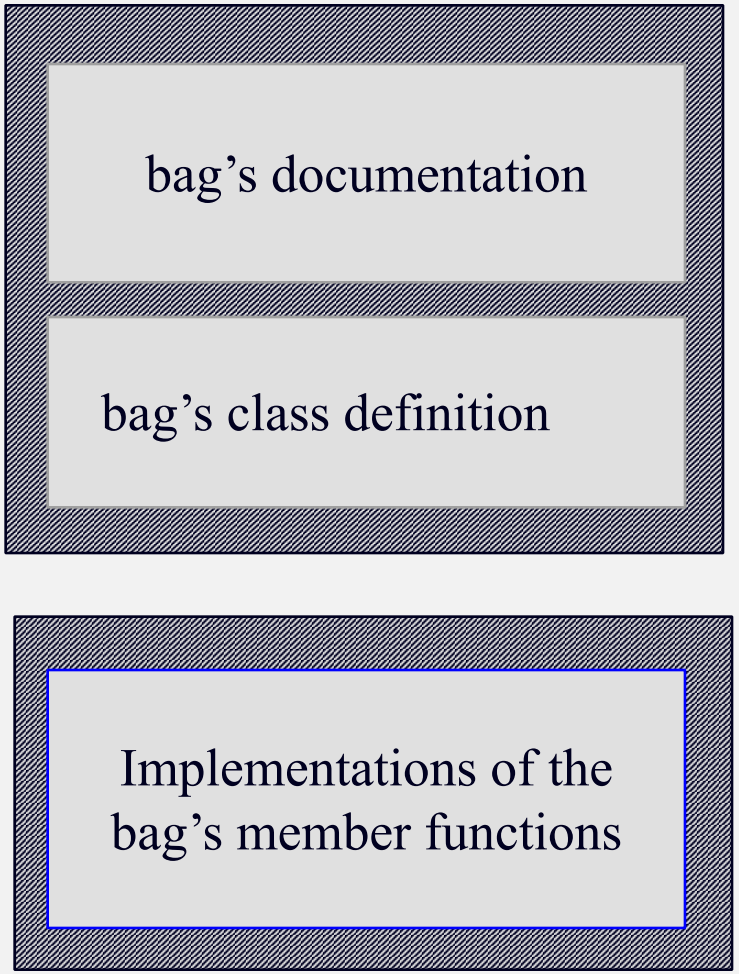
- After the documentation, the header file has the class definition that we've seen before:

```
class bag
{
public:
        bag( );
        void insert(...
        void remove(...
        ...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 **definitions** 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 declaration for the bag.

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

A Quiz

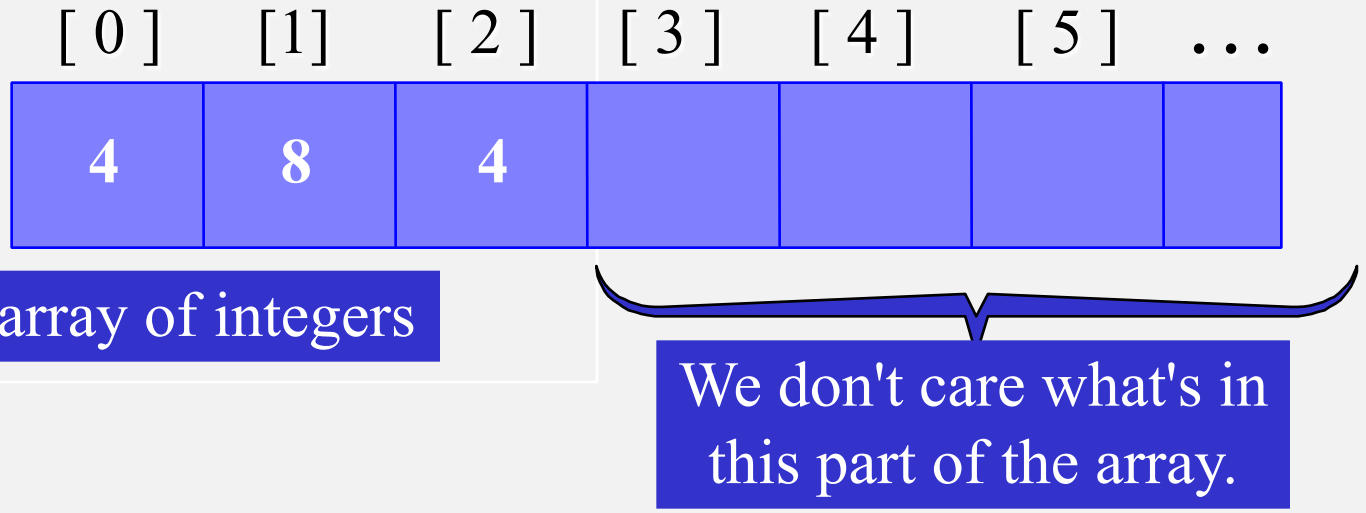
Suppose that a Mysterious Benefactor provides you with the bag class, but you are only permitted to read the documentation in the header file. You cannot read the class definition or implementation file. Can you write a program that uses the bag data type ?

