Container Classes

The Bag Class - Part 1

Read Chapter 3

Abstraction

Abstraction

When I use a computer...

- I’m not thinking about every little detail that makes it work

I don’t care that:

- the processor is performing billions of operations per second...
- the operating system is simultaneously managing memory (cache, real, and virtual), giving the illusion of true concurrency, and enforcing security...
- the program I’m running was written in Ruby, which is executed by an interpreter written in C, which was compiled into machine code specific to my system...
- the little black console window is buffered and updates only when a certain amount of data has been sent or a specific character is seen...
- the image being sent to my monitor is being redrawn dozens of times per second...
- all to see “Hello world!” get printed to my screen. O_o

Abstraction:

thinking in terms of higher-level concepts
while ignoring the particular implementation details

Abstract Data Types

We humans are really good at abstracting

- we generally don't care <u>how</u> something works
- <u>as long as it does what it says it does</u>

This is what an abstract data type (ADT) is all about!

An abstract data type:

- provides a specific service or represents a specific thing
- exposes a public interface that allows us to interact with it
- completely <u>hides the details</u> of how it is implemented

Abstract Data Types

You've already used several ADTs:

- the string class
- stream classes (`cin`, `cout`, and file streams)

They perfectly hid their implementation details from you!

The string class:

- encapsulates character arrays
- dynamically resizes itself when needed, automatically allocating/deallocating memory
- overloads numerous operators to behave more naturally
- and adds all sorts of methods besides!

Abstract Data Types

You've already used several ADTs:

- the string class
- stream classes (`cin`, `cout`, and file streams)

They perfectly hid their implementation details from you!

The stream classes:

- are incredibly complicated implementations that know how to create, read, and write files (even over networks); abstract away differences between different operating systems; and somehow <u>just work</u>
- provide an incredibly easy-to-use interface, despite their actual complexity

Abstract Data Types

A good ADT is:

- <u>reusable</u>
    - think about how many times have you used strings or `cout` in a program!
- <u>reliable</u>
    - I doubt you've ever found a bug in the string or stream classes...
- <u>efficient</u>
    - strings and streams use highly efficient (fast) algorithms to "do their thing"
- <u>flexible</u>
    - think about output formatting and how much we can control about a stream's behavior...

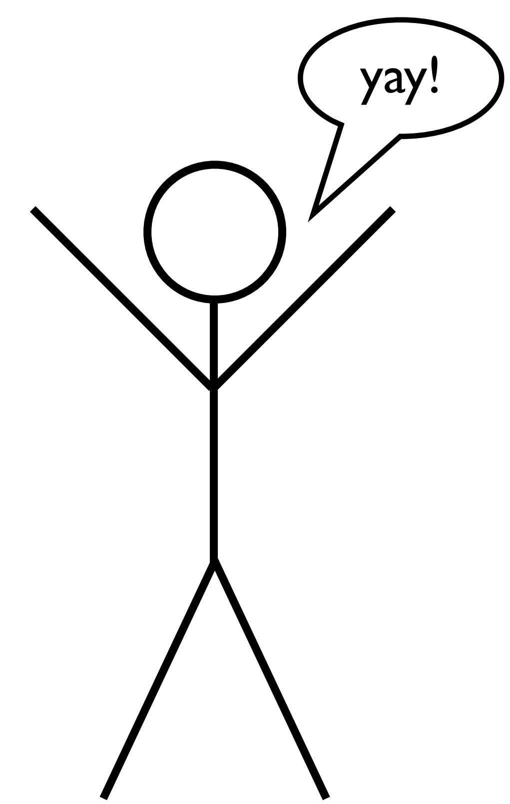
Abstract Data Types

Understanding ADTs (data structures):

- empowers you to write more efficient, robust, and maintainable code
- makes you more productive
- saves impoverished children from starving to death

