CSCE 121

More Object Examples
Container Classes

Dr. Tim McGuire

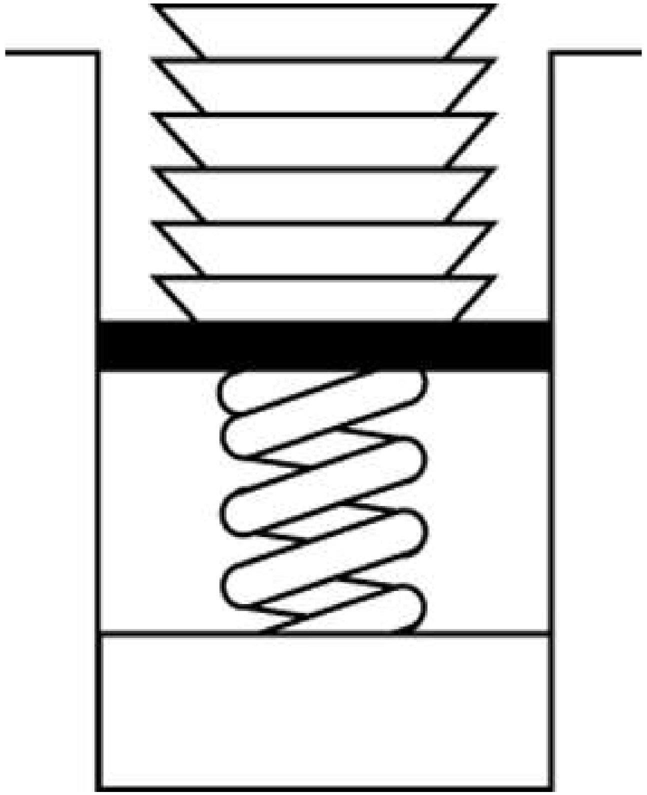
Some of the material and images from Michael Main, University of Colorado, and Tony Gaddis, Haywood Community College

Introduction to the Stack Abstract Data Type

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria
  - return addresses for function calls
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

Developing an ADT During the Design of a Solution

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria



Example:

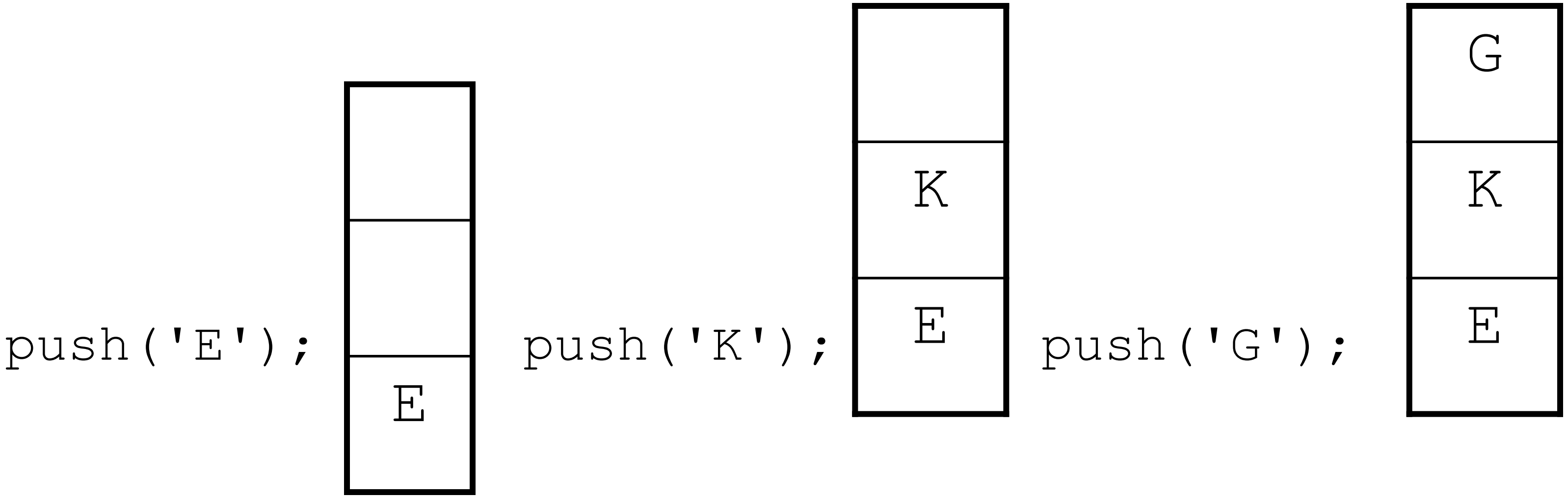
Stack of cafeteria dishes

Stack Operations and Functions

- Operations:
  - push: add a value onto the top of the stack
  - pop: remove a value from the top of the stack
- Functions:
  - `isFull`: `true` if the stack is currently full, *i.e.*, has no more space to hold additional elements
  - isEmpty: `true` if the stack currently contains no elements

Stack Operations - Example

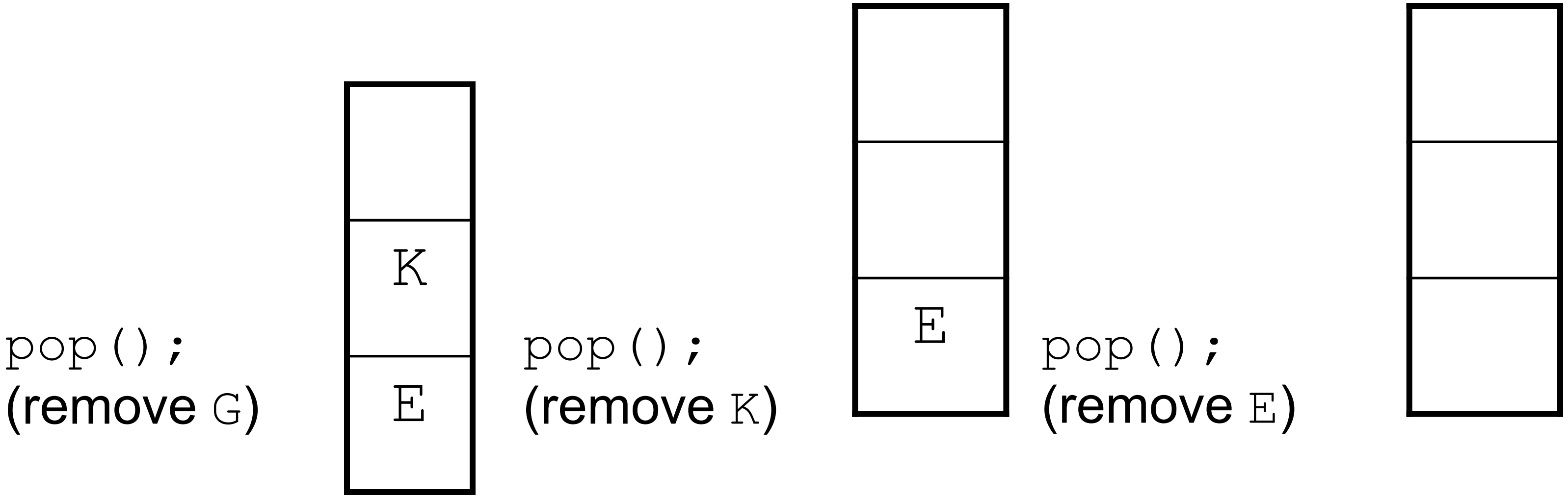
- A stack that can hold `char` values:



```
push('E');
push('K');
push('G');
```