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.

Implementation Details

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



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				

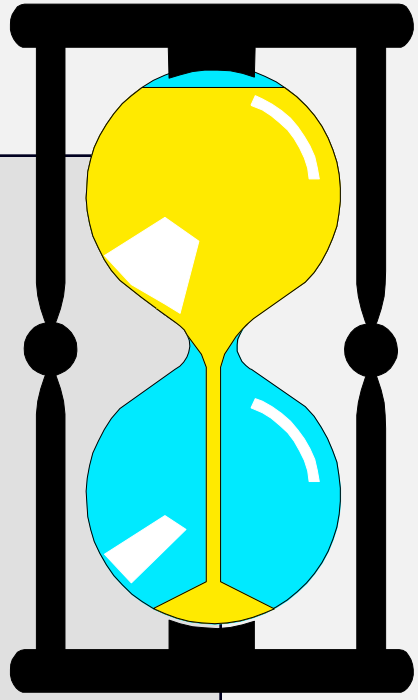
An array of integers

We don't care what's in this part of the array.

Bag: Private Members (1st Try)

One solution:

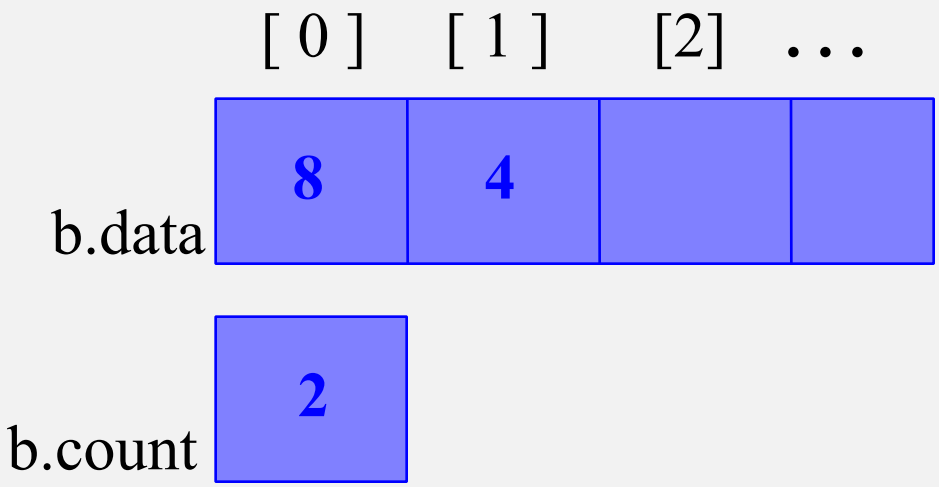
```
class bag
{
public:
    ...
private:
    int  data[20];
    size_t count;
};
```



An Example of Calling Insert

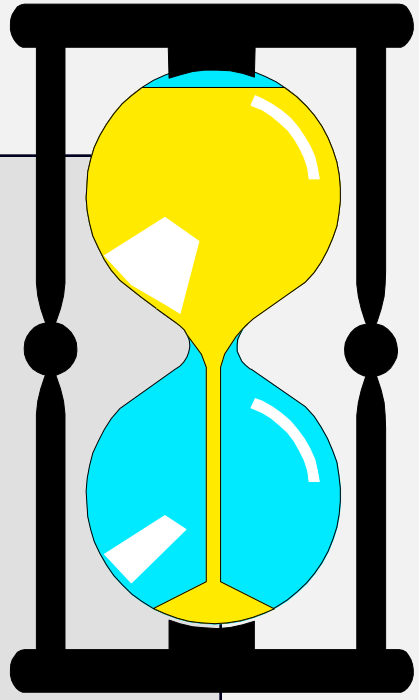
```
void bag::insert(int new_entry)
```

Before calling insert, we might have this bag b:



Bag: Private Members (2nd Try)

A more flexible solution:

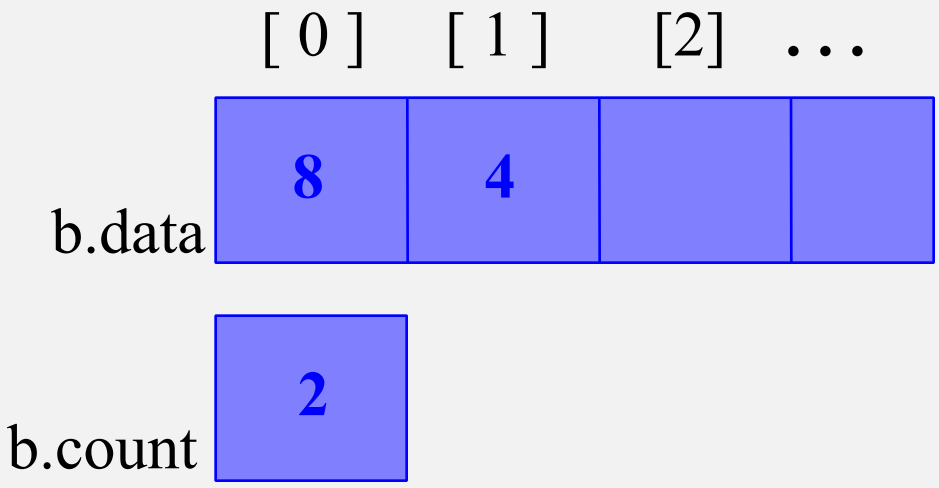