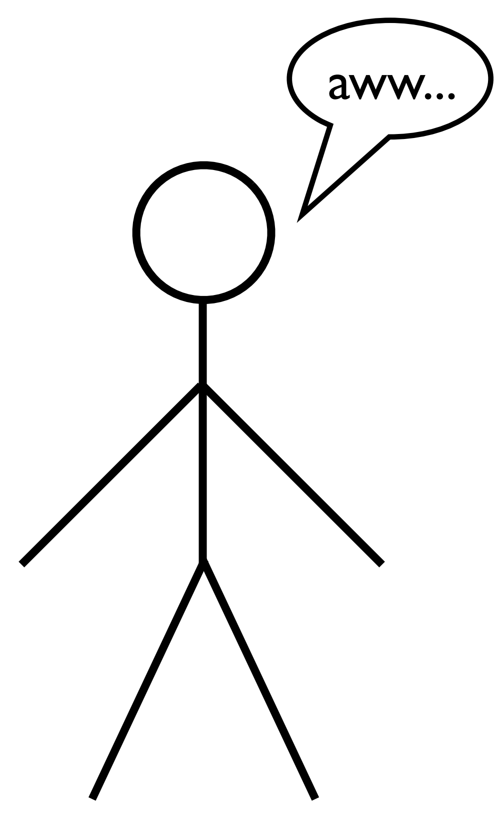
What we're going to do now:

- design and implement our own simple container classes (ADTs)
- learn additional features of C++ that make life easier
- save impoverished children



But before we begin...

(just wait a little while longer, impoverished children!)



Namespaces

A namespace:

- prevents name collisions (e.g., distinguishes your `Point` class from .Net's `Point` class)
- groups code (classes, constants, etc...) that serve a common purpose

Examples:

- the `std` namespace includes the classes and objects that are declared as part of standard C++ distributions
- the .NET environment contains many namespaces for drawing, networking, security, and many, many other things

We will use namespaces for all future projects in this class

- part of your grade depends on it =)

Namespaces

Syntax:

```
// creates a namespace called CS262
namespace CS262 {
    // this code is part of the CS262 namespace
}
```

This creates a `CS262` namespace

- any classes, constants, or functions declared inside the namespace will become part of the namespace
- this eliminates the possibility of name conflicts with code outside of the namespace

Namespaces

Example:

```
// Point.h header file
namespace CS262 {
    // a Point class, part of the CS262 namespace
    class Point {
        int x, y;
      public:
        Point(int x, int y);
    };
}
```

Namespaces

Implementing the Point constructor:

```
// Point.cpp implementation file
namespace CS262 {

    // implementation of Point constructor
    Point::Point(int X, int Y) { /* ... */ }

}
```

Notice:

- just like the class declaration, the implementation is inside a namespace block

Namespaces

Alternatively, we could use the scope resolution operator (::)

```
// Point.cpp implementation file



// implementation of Point constructor
CS262::Point::Point(int X, int Y) { /* ... */ }
```

namespace prefix

Notice:

- when prefixed with the namespace directly using the scope resolution operator, the implementation should not be inside a namespace block

Namespaces

To use the class, prefix the class name with its namespace:

```cpp
// main.cpp file
#include "Point.h"



int main() {
    // create a Point (notice the namespace prefix!)
    CS262::Point p1;

    return 0;
}
```

Namespaces

Alternatively, we could add a using directive:

```
// main.cpp file
#include "Point.h"
using namespace CS262;  // look familiar? =)

int main() {
    // no namespace prefix needed!
    Point p1;

    return 0;
}
```

Namespaces

We could make use of a specific item in the namespace like this:

```
// main.cpp file
#include "Point.h"
using CS262::Point; // uses only the Point class

int main() {
    // no namespace prefix needed!
    Point p1;

    return 0;
}
```

Namespaces

Try not to put using statements in header files

```
// Point.h header file
using namespace std; // BAD

namespace CS262 {
    string str;
}
```

If you do...

- you're forcing anyone who `#includes` your header file to be using the same things
- this might introduce name conflicts into their code—through no fault of their own!

Namespaces

Instead, use fully qualified names:

```
// Point.h header file



namespace CS262 {
    std::string str; // fully qualified type name
}
```

This achieves the same effect...