Stack Operations - Example

- A stack that can hold `char` values:

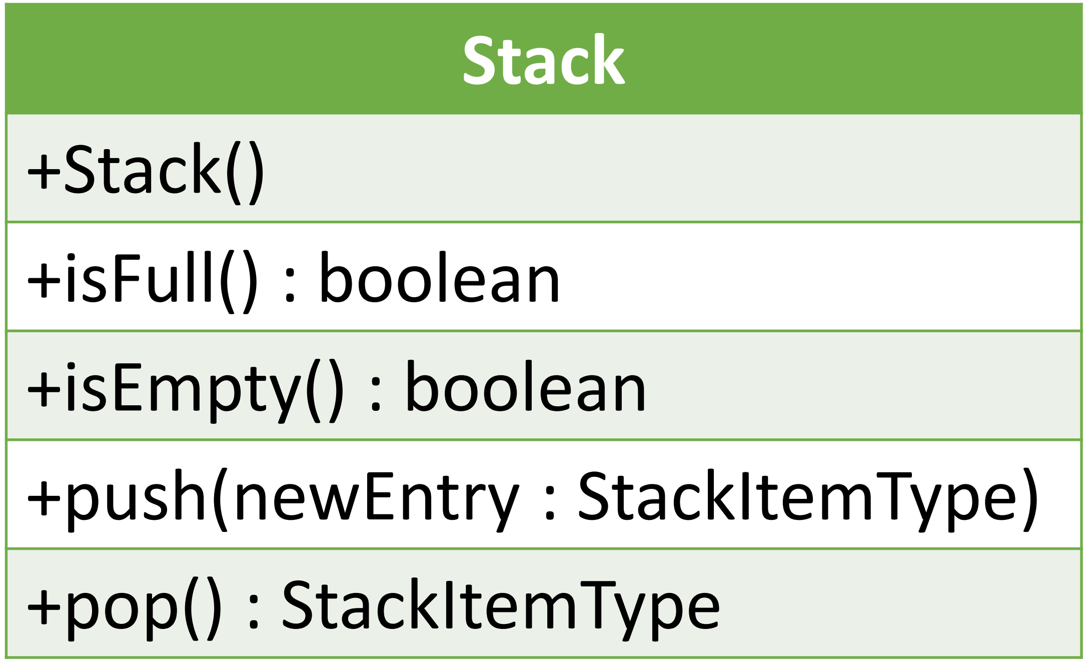


`pop();`
(remove `G`)

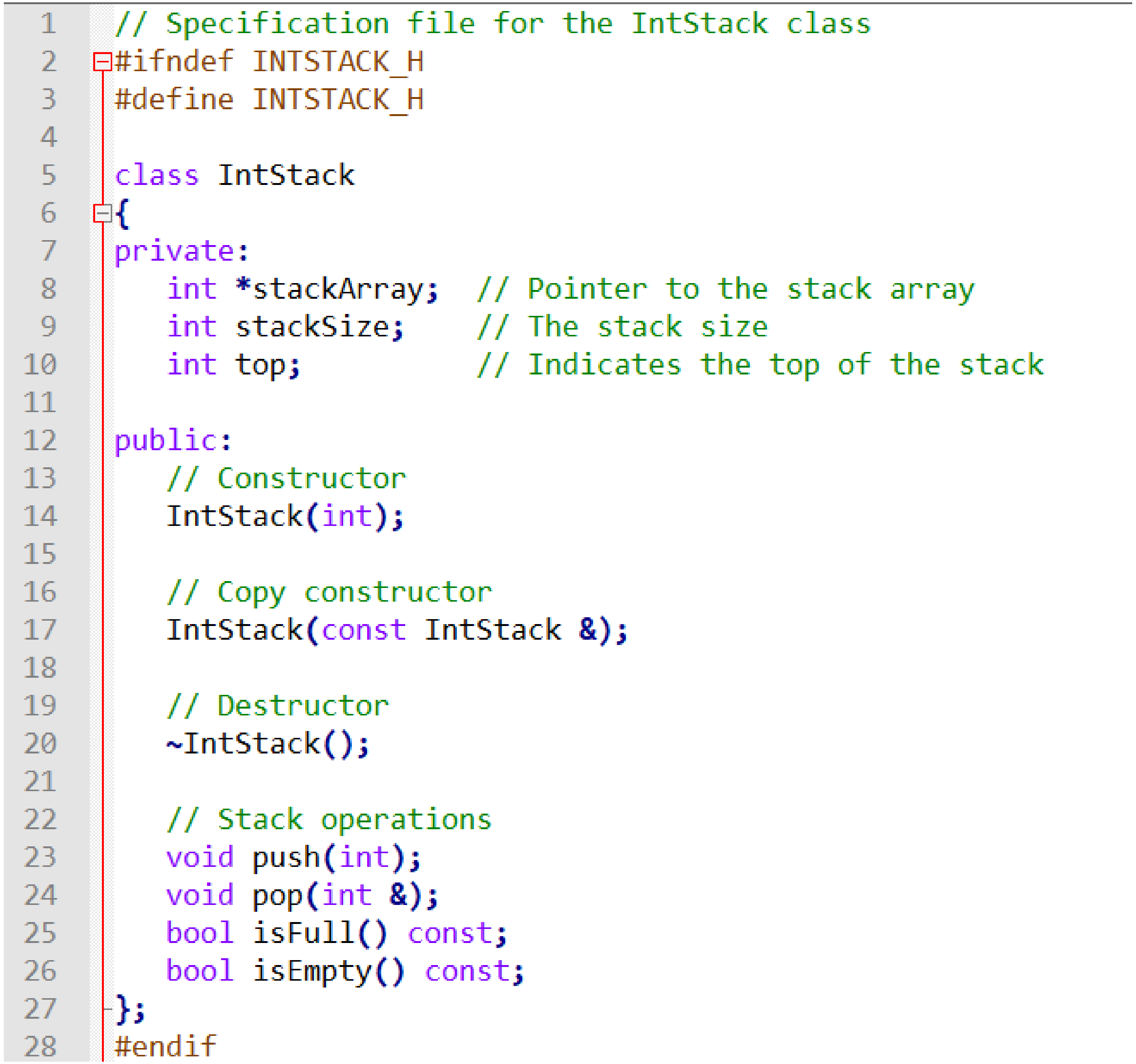
`pop();`
(remove `K`)