```
class bag
{
public:
    static const size_t CAPACITY = 20;
    ...
private:
    int  data[CAPACITY];
    size_t count;
};
```

An Example of Calling Insert

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

We make a function call
b.insert(17)

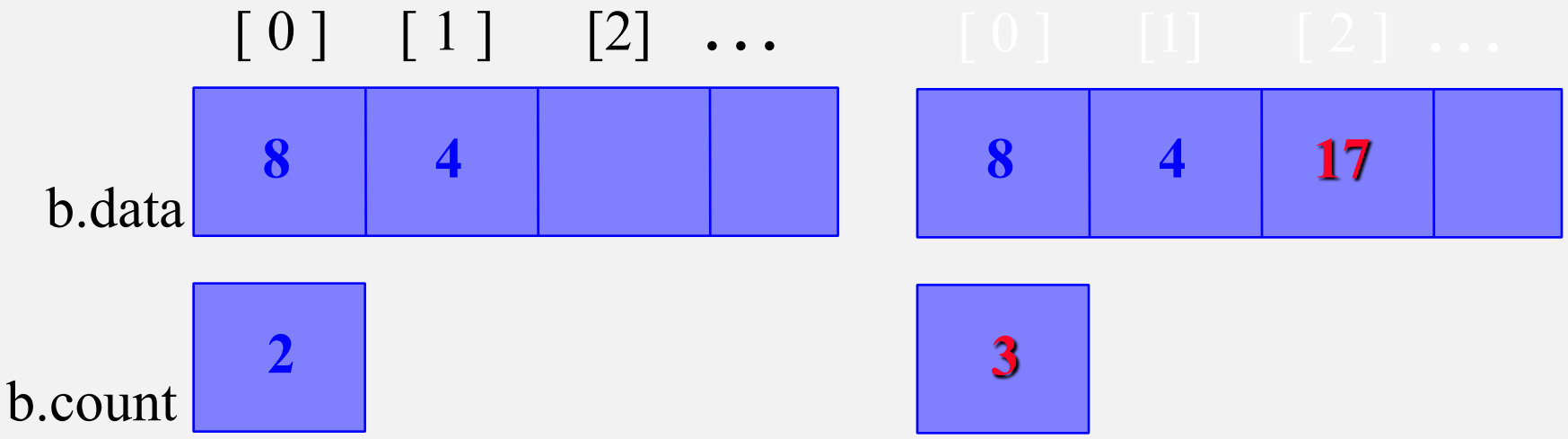


What values will be in b.data and b.count after the member function finishes ?

An Example of Calling Insert

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

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



Pseudocode for bag::insert

1. assert(size( ) < CAPACITY);
2. Place new_entry in the appropriate location of the data array.
3. Add one to the member variable count.

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

Pseudocode for bag::insert

1. assert(size( ) < CAPACITY);
2. Place new_entry in the appropriate location of the data array.
3. Add one to the member variable count.

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

Pseudocode for bag::insert

1. assert(size( ) < CAPACITY);
2. Place new_entry in the appropriate location of the data array.
3. Add one to the member variable count.

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

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 ?

A typedef for value_type

- Instead of forcing the bag to always include integers, we use the name value_typefor 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**

Implementation: A typedef for 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;
    ...
```

