Container Classes

The Bag Class with Dynamic Memory

A Dynamic Bag

Previously we made a bag that could hold a fixed number of items...

- lame, huh?

Now, we're going to make it dynamic

- the bag should transparently resize itself if it isn't large enough to support a given operation
- it is, in effect, a magic bag of holding!

The users of our bag will rejoice!

- how they use the class won't change...
- but it will be able to hold as many items as they need it to

Updated Specification

Updated 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)
```

Updated Specification

Type definitions and member constants:

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

Updated Specification

Type definitions and member constants: `// CHANGED!`

```
// the default number of items a bag can hold
// used by the default constructor
static const size_type DEFAULT_CAPACITY = ___;
```

Updated Specification

Constructors: // CHANGED!

```
// creates an empty bag with the given capacity
bag(size_type initial_capacity = DEFAULT_CAPACITY);

// postcondition:
//   the bag is empty with the given initial capacity
//   the insert function will work efficiently (without
//     allocating new memory) until this capacity is
//     reached
```

Updated 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
```

Updated 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
```

Updated Specification

Modification member functions:

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

// postcondition:
//   a new copy of @entry has been added to the bag
```

Updated Specification

Modification member functions: // NEW!

```
// allocates a new data array of the requested size
void reserve(size_type new_size);

// postcondition:
//   The bag's current capacity is changed to new_size
//     (but not less than the number of items already in
//     the bag).
//   The insert function will work efficiently (without
//     allocating new memory) until the new capacity is
//     reached
```

Updated Specification

Modification member functions:

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

// postcondition:
//   each item in addend has been added to the bag
```

Updated 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
```

Updated 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
```

Updated Specification

Non-member functions:

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

// postcondition:
//   the bag returned is the union of b1 and b2
```

Updated Specification

Value semantics:

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

Updated Specification

Dynamic memory usage: `// NEW!`

```
// If there is insufficient dynamic memory, then the
// following functions throw bad_alloc:
//   constructors
//   reserve
//   insert
//   operator +=
//   operator +
```

Updated Specification

Not much changed, did it?

- the specification is for the <u>users</u> of the class; the main change they see is that the bag is no longer limited to a fixed size

The updated specification did hint at a few implementation details...

- which functions use dynamic memory were listed
- two methods (the constructor and `reserve`) directly deal with the underlying capacity of the dynamically allocated array

Ideally, how a class is implemented should be completely transparent

- however, the added details still address how the class should be used
- good programmers will want to be able to handle errors appropriately and to use the class efficiently

Updated Implementation

Updated Implementation

Starting the header file:

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

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

Updated 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
}
```

Updated Implementation

Type definitions and member constants:

```
class bag {
    public:
        typedef int value_type;
        typedef std::size_t size_type;
        static const size_type DEFAULT_CAPACITY = 30;

    private:
        value_type* data;
        size_type capacity;
        size_type used;
};
```

Updated Implementation

Here's the updated private section:

```
private:
    value_type* data;    // pointer to dynamic array
    size_type capacity; // current capacity of array
    size_type used;      // number of items in array
```

Compare that with the static version's data members:

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

Updated Implementation

Here's the updated private section:

```
private:
    value_type* data;   // pointer to dynamic array
    size_type capacity; // current capacity of array
    size_type used;     // number of items in array
```

Our implementation still uses an array...

- unlike before (when the array was static), we're now using a dynamically allocated array; this will allow us to resize the bag whenever needed by allocating a new array
- accordingly, we need to keep track of the current size of the array; this is stored in the member variable `capacity`
- again, the number of elements in the bag will be tracked by the `used` member variable

Updated Implementation

Document invariant in implementation file:

```
// bag.cpp implementation file
// INVARIANT for bag class:
// 1. the number of items in the bag is stored in the
//    member variable used
// 2. The actual items of the bag are stored in a
//    partially filled array. The array is a dynamic
//    array, pointed to by the member variable data
// 3. the size of the dynamic array is in the member
//    variable capacity
```

Updated Implementation

The constructor prototype:

```
// creates an empty bag with the given capacity
bag(size_type initial_capacity = DEFAULT_CAPACITY);
```

The user can now (optionally) set the initial capacity of the bag:

```
bag b1;        // a bag with the default capacity
bag b2(1000); // a bag with a capacity of 1000
```

Try writing this function

Updated Implementation

The constructor implementation:

```
// creates an empty bag with the given capacity
bag::bag(size_type initial_capacity) {
    data = new value_type[initial_capacity];
    capacity = initial_capacity;
    used = 0;
}
```

Updated Implementation

Another constructor implementation (using an initialization list):

