



## Container Classes



---



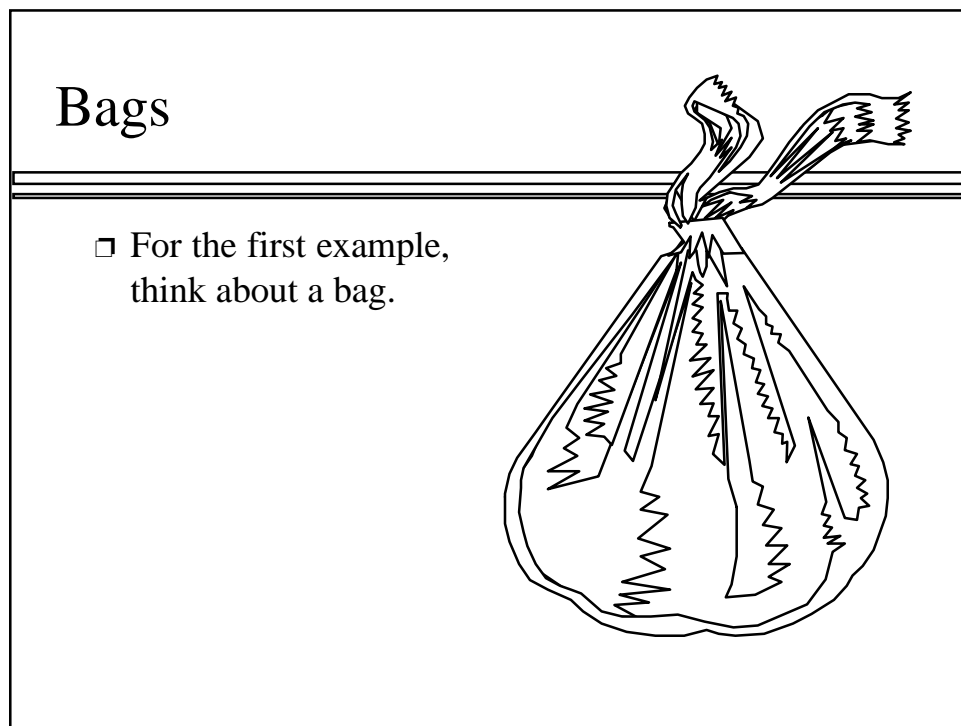
**Data Structures  
and Other Objects  
Using C++**

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

This lecture introduces container classes from Chapter 3. Before this lecture, students should know about these items:

1. How to implement simple classes, such as those in Chapter 2.
2. How to separate the class into a header file and an implementation file.

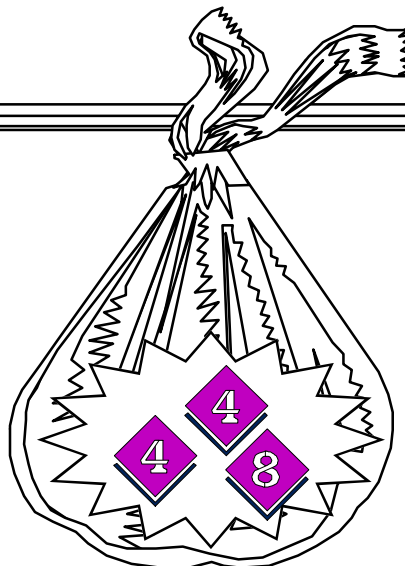
This lecture presents container classes using an example: the Bag class. After this lecture, it would be a good idea to give further coverage of some issues from Section 3.1. In particular, you should cover the use of the typedef in the Bag implementation, and the += and + operators, which are not covered in this presentation.



Start by thinking about a bag -- a gym bag, a grocery bag, whatever your favorite bag is.