`pop();`
(remove `E`)

UML Diagram for the class `Stack`



Stack
+Stack()
+isFull() : boolean
+isEmpty() : boolean
+push(newEntry : StackItemType)
+pop() : StackItemType



```cpp
// Specification file for the IntStack class
#ifndef INTSTACK_H
#define INTSTACK_H

class IntStack
{
private:
   int *stackArray;  // Pointer to the stack array
   int stackSize;    // The stack size
   int top;          // Indicates the top of the stack

public:
   // Constructor
   IntStack(int);

   // Copy constructor
   IntStack(const IntStack &);

   // Destructor
   ~IntStack();

   // Stack operations
   void push(int);
   void pop(int &);
   bool isFull() const;
   bool isEmpty() const;
};
#endif
```

See:
IntStack.h
IntStack.cpp
IntStackDemo.cpp

Templated Stacks

See Stack.h, TemplatedStackDemo.cpp

Dynamic Stacks

Dynamic Stacks

- Grow and shrink as necessary
- Can't ever be full as long as memory is available
- Implemented as a linked list

Implementing a Stack

- Programmers can program their own routines to implement stack functions

- See `DynIntStack` class for an example.
- A templated class example is in `DynamicStack.h`

- Can also use the implementation of stack available in the STL

The STL `stack` Container

The STL `stack` container

- Stack template can be implemented as a `vector`, a linked list, or a `deque`
- Implements `push`, `pop`, and `empty` member functions
- Implements other member functions:
  - `size`: number of elements on the stack
  - `top`: reference to element on top of the stack

Defining a `stack`

- Defining a stack of `char`s, named `cstack`, implemented using a `vector`:

```
stack< char, vector<char>> cstack;
```

- implemented using a list:

```
stack< char, list<char>> cstack;
```

- implemented using a `deque`:

```
stack< char > cstack;
```

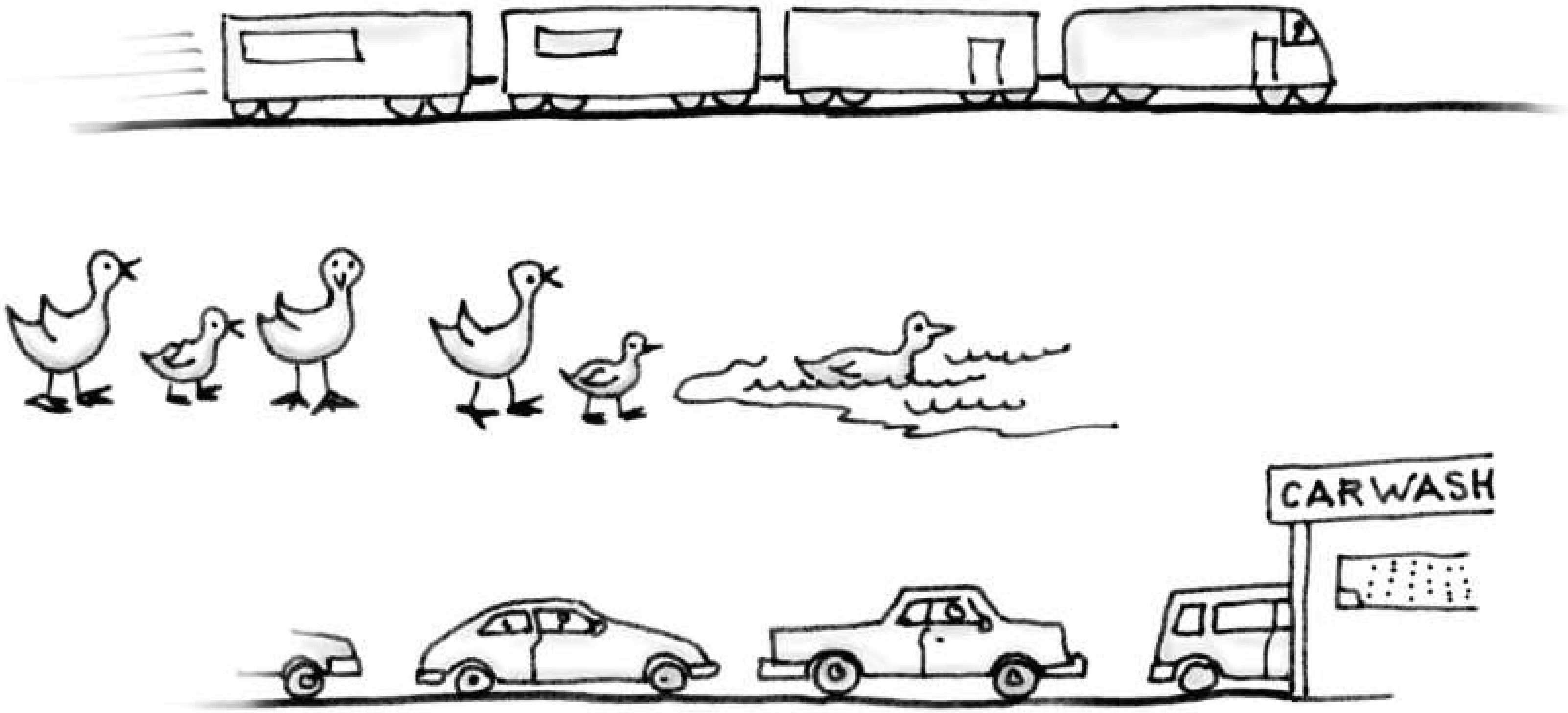
- See StackSTLdemo.cpp for an example

Introduction to the Queue ADT

Introduction to the Queue ADT

- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print jobs sent to a printer
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

Some everyday queues.