```
// creates an empty bag with the given capacity
bag::bag(size_type initial_capacity) :
    capacity(initial_capacity), used(0)
{
    data = new value_type[initial_capacity];
}
```

Updated Implementation

The reserve function prototype:

```
// allocates a new data array of the requested size
void reserve(size_type new_size);
```

The user can request a resize via the reserve method

- it should dynamically allocate a new array of the requested size
- if the new capacity is the same as the current capacity, it doesn't need to do anything
- likewise, if the requested capacity is less than the current capacity, it cannot result in a loss of data (cannot be smaller than used)

Try writing this function

Updated Implementation

The reserve function implementation:

```
void bag::reserve(size_type new_size) {
    if (new_size == capacity) return;
    if (new_size < used) new_size = used;

    value_type* new_array = new value_type[new_size];
    copy(data, data + used, new_array);

    delete [] data;
    data = new_array;
    capacity = new_size;
}
```

Updated Implementation

The erase function prototype:

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

Does the implementation of this function need to change?

- Nope!

Updated Implementation

The erase function implementation (no change):

```
// removes all copies of @target from the bag
bag::size_type bag::erase(const value_type& target) {
    size_type i = 0, num_erased = 0;
    while (i < used) {
        if (data[i] == target) {
            data[i] = data[--used];
            num_erased++;
        } else i++;
    }
    return num_erased;
}
```

Updated Implementation

The erase_one function prototype:

```
// removes a single copy of @target from the bag
bool erase_one(const value_type& target);
```

Does the implementation of this function need to change?

- Nope!

Updated Implementation

The erase_one function implementation (no change):

```
// 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;
}
```

Updated Implementation

The insert function prototype:

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

Does the implementation of this function need to change?

- Yes!
- insert must now allocate additional space, if necessary
- this should replace the assertion from the earlier implementation

Try writing this function

Updated Implementation

The insert function implementation (updated):

```
// inserts a new copy of @entry into the bag
void bag::insert(const value_type& entry) {
    // increase the bag's capacity, if necessary
    if (used == capacity) {
        reserve(2 * used);
    }

    data[used++] = entry;
}
```

Updated Implementation

The operator += function prototype:

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

Does the implementation of this function need to change?

- Yes!
- must now allocate additional space, if necessary
- this should replace the assertion from the earlier implementation

Try writing this function

Updated Implementation

The operator += function implementation (updated):

```
// inserts a copy of each item in @b into the bag
void operator +=(const bag& b) {
    // increase the bag's capacity, if necessary
    if (used + b.used > capacity) {
        reserve(used + b.used);
    }

    // copies all items in b.data to the end of data
    copy(b.data, b.data + b.used, data + used);
    used += b.used;
}
```

Updated Implementation

The size function:

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

Does the implementation of this function need to change?

- Yes! It's much too simple... Rabble rabble rabble!!!

Updated Implementation

The count function prototype:

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

Does the implementation of this function need to change?

- Nope!

Updated Implementation

The count function implementation (no change):

```
// 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;
}
```

Updated Implementation

The operator + function prototype:

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

Does the implementation of this function need to change?

- Yes!
- it must create a bag big enough to hold the items from both bags
- this can be done using the constructor's optional initial_capacity argument

Try writing this function

Updated Implementation

The operator + function implementation (updated):

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

    union += b1;
    union += b2;

    return union;
}
```

Updated Implementation

The bag class now uses dynamic memory...

- this means the automatic implementations of the copy constructor, assignment operator, and destructor are no longer sufficient

Why weren't these part of our specification?

- we specified copying as okay as part of the value semantics of our class, and copying and assignment are always done exactly the same way...
- users never explicitly call the destructor

Updated Implementation

The copy constructor prototype:

```
// creates a new bag as a copy of @source
bag(const bag& source);
```

The copy constructor must:

- allocate a new array of the same size as that in the source bag
- copy all the items from the source array to the new array
- update `used` and `capacity`

Try writing this function

Updated Implementation

The copy constructor implementation:

```
// creates a new bag as a copy of @source
bag::bag(const bag& source) {
    data = new value_type[source.capacity];

    used = source.used;
    capacity = source.capacity;

    copy(source.data, source.data + used, data);
}
```

Updated Implementation

The assignment operator prototype:

```
// assigns this bag as a copy of @source
bag& operator =(const bag& source);
```

The assignment operator must:

- check for self-assignment
- allocate a new array of the same size as that in the source bag, if necessary
- copy all the items from the source array to the calling object bag
- update `used` and `capacity`
- ideally, it should also return the calling object by reference (for assignment chaining)