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

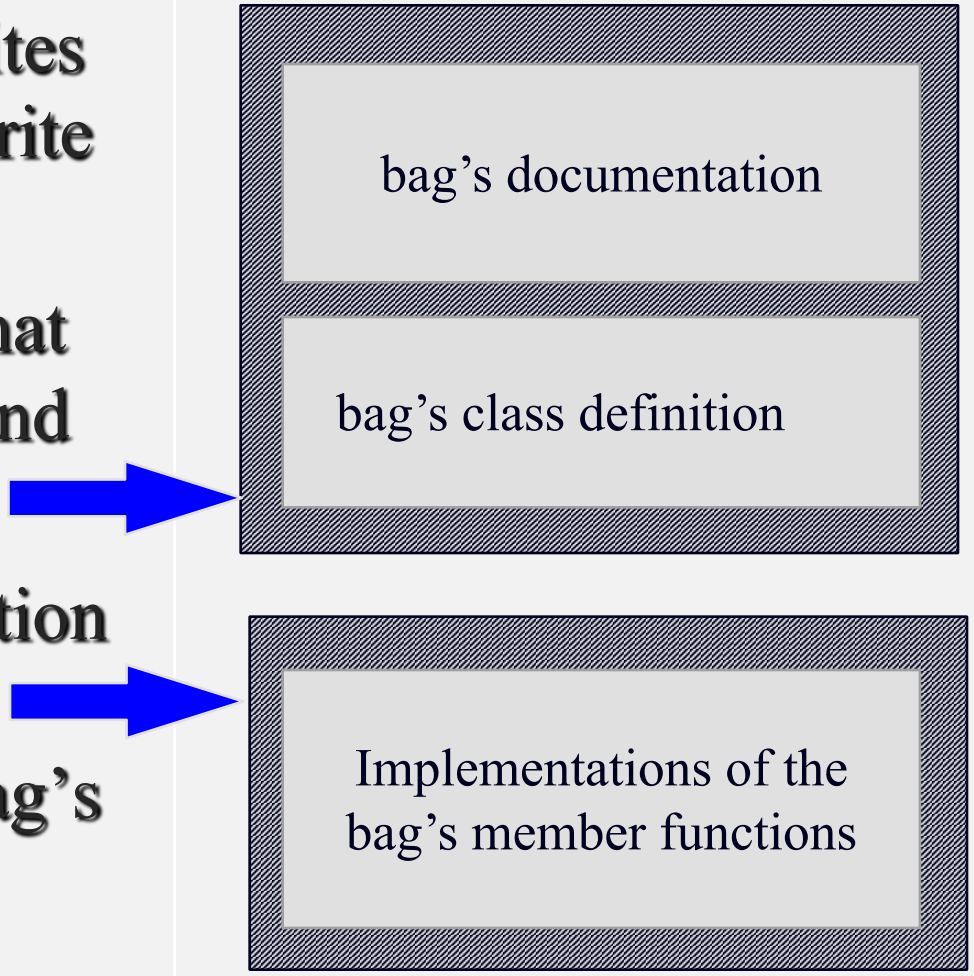
```
class bag
{
public:
    typedef int value_type;
    typedef std::size_t size_type;
    ...
```

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 Header File and Implementation File

- The programmer who writes the new bag class must write two files:
- **bag1.hpp**, 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 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.
```

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).
//
// bool 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.
```

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 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**
- **check** https://en.wikipedia.org/wiki/Class_invariant

+= Operator

```
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?**

"+=" Correct implementation

- 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;
}
```

Time Analysis for the Bag Functions

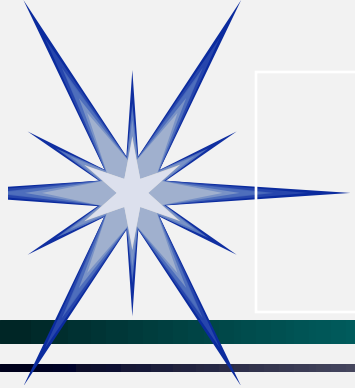
Operation	Time Analysis	
Default constructor	$O(1)$	Constant time
count	$O(n)$	n is the size of the bag
erase_one	$O(n)$	Linear time
erase	$O(n)$	Linear time

- `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



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

The End

- **The following copyright notices apply to some of the materials in this presentation:**
- Presentation copyright 2010, Addison Wesley Longman
- For use with *Data Structures and Other Objects Using C++*
- by Michael Main and Walter Savitch.
- Some artwork in the presentation is used with permission from Presentation Task Force (copyright New Vision Technologies Inc.) and Corel Gallery Clipart Catalog (copyright Corel Corporation, 3G Graphics Inc., Archive Arts, Cartesia Software, Image Club Graphics Inc., One Mile Up Inc., TechPool Studios, Totem Graphics Inc.).
- C++: Classes and Data Structures Jeffrey S. Childs, Clarion University of PA,
- © 2008, Prentice Hall
- Data Structures and Algorithm Analysis in C++, by Mark A. Weiss, ©Pearson
- Data Structures and Algorithms in C++, 2nd Edition, Michael T. Goodrich, Roberto Tamassia, David M. Mount, February 2011, ©2011