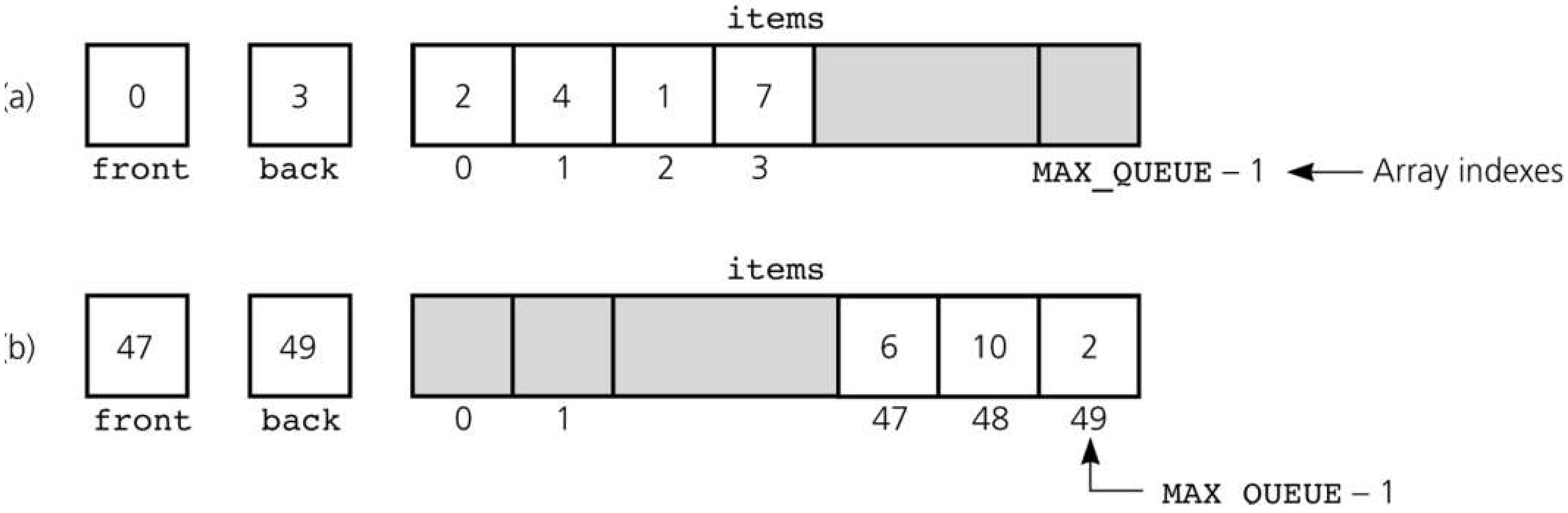
The Abstract Data Type Queue

- Queues
  - Are appropriate for many real-world situations
    - Example: A line to buy a movie ticket
  - Have applications in computer science
    - Example: A request to print a document
- A simulation
  - A study to see how to reduce the wait involved in an application

Queue Locations and Operations

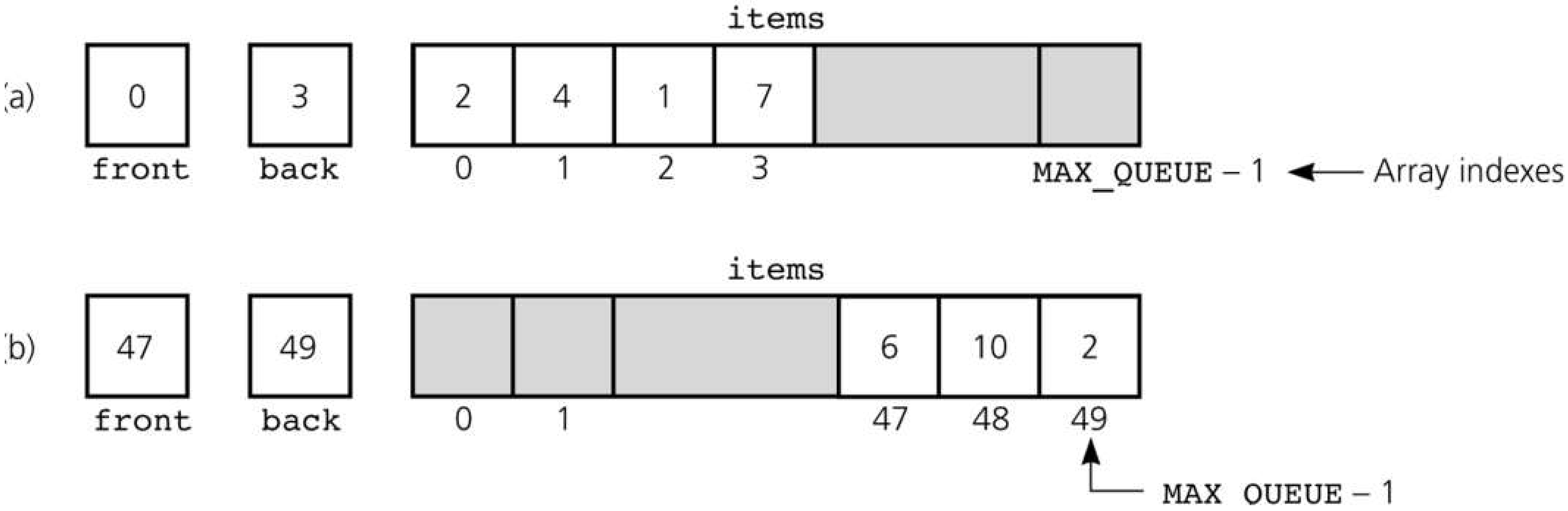
- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

An Array-Based Implementation (1st Attempt)



a) A naive array-based implementation of a queue;
b) rightward drift can cause the queue to appear full