- but eliminates the risk of introducing name conflicts
- yes, it's a pain... but so is losing points on your assignments =)

Namespaces

...or put the using directive inside your namespace:

```
// Point.h header file




namespace CS262 {
    // okay - only affects this scope
    using namespace std;

    string str;
}
```

Namespaces

Namespaces can also be nested:

```
// creates a namespace called Mines
namespace Mines {
    namespace CS262 {
        // part of the Mines::CS262 namespace
        class Point { ... };
    }
}
```

To access `Point`, use both namespaces:

```
Mines::CS262::Point p1;
```

Macro Guards

You can include the same header file from multiple different files...

- however, C++ cannot see the same class declaration more than once (just like you can't declare the same variable more than once in any given scope)
- if it did, you would get a "duplicate class definition" compiler error

To prevent this issue, all header files you write should have this line:

```
#pragma once
```

Problem solved!

The book suggests a different macro

- don't use it

typedef

Syntax:

```
// declares an alias for the given data type
typedef data_type alias;
```

Example:

```
// makes Array4D an alias for int****
typedef int**** Array4D;

// the following statements are now equivalent:
int**** array_1;
Array4D array_2;
```

size_t

`size_t` is an integer data type that:

- can store only non-negative values
- has a range of values large enough to hold the size of any variable that can be declared on your machine (guaranteed by all C++ implementations)
- is defined in the `cstdlib` library as part of the `std` namespace

Any time you need to store the size of something in a variable...

- use `size_t` as the data type

static members

a static member:

- is shared by all objects of a class—there is only every a single copy of it
- conversely, each object gets its own copy of any non-static members
- only integer-type data can be initialized like you see below

Example:

```
class Triangle {
    public:
        // one constant value shared by all Triangles
        static const int NUM_SIDES = 3;
};
```

static members

For non-integer type static members, you have to do a bit more work:

```
class Circle {
    public:
        // constant shared by all Circle objects
        static const double PI; // no value here...
};
```

Then, in your implementation file (not inside any function):

```
// the name of the constant is prefixed with Circle::
const double Circle::PI = acos(-1);
```

No static keyword here!

name prefixed w/ class

The Bag Class:

Specification

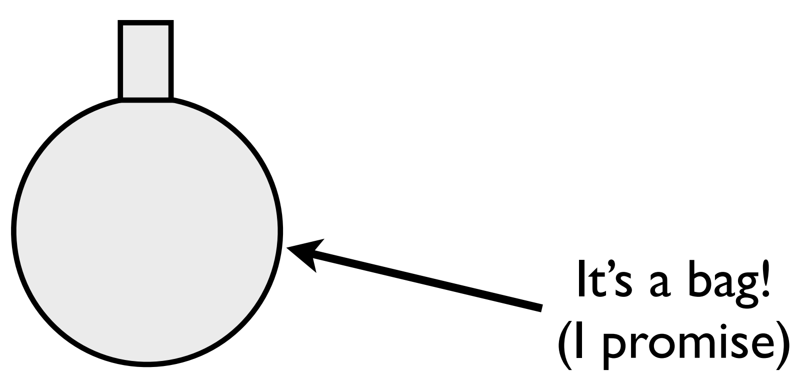
The Bag Class—Specification

Container class:

- a class where each object contains a collection of items

Our Bag will be container class:

- it will hold a collection of items that all share the same type
- items in the bag will have no apparent sequence or order
- the same values can appear multiple times in a bag
- each bag will have the same fixed capacity (max number of items)
- it must be easy to change the type of data that bags hold (`typedef`!)