## Bags

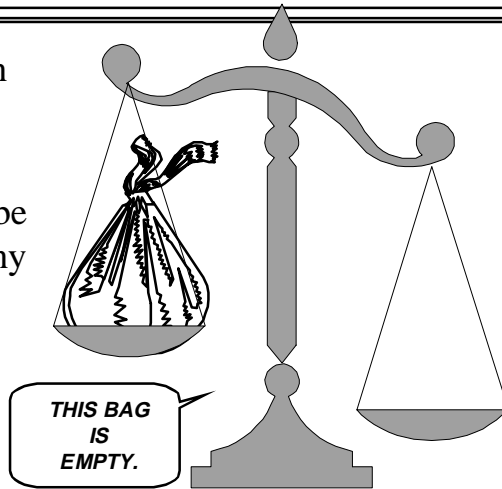
- For the first example, think about a bag.
- Inside the bag are some numbers.



Inside the bag, is a collection of numbers, such as the collection of two fours and an eight shown here.

## Initial State of a Bag

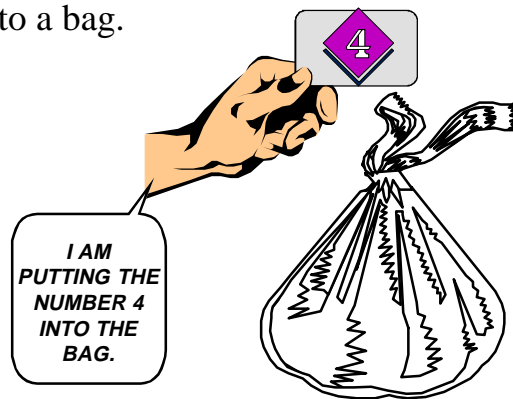
- ❑ When you first begin to use a bag, the bag will be empty.
- ❑ We count on this to be the **initial state** of any bag that we use.



This bag will be our first example of a container class, which is a class where each object can contain a collection of items (such as numbers). One of the important facets of a container class is that each object begins in a known configuration. In the case of a bag, we will count on each bag being initially empty. This is called the initial state of a bag.

## Inserting Numbers into a Bag

- Numbers may be inserted into a bag.



We will use a fixed collection of operations to manipulate bags. One of the simplest operations that we permit is the action of inserting a new number into a bag.

## Inserting Numbers into a Bag

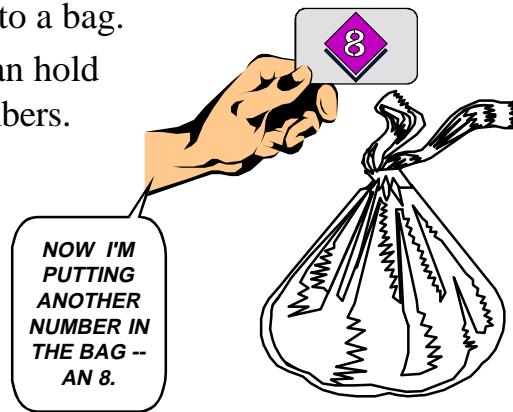
- Numbers may be inserted into a bag.



In this example, we have inserted the first number -- a four -- into a bag that was previously empty.

## Inserting Numbers into a Bag

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



We can continue inserting new numbers into the bag.

## Inserting Numbers into a Bag

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



Now we have a four and an eight in the bag.



## 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.



We can even insert a second four into the same bag, as shown here.

## 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.

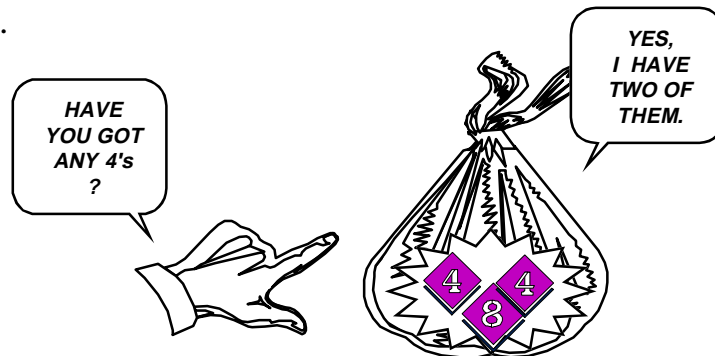


At this point we have two fours and one eight in the bag of numbers.