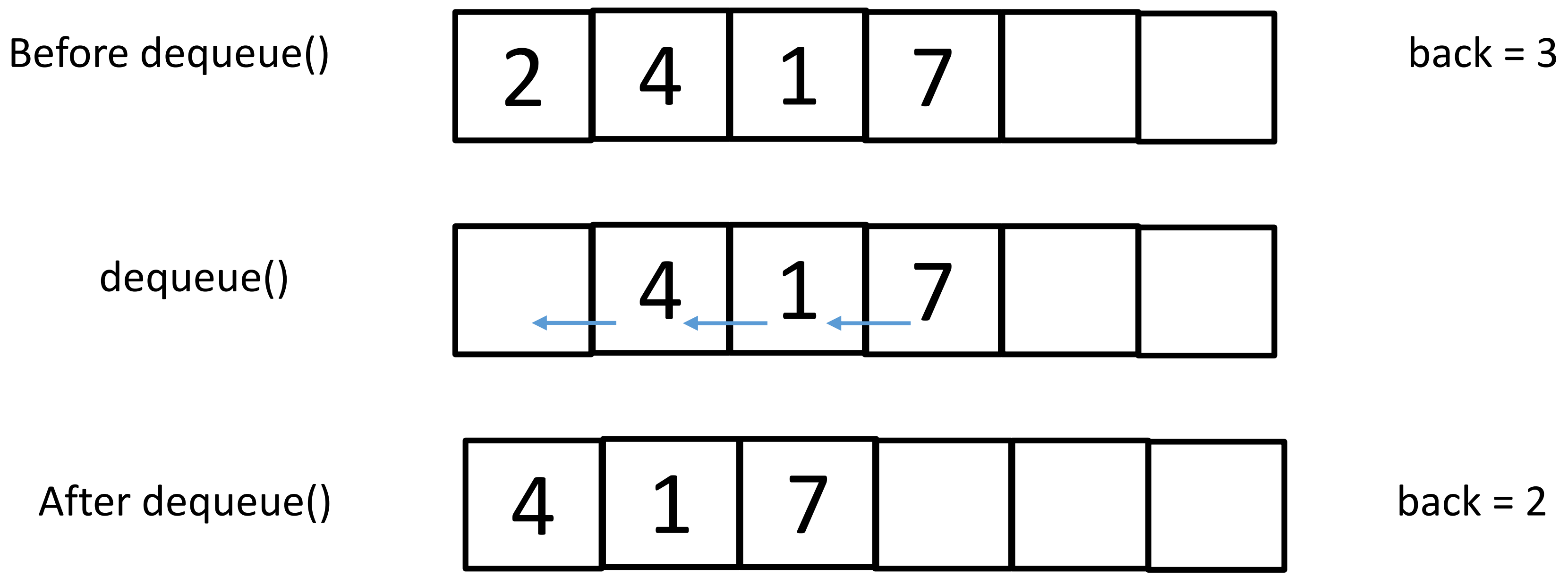
A Possible solution



When removing an item (dequeue() operation) shift all items toward the front

In that way, just like the stack, if the back pointer ever reaches MAX – 1, the queue is full.

A Possible solution



Before dequeue()	2	4	1	7			back = 3
dequeue()		4	1	7			
After dequeue()	4	1	7				back = 2

When removing an item (dequeue() operation) shift all items toward the front

In that way, just like the stack, if the back pointer ever reaches MAX – 1, the queue is full.

Contents of IntQueue.h

```
// Specification file for the IntQueue class
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
private:
   int *queueArray;  // Points to the queue array
   int queueSize;    // The queue size
   int front;        // Subscript of the queue front
   int rear;         // Subscript of the queue rear
   int numItems;     // Number of items in the queue
```

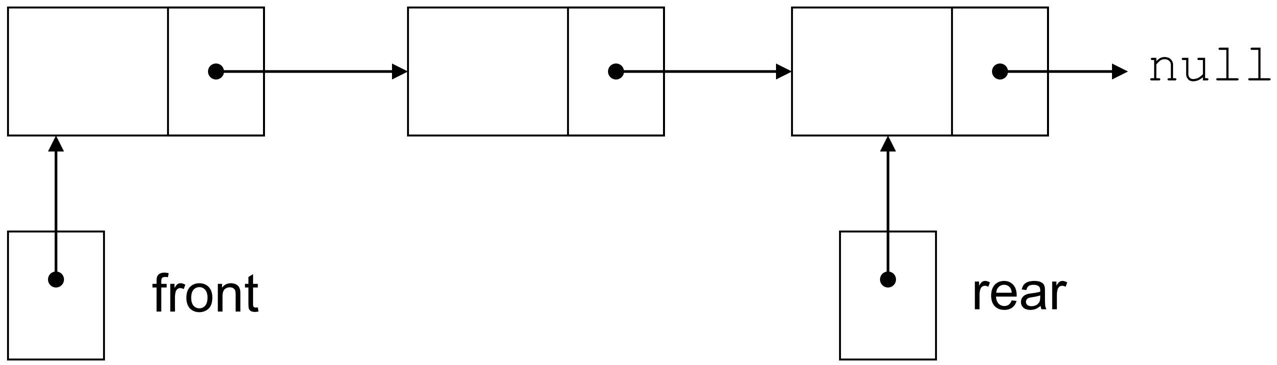
Contents of `IntQueue.h` (Continued)

```
public:
   // Constructor
   IntQueue(int);

   // Copy constructor
   IntQueue(const IntQueue &);

   // Destructor
   ~IntQueue();

   // Queue operations
   void enqueue(int);
   void dequeue(int &);
   bool isEmpty() const;
   bool isFull() const;
   void clear();
};
#endif
```

Dynamic Queues

Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices



Implementing a Queue

- Programmers can program their own routines to implement queue operations
- See the `DynIntQue` class for an example of a dynamic queue
- Can also use the implementation of queue and dequeue available in the STL

The STL `deque` and `queue` Containers

The STL `deque` and `queue` Containers

- `deque`: a double-ended queue. Has member functions to enqueue (`push_back`) and dequeue (`pop_front`)
- `queue`: container ADT that can be used to provide queue as a `vector`, list, or `deque`. Has member functions to enque (`push`) and dequeue (`pop`)

Defining a queue

- Defining a queue of chars, named cQueue, implemented using a deque:

```
deque<char> cQueue;
```

- implemented using a queue:

```
queue<char> cQueue;
```

- implemented using a list:

```
queue<char, list<char>> cQueue;
```

Container Classes

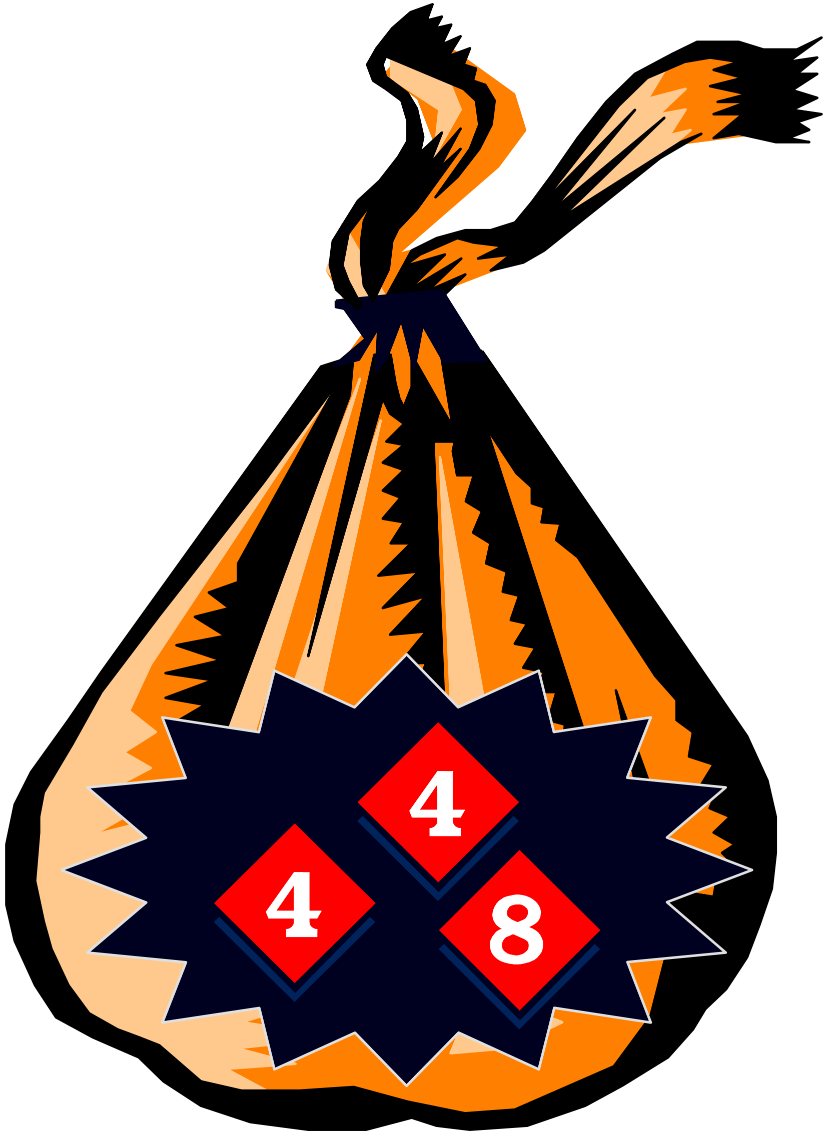
- Stacks and queues are examples of container classes.
- A container class is a data type that is capable of holding a collection of items.
- In C++, container classes can be implemented as a class, along with member functions to add, remove, and examine items.