Try writing this function

Updated Implementation

The assignment operator implementation:

```
bag& bag::operator =(const bag& source) {
    if (this == &source) return *this;

    if (capacity != source.capacity) {
        delete [] data;
        data = new value_type[source.capacity];
        capacity = source.capacity;
    }

    used = source.used;
    copy(source.data, source.data + used, data);
    return *this;
}
```

Updated Implementation

The destructor prototype:

```
// frees the memory used by this bag
~bag();
```

The destructor must:

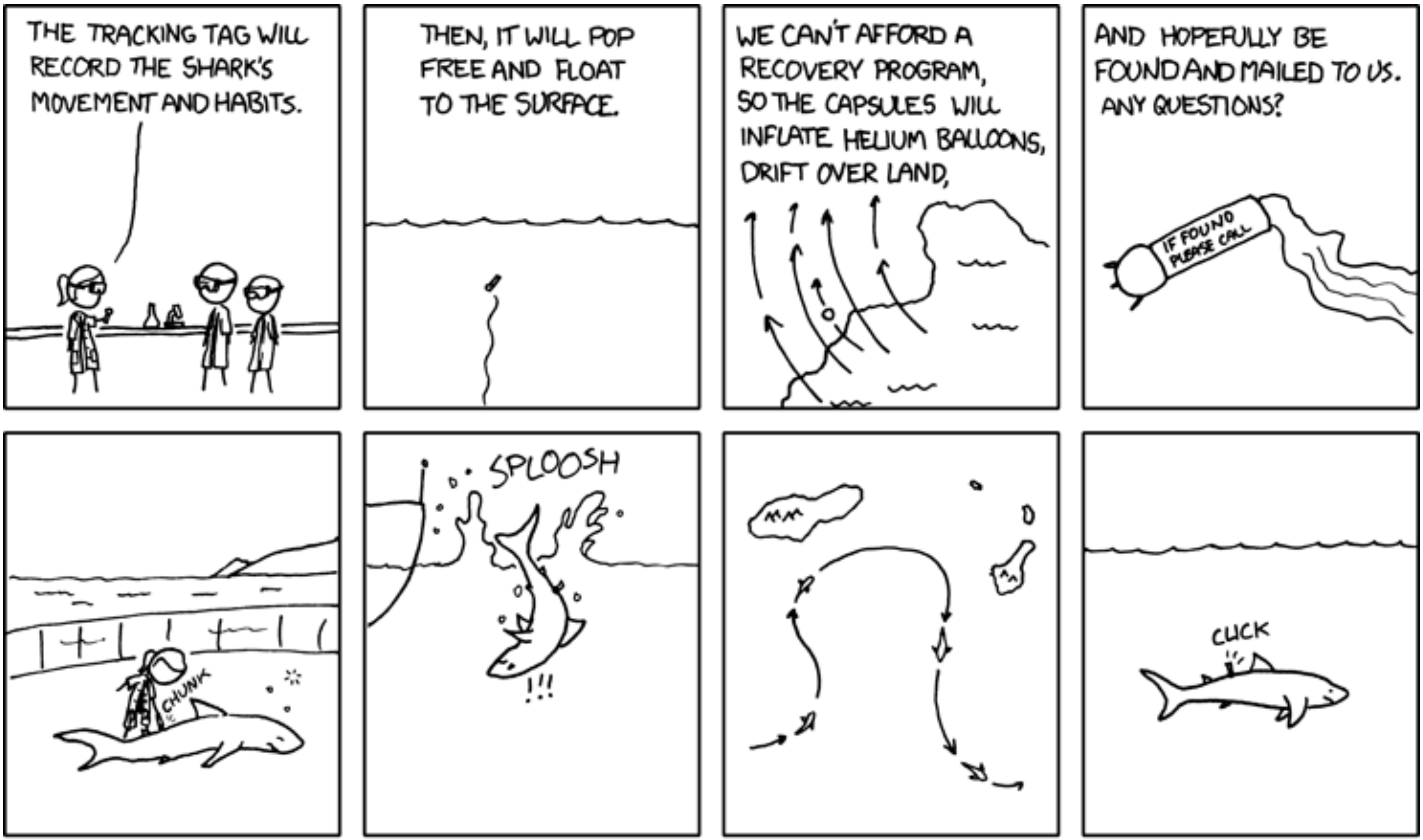
- deallocate any dynamically allocated memory used by the bag