I want to make a small point on the side: A bag's behavior is a bit different than a set of numbers. A set is not allowed to have two copies of the same number. But a bag can have many copies of the same number.

## Examining a Bag

- We may ask about the contents of the bag.

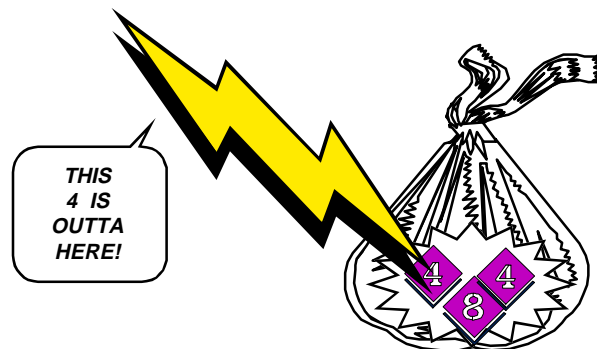


At this point, we know the initial state of a bag (it is empty) and we also have one operation (inserting a number). Here is our second bag operation: an ability to query a bag about its contents. In particular, we can ask the bag how many copies does it contain of a given number.

In this example, what would the bag respond if I asked for eights instead of fours. ("Yes, I have one eight.") What if I ask for tens instead of fours. ("No, I have no tens.").

## Removing a Number from a Bag

- We may remove a number from a bag.



Here is our third bag operation: removing 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.



In this example, I removed one of the fours, but the other four remains intact. Numbers are removed one at a time, so that if there are many fours, and I remove one of them, other fours can remain in the bag.

## How Many Numbers

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



Another operation allows us to find out how many numbers are in the bag at the moment.

## Summary of the Bag Operations



- ❶ A bag can be put in its **initial state**, which is an empty bag.
- ❷ Numbers can be **inserted** into the bag.
- ❸ You may check how many **occurrences** of a certain number are in the bag.
- ❹ Numbers can be **removed** from the bag.
- ❺ You can check **how many** numbers are in the bag.

We have talked about four bag operations, but we actually have five since the process of putting a bag into its initial state counts as an operation. This slide just summarizes the five bag operations. By the way, which of these five operations is likely to be implemented via the bag constructor?

...

At this point, the bag is truly abstract, since we haven't decided how the bag will be implemented. Nevertheless, we still have a good idea of how a bag might be used to solve various problems.

Note: At this point, you might pause the lecture to demonstrate how a paper bag can be used to remember the ages of several students in the class. Start by initializing the bag, then insert several students' ages, make a few queries about the bag, and finally remove the ages. This is a good place to point out that additional bag operations might be useful, such as an operation of combining two bags. The example bag in Section 3.1 does have some additional operations.

## 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
```

Let's start to look at the implementation of a bag as a C++ class. The class definition begins as shown in this slide.

A question: Suppose this class definition has been completed. How would a program declare variables for three different bags that the program uses? Answer:

```
Bag a, b, c;
```



## The Bag Class

- ❑ C++ classes (introduced in Chapter 2) can be used to implement a container class such as a Bag.
- ❑ The class definition includes:
  - ✓ **The heading of the definition**
  - ✓ **A constructor prototype**

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

In the public part of the Bag class definition, we begin by listing the constructor prototype...

## The Bag Class

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

```
class Bag
{
public:
    Bag( );
    void insert(...
    void remove(...
    ...and so on
```

In the public part of the Bag class definition, we begin by listing the constructor prototype... and then we list the prototypes for the other member functions.

In our example, we have at least four member functions to list. As a general rule, querying operations (which don't change the contents of the ADT) should be implemented as const member functions. Other operations (which do change the contents) must be implemented as ordinary member functions. Question: Will either insert or remove be a const member function?

## 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.
};

```

Important note: The intention with a class is that the only way that an object is manipulated is through its public operations. Even if we have “inside information” about the private member variables, we don’t want our programs to use that information. Instead, the only ways that our programs will manipulate a bag is through the public functions.