Bags

- For example, think about a bag.

Bags

- Inside the bag are some numbers.



Initial State of a Bag

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

Inserting Numbers into a Bag

- Numbers may be inserted into a bag.

Inserting Numbers into a Bag

- Numbers may be inserted into a bag.

Inserting Numbers into a Bag

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

Inserting Numbers into a Bag

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

THE 8 IS ALSO IN THE BAG.
4
8

Inserting Numbers into a Bag

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

Inserting Numbers into a Bag

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

NOW THE BAG HAS TWO 4'S AND AN 8..
4
4
8

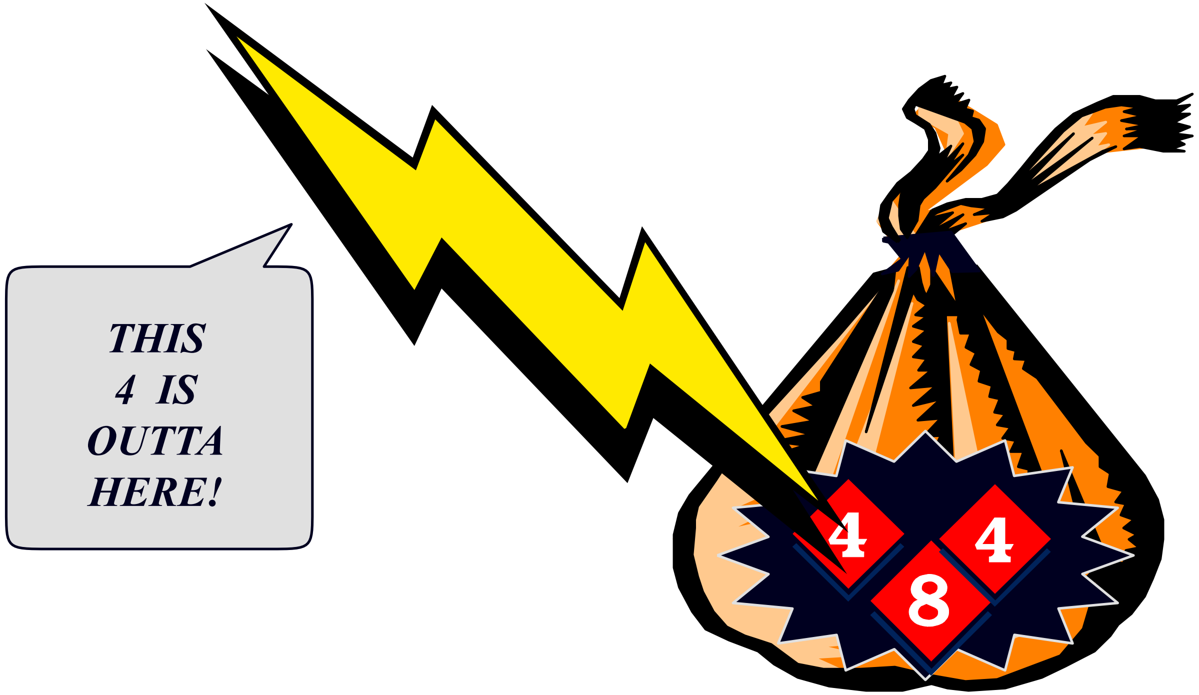
Examining a Bag

- We may ask about the contents of the bag.

HAVE YOU GOT ANY 4's ?
YES, I HAVE TWO OF THEM.
4
4
8

Removing a Number from a Bag

- We may remove a number from a bag.



Removing a Number from a Bag

- We may remove a number from a bag.
- But we remove only one number at a time.

ONE 4 IS GONE, BUT THE OTHER 4 REMAINS.
8
4

How Many Numbers

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

Summary of the Bag Operations

- A bag can be put in its **initial state**, which is an empty bag.
- Numbers can be **inserted** into the bag.
- You may 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.

A Quiz

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

- Yes I can.
- No. Not unless I see the class declaration for the bag.
- No. I need to see the class declaration for the bag , and also see the implementation file.

A Quiz

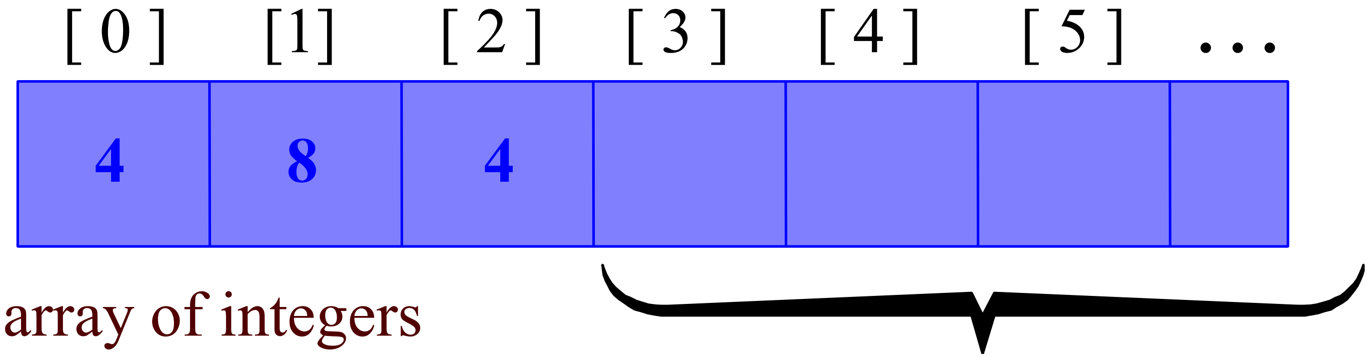
Suppose that a Mysterious Benefactor provides you with the bag class, but you are only permitted to read the documentation in the header file. You cannot read the class definition or implementation file. Can you write a program that uses the bag data type ?