The Bag Class—Specification

Type definitions and member constants:

```
// data type of items in the bag
typedef _________ value_type;

// value_type must be a built-in type, or support:
//  - instantiation via a default constructor
//  - assignment operator   (x = y)
//  - equality operator     (x == y)
//  - non-equality operator (x != y)
```

The Bag Class—Specification

Type definitions and member constants:

```
// data type of variables that track a bag's size
typedef _________ size_type;

// the max number of items a bag can hold
static const size_type CAPACITY = ___;
```

The Bag Class—Specification

Constructors:

```
// creates an empty bag
bag();

// postcondition:
//   the bag has been initialized as an empty bag
```

The Bag Class—Specification

Value semantics:

```
// bag objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

The Bag Class—Specification

Modification member functions:

```
// removes all copies of @target from the bag
// returns the number of elements removed
size_type erase(const value_type& target);

// postcondition:
//   all copies of target have been removed from the bag
//   return value is the number of elements removed
```

The Bag Class—Specification

Modification member functions:

```
// removes a single copy of @target from the bag
// returns true if a value was removed; false otherwise
bool erase_one(const value_type& target);

// postcondition:
//   if @target was in the bag, then one copy is removed
//   otherwise, the bag remains unchanged
//   return value is true if a value was removed,
//     or false otherwise
```

The Bag Class—Specification

Modification member functions:

```
// inserts a new copy of @entry into the bag
void insert(const value_type& entry);

// precondition:
//   size() < CAPACITY
// postcondition:
//   a new copy of @entry has been added to the bag
```

The Bag Class—Specification

Modification member functions:

```
// inserts a copy of each item in @addend into the bag
void operator +=(const bag& addend);

// precondition:
//   size() + addend.size() < CAPACITY
// postcondition:
//   each item in addend has been added to the bag
```

The Bag Class—Specification

Constant member functions:

```
// returns the total number of items in the bag
size_type size() const;

// postcondition:
//   return value is the number of items in the bag
```

The Bag Class—Specification

Constant member functions:

```
// returns the total number of occurrences of @target
size_type count(const value_type& target) const;

// postcondition:
//   return value is the number of times @target occurs
//     in the bag
```

The Bag Class—Specification

Non-member functions:

```
// returns a new bag that is the union of @b1 and @b2
bag operator +(const bag& b1, const bag& b2);

// precondition:
//   b1.size() + b2.size() < CAPACITY
// postcondition:
//   the bag returned is the union of b1 and b2
```

The Bag Class:

Implementation

The Bag Class—Implementation

Starting the header file:

```
// bag.h header file
// specification documentation
#pragma once
#include <cstdlib>

namespace CS262 {
    class bag {
        // bag class declaration
    };
}
```

The Bag Class—Implementation

Starting the implementation file:

```
// bag.cpp implementation file
// INVARIANT (coming soon)
#include "bag.h"
#include <algorithm>
#include <cassert>
using namespace std;

namespace CS262 {
    // member implementations
}
```

The Bag Class—Implementation

Type definitions and member constants:

```
class bag {
    public:
        // typedefs and member constants
        typedef int value_type;
        typedef std::size_t size_type;
        static const size_type CAPACITY = 30;

    private:
        value_type data[CAPACITY];
        size_type used;
};
```

The Bag Class—Implementation

Notice the private section of the class:

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

Our implementation will store items in an array

- the static array we're using will have a fixed size, called `CAPACITY`
- the number of elements in the bag will be tracked by the `used` member variable
- the data type aliased as value_type must have a default constructor, since arrays use the default constructor to initialize objects

The Bag Class—Implementation

How we use the member variables are used is the class <u>invariant</u>

- this should be clearly documented in your class implementation file
- it does not belong in the header file, since it represents the implementation details
- anyone implementing the class needs to know exactly how the class' member variables are used to represent the object

Document invariant in implementation file:

```
// bag.cpp implementation file
// INVARIANT for bag class:
// 1. the number of items in the bag is stored in used
// 2. for an empty bag, we do not care what is in the
//       array; for a non-empty bag, items are in
//       data[0] - data[used-1]
```

The Bag Class—Implementation

Constructor:

```
class bag {
    public:
        // typedefs and member constants

        // default constructor creates an empty bag
        bag() : used(0) { }

    private:
        value_type data[CAPACITY];
        size_type used;
};
```

The Bag Class—Implementation

Value semantics:

```
// bag objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

The implementation for this is easy...

- we will use the default assignment operator and copy constructor provided by C++
- this is fine, since both will simply copy any member variables (`data` and `used`)
- exactly what we want!

