

Exception Safety

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, March 28, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview

Sequences and Their Implementations

Major Topics

- ✓ • Data Abstraction
- ✓ • Sequence Data
- ✓ • Interfaces to Data
- ✓ • Data Structure Implementation
 - Exception Safety
 - Allocation & Efficiency
 - Generic Containers
 - Node-Based Structures
 - Linked Lists
 - Sequences in Practice

Review

Sequence Data — ADT Sequence

ADT **Sequence**

- **Data**
 - An ordered sequence of values, all the same type, indexed by 0, ..., size-1.
- **Operations**
 - **CreateEmpty**
 - Creates empty Sequence (with size 0, i.e., no data).
 - **CreateSized**
 - Given a size, create a Sequence with that size.
 - **Destroy**
 - Destroys a Sequence.
 - **Copy**
 - Make a copy of a given Sequence.
 - **LookUpByIndex**
 - Given a valid index, returns Sequence item in modifiable form.
 - **Size**
 - Returns size of Sequence.
 - **Empty**
 - Returns whether the Sequence is empty, that is, has size zero.
 - **Sort**
 - Sort a Sequence, using some given comparison function.
 - **Resize**
 - Changes size of Sequence. Data for indices 0, ..., $\min(\text{old size}, \text{new size})-1$ remains identical.
 - **InsertByIter**
 - Given an iterator (or pointer?) and an item, insert the item at the specified position.
 - **RemoveByIter**
 - Given an iterator, remove the item at that position.
 - **InsertBeg**
 - Given an item, insert it at the beginning.
 - **RemoveBeg**
 - Remove the first item.
 - **InsertEnd**
 - Like insertBeg, but at the end.
 - **RemoveEnd**
 - Like removeBeg, but at the end.
 - **Splice**
 - Move a contiguous subsequence from one Sequence to another.
 - **Traverse**
 - Performs some operation on every item in the Sequence, in order.
 - **Swap**
 - Swap the values of two given Sequences.

Review

Interfaces to Data — Characteristics of Good Interfaces

An interface should be **complete**.

- All relevant operations should be *possible*.
 - But what is “relevant”? Sometimes we place restrictions, as we will see with ADTs Stack