- ✶ 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.



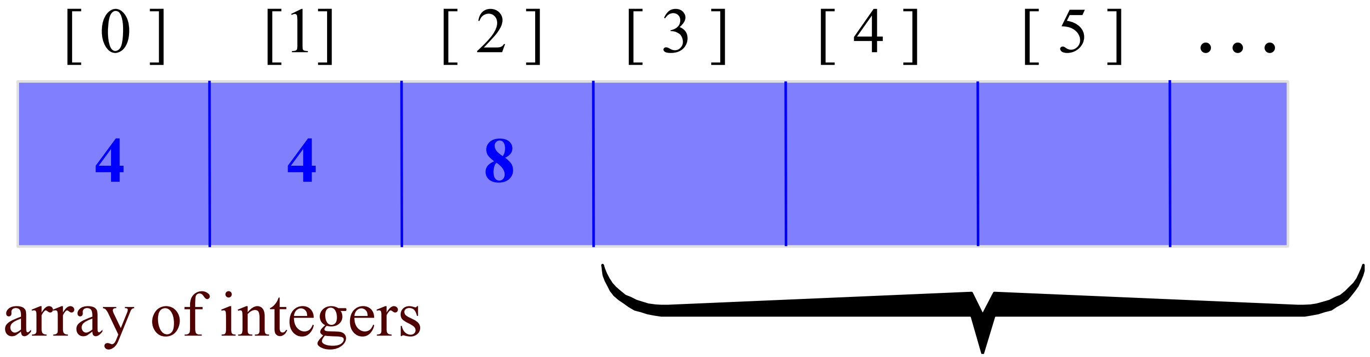
[0]	[1]	[2]	[3]	[4]	[5]	. . .
4	8	4				

An array of integers

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

Implementation Details

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



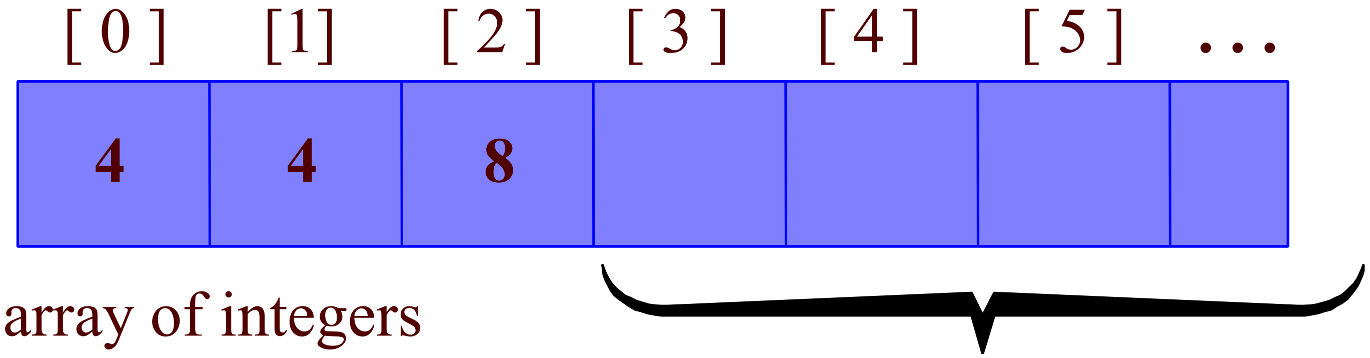
[0]	[1]	[2]	[3]	[4]	[5]	. . .
4	4	8				

An array of integers

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

Implementation Details

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



[0]	[1]	[2]	[3]	[4]	[5]	. . .
4	4	8				

An array of integers

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

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

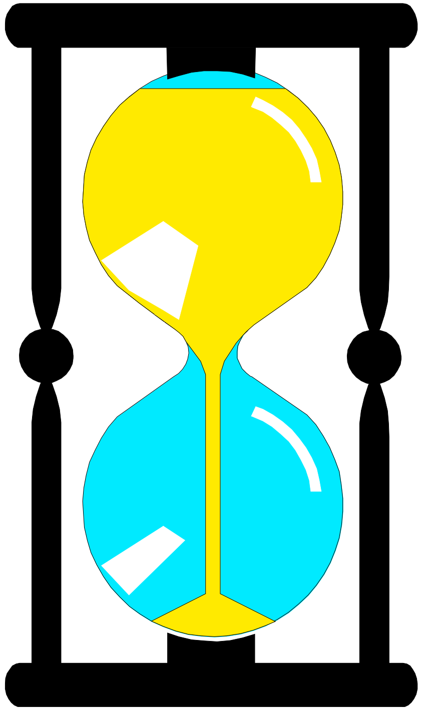
[0]	[1]	[2]	[3]	[4]	[5]	. . .
8	4	4				

An array of integers

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

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.*

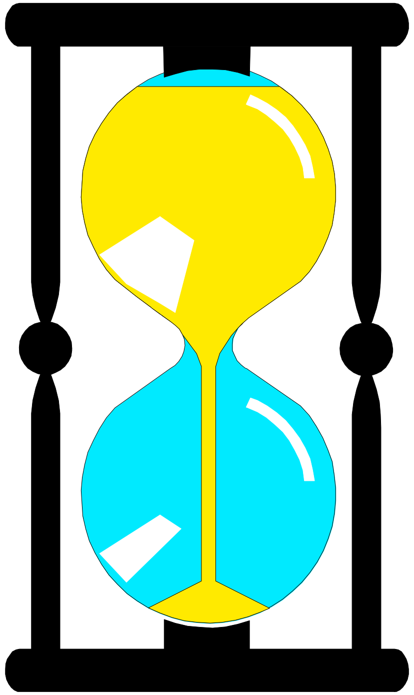


You have 60 seconds to write the declaration.