This approach is called information hiding. What advantages do you see to the approach? 1. The programmer who uses the class does not need to clutter his or her thinking with details of the implementation. 2. At a later point, we can decide to implement the class in a different way, and programs which use the class will still work.

This should sound familiar: They are the same reasons that we use preconditions and postconditions to specify what a function does without indicating how the function does its work.

## The Bag's Default Constructor

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

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

In fact, it's a good idea to provide a precondition/postcondition specification with each of the class's operations. Here's an example of what the specification looks like for the bag's constructor. Note that there is no precondition, so we have omitted it from the listing.

## The Insert Function

- Inserts a new number in the bag

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

The specification for the bag's insert function is shown here.

## The Size Function

- Counts how many integers are in the bag.

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

The specification for the bag's size function is shown here. Notice that the function returns an integer telling how many items are in the bag. Also, this is a const member function because it does not change the contents of a bag.

There's actually a better data type to use for the return value, as shown here...

## The Size Function

- Counts how many integers are in the bag.

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

I have changed the return type to `size_t`, which is a data type that is defined in `stdlib.h`. This data type can be used for any non-negative integer values. Also, every C++ implementation guarantees that the `size_t` values are large enough to hold the size of any object that can be declared on the current machine. Therefore, it's a good idea to use the `size_t` type when you are describing the size of an object (such as the size of a bag).

## The Occurrences Function

- Counts how many copies of a number occur

```
size_t Bag::occurrences(int target) const
// Postcondition: The return value is the number
// of copies of target in the Bag.
{
    ...
}
```

The specification for the bag's occurrences function is shown here. Notice that the return value is once again a `size_t` value, and this is also a `const` member function.



## The Remove Function

- Removes one copy of a number

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

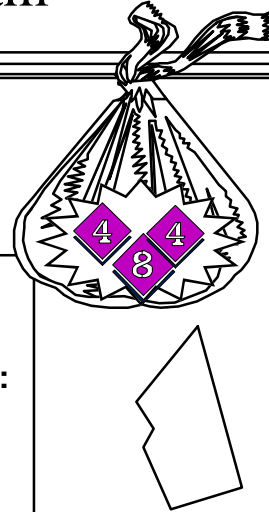
The specification for the bag's remove function is shown here.

## 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);
```



Let's take a quick look at how a program might declare and use a bag. In this example, the program declares a bag called `ages`, and inserts the three numbers 4, 8, and 4 into the `ages` bag.

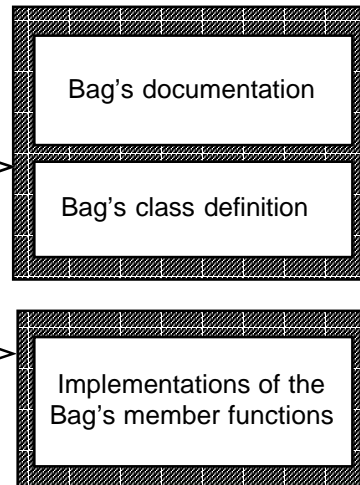
Question: What include statement would be needed for this program to declare and use a bag?

...

Answer: The new Bag class should be implemented in two separate files: A header file and an implementation file. In order to use the Bag class, a program must include the bag's header file.

## The Header File and Implementation File

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



Let's look at some details of the bag's header file and implementation file.

The header file has two parts: Documentation telling how to use the Bag class, and the actual Bag class definition. In this example, we have called the header file `bag1.h` (because we intend to have further implementations that will be `bag2`, `bag3`, and so on).

The implementation file, called `bag1.cxx`, contains the implementations of the Bag 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.

Bag's documentation

Bag's class definition

Implementations of the Bag's member functions

Keep in mind that the bag's documentation should list the prototypes and specifications for all of the bag's functions.

## 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(...
```