The Bag Class—Implementation

The size member function:

```
class bag {
    public:
        // other code

        // returns the total number of items in the bag
        size_type size() const { return used; }

    private:
        value_type data[CAPACITY];
        size_type used;
};
```

The Bag Class—Implementation

The count member function:

```
// returns the total number of occurrences of @t
bag::size_type bag::count(const value_type& t) const {
    size_type answer = 0;

    for (size_type i = 0; i < used; i++)
        if (data[i] == t)
            answer++;

    return answer;
}
```

The Bag Class—Implementation

The insert member function:

```
// inserts a new copy of @entry into the bag
void bag::insert(const value_type& entry) {
    assert(size() < CAPACITY);
    data[used] = entry;
    used++;
}
```

The Bag Class—Implementation

The erase_one member function...

If the target is found, we need to:

- remove the item from the array (without leaving a hole)
- decrement used

Here's a clever way to do it:

- find the first occurrence of the target value
- swap the value with the final item in the bag
- decrement used

The Bag Class—Implementation

The erase_one member function:

```
// removes a single copy of @target from the bag
bool bag::erase_one(const value_type& target) {
    size_type i = 0;

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

    if (i == used) return false;

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

The Bag Class—Implementation

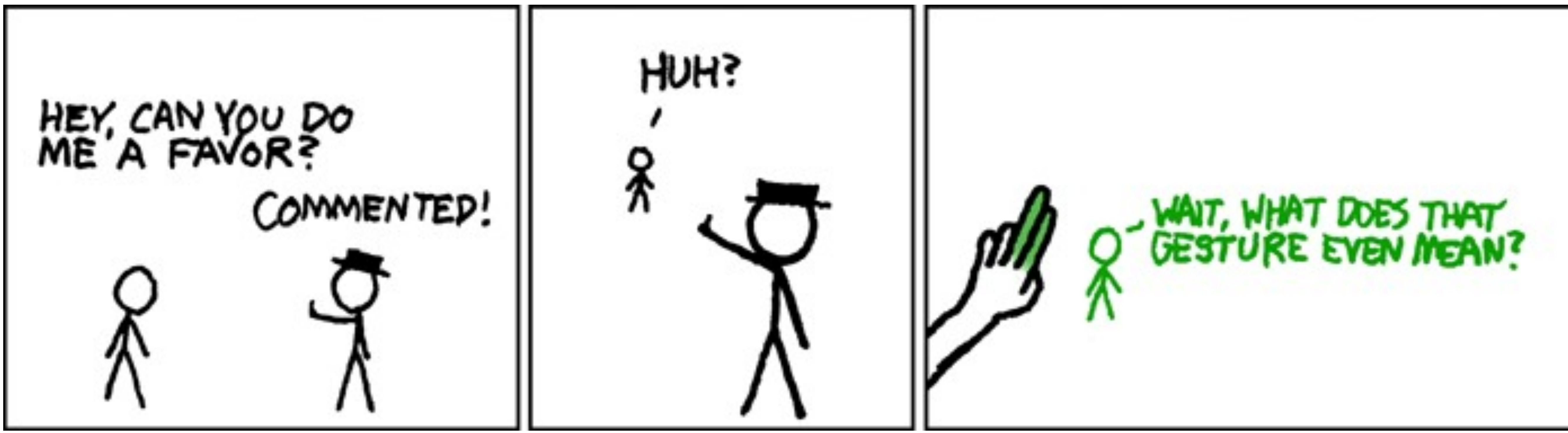
How would you implement the erase member function?

```
// removes all copies of @target from the bag
// returns the number of elements removed
size_type erase(const value_type& target);

// postcondition:
//   all copies of target have been removed from the bag
//   return value is the number of elements removed
```

The Bag Class—Implementation

How would you implement the operator += member function?

```
// inserts a copy of each item in @addend into the bag
void operator +=(const bag& addend);

// precondition:
//   size() + addend.size() < CAPACITY
// postcondition:
//   each item in addend has been added to the bag
```


HEY, CAN YOU DO ME A FAVOR?
COMMENTED!
HUH?
WAIT, WHAT DOES THAT GESTURE EVEN MEAN?