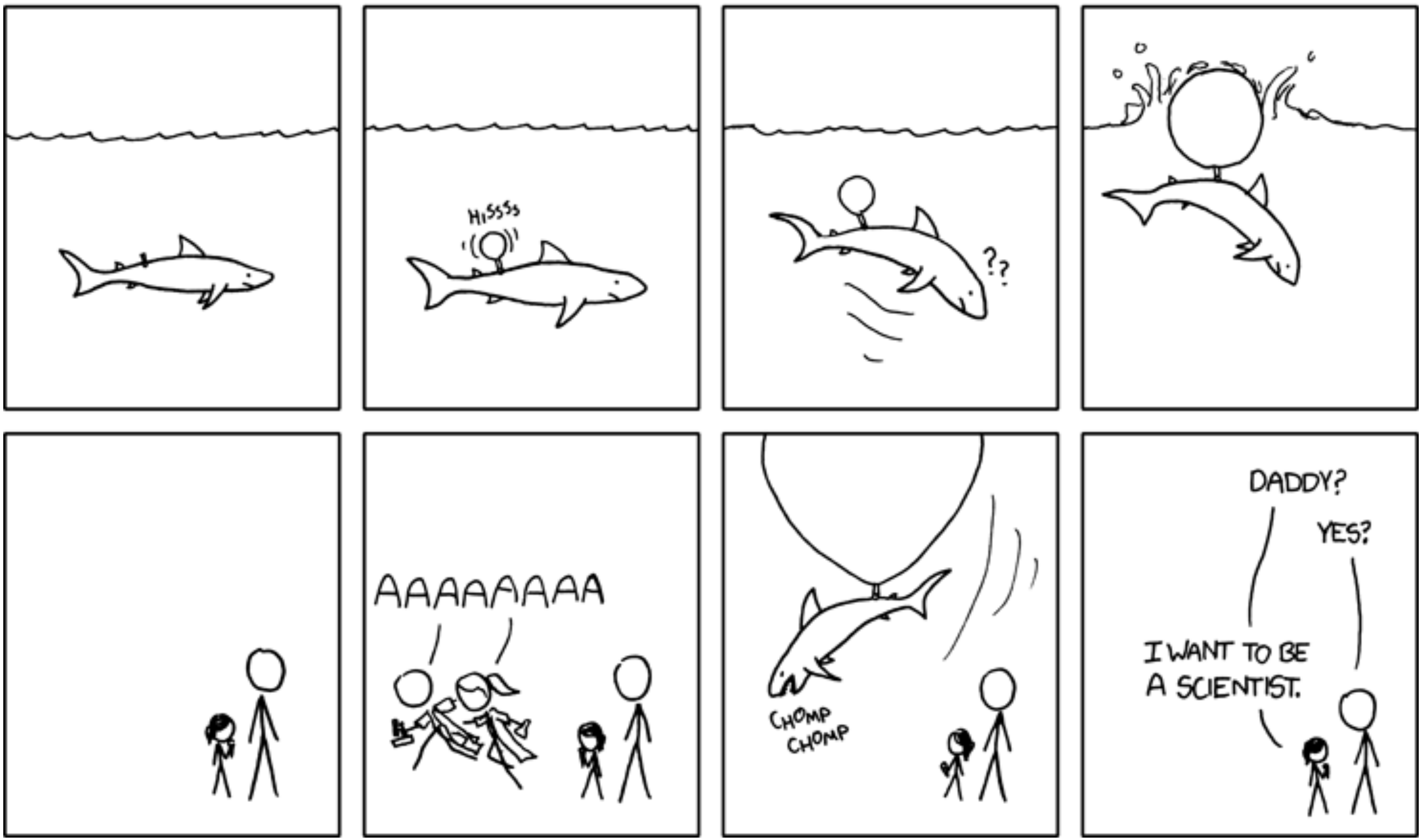
Try writing this function

Updated Implementation

The destructor implementation:

```
// frees the memory used by this bag
bag::~bag() {
    delete [] data;
}
```


THE TRACKING TAG WILL RECORD THE SHARK'S MOVEMENT AND HABITS.
THEN, IT WILL POP FREE AND FLOAT TO THE SURFACE.
WE CAN'T AFFORD A RECOVERY PROGRAM, SO THE CAPSULES WILL INFLATE HELIUM BALLOONS, DRIFT OVER LAND,
AND HOPEFULLY BE FOUND AND MAILED TO US. ANY QUESTIONS?
IF FOUND PLEASE CALL
CHUNK
SPLOOSH
!!!
CLICK


HISSSS
??
AAAAAAAA
CHOMP
CHOMP
DADDY?
YES?
I WANT TO BE
A SCIENTIST.