Bag's documentation

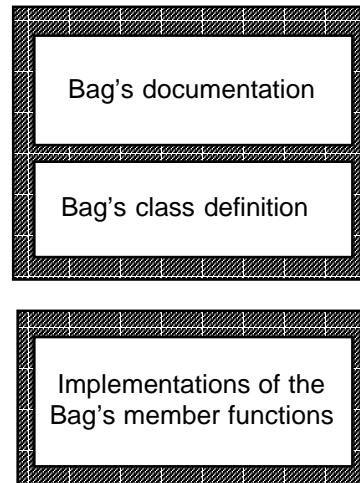
Bag's class definition

Implementations of the  
Bag's member functions

The Bag's class definition is the definition that we have already seen (or at least seen most of). It lists the prototypes of the bag member functions in the public section, and will list private member variables in the private section.

## 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.



The bodies for the bag's functions appear in the separate 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.
- 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.

Time for a quiz . . .

## 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.

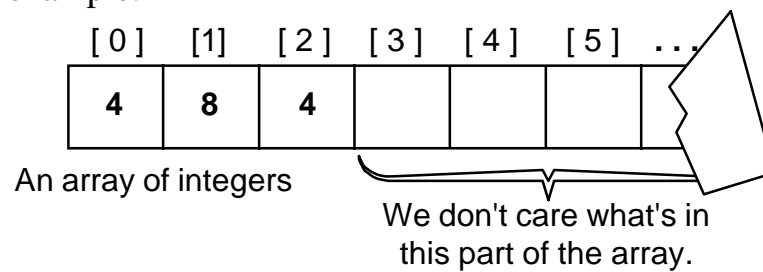
We'll see this concept over and over again. In order to use something, you don't need to know implementation details -- and, in fact, your thinking is usually clearer if implementation details remain hidden.

By the way, a sample program using a bag is given in Section 3.1 of the text, and this program actually appears before the implementation details of the Bag class.



## Implementation Details

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



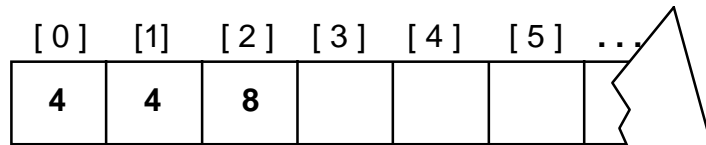
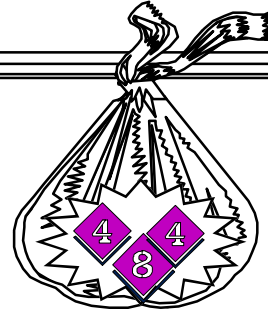
Just to finish things off, let's look at some details of how the Bag class could be implemented. This isn't the only way to implement the class, but it is a simple approach.

The plan is to store the entries of a bag in the front part of an array, sometimes called a partially-filled array.

For example, if we have an array that contains two fours and an eight, then we would place those three numbers in the first three components of an array. We don't care what appears beyond those first three components.

## Implementation Details

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



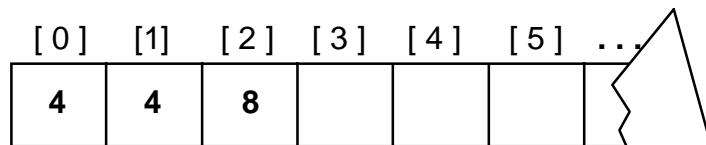
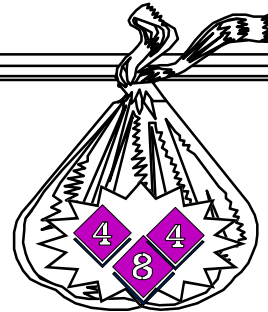
An array of integers

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

We also don't care what order the entries appear in. We might have both fours first . . .