An Exercise

One solution:

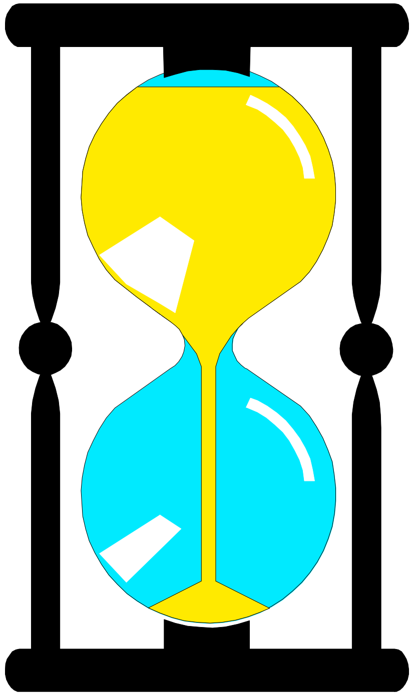
```
class bag
{
public:
     ...
private:
     int  data[20];
     int count;
};
```



An Exercise

A more flexible solution:

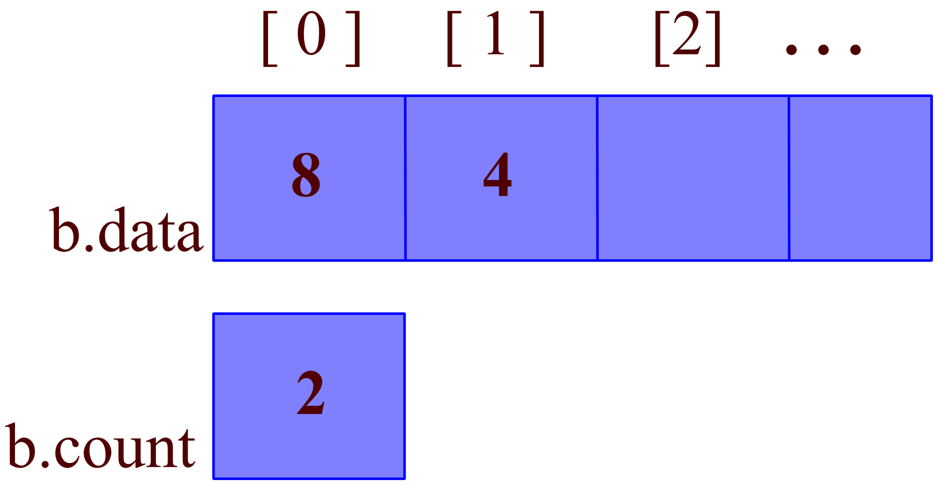
```
class bag
{
public:
    static const int CAPACITY = 20;
    ...
private:
    int  data[CAPACITY];
    int count;
};
```



An Example of Calling Insert

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

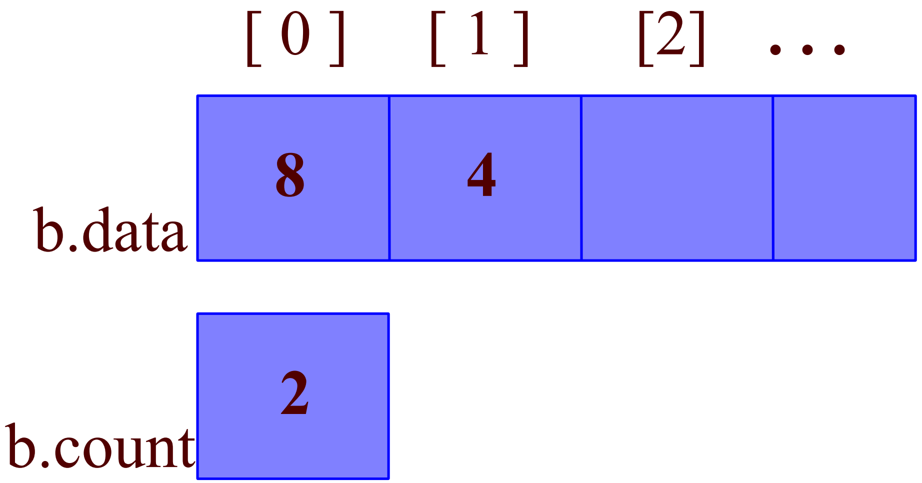
Before calling insert, we might have this bag b:



An Example of Calling Insert

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

We make a function call b.insert(17)

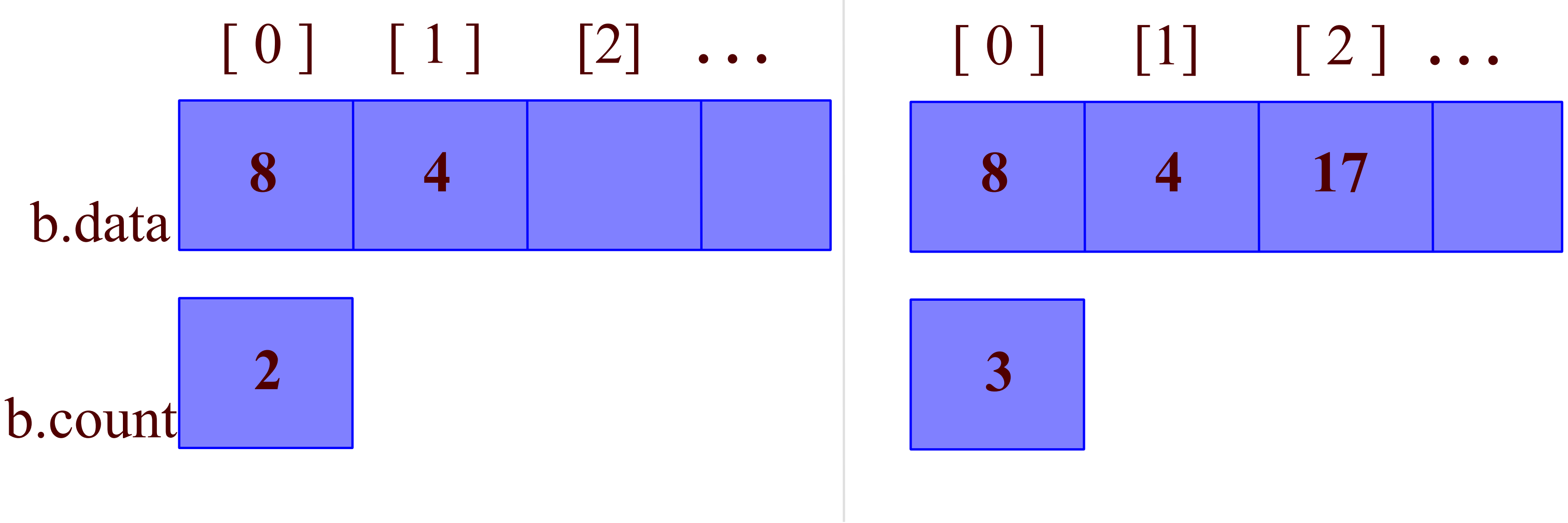


What values will be in b.data and b.count after the member function finishes ?

An Example of Calling Insert

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

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



[0]
[1]
[2]
. . .
b.data
8
4
b.count
2
[0]
[1]
[2]
. . .
8
4
17
3