## Implementation Details

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



An array of integers

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

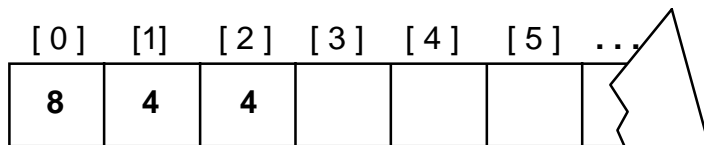
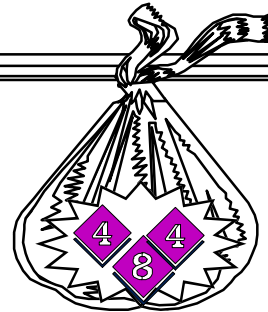
. . . or we might have the eight first.

## 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



An array of integers

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

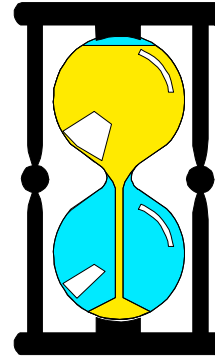
There is one more item that we do need to keep track of, and that's the total number of entries in the bag. For this example, the total number of entries in the bag is three.

What would go wrong if we didn't keep track of this number? Answer: We wouldn't know what part of the array was being used, and what part of the array was just garbage.

Another question: How would an empty bag be represented? Answer: The bag's size is zero, and the entire array may be garbage since we are not using any of the array.

## An Exercise

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



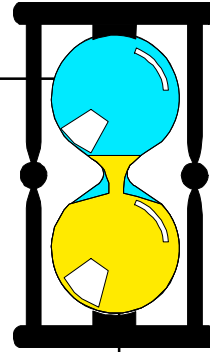
***You have 60 seconds to write the declaration.***

Time for another quiz . . .

## An Exercise

One solution:

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



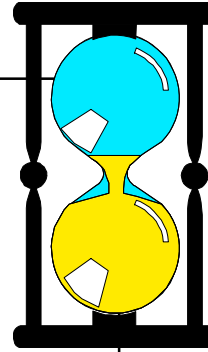
This is not the only solution. You might have used different names for the two private member variables. The important idea is that a private member variable can be an array.

There's at least one other facet that we should improve. The number 20 is somewhat arbitrary. I might want a bag with a capacity of 200 or 2000 or even 20000. In order to make it clear that the capacity can be easily changed, we should declare the capacity as a constant in the public portion of the bag, like this...

## An Exercise

A more flexible solution:

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



The definition “static const size\_t CAPACITY = 20” defines a number called CAPACITY that can be used anywhere within the bag implementation.

The keyword static means that all bags have the same CAPACITY.

The keyword const means that this number cannot change while a program is running (although we could change it and recompile).

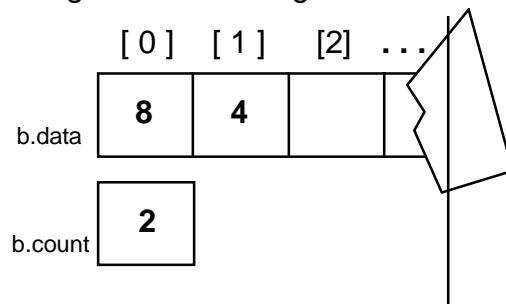
The constant is declared with all capital letters; this isn’t a requirement in C++, but most C++ programmers follow the practice of using capitals for constants’ names.

Within a program, you can declare a bag b, and refer to the constant as b.CAPACITY.

## An Example of Calling Insert

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

Before calling insert, we  
might have this bag b:



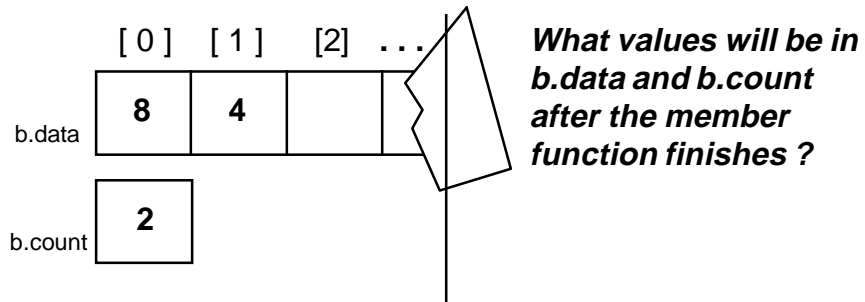
Now that we know about the implementation details of the bag member variables, let's look at an example of calling `Bag::insert`. In the example, we start with the bag shown here . . .



## An Example of Calling Insert

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

We make a function call  
`b.insert(17)`



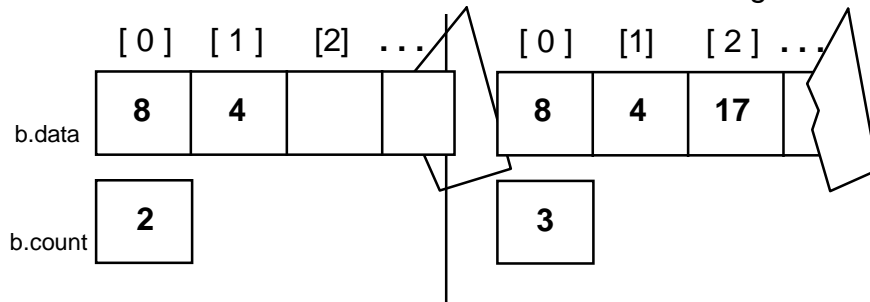
... and we call `Bag::insert` to insert the number 17.

What will be the values of `b.data` and `b.count` after the insertion?

## An Example of Calling Insert

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

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



We have added the new number, 17, to the next spot in the array; and we have incremented `b.count` by one to indicate that we are now using one more spot in the array.

## Pseudocode for Bag::insert

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

*What is the “appropriate location” of the data array ?*

Here’s the pseudocode for implementing `Bag::insert`. We start by checking that the precondition is valid -- in other words that the bag has room for another entry. An important note: These assertions should be carried out using public members (such as `size` and `CAPACITY`) rather than private members (such as `count`). The use of public members makes the potential error messages more meaningful.

Next we place the `new_entry` in the array.

Finally we add one to the count.

What would the C++ code look like for Steps 2 and 3?

## Pseudocode for Bag::insert

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

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

Here is one solution for Steps 2 and 3. In our example, this would cause the `new_entry` (17) to be placed at index [2] of the array `b.data`, and then `count` is incremented to 3.

## Pseudocode for Bag::insert

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

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

Here is an alternative that combines Steps 2 and 3. In this alternative, the index `[count++]` is evaluated and used before `count` is incremented. If we wanted the incrementing to occur before evaluation, we would write `++count` instead.

## The Other Bag Operations

- ❑ Read Section 3.1 for the implementations of the other bag member functions.
- ❑ Remember: If you are just **using** the Bag class, then you don't need to know how the operations are implemented.
- ❑ Later we will **reimplement** the bag using more efficient algorithms.
- ❑ We'll also have a few other operations to manipulate bags.

You can read Section 3.1 for the complete bag implementation. However, the bag class itself is not particularly important; the important point is the concept of a container class, and the advantages that classes provide.

Later we will reimplement this same bag in more efficient ways.

## Other Kinds of Bags


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

*Suppose you wanted one of these other bags. How much would you need to change in the implementation ?*

*Section 3.1 gives a simple solution using the C++ typedef statement.*

Here's one last question for you to think about. Of course, the answer is that there is very little difference between a bag of integers and a bag of any other type.

For more on this issue, you should read about typedef statements in Section 3.1. Typedef statements are the first simple way to write a container class where the underlying data type can be easily changed. Later (in Chapter 6) we will see a more powerful alternative to typedef statements.



## Summary

---

---

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

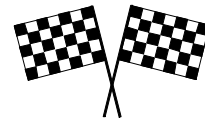
A quick summary . . .



Presentation copyright 1997, 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.).

Students and instructors who use *Data Structures and Other Objects Using C++* are  
welcome to use this presentation however they see fit, so long as this copyright notice  
remains intact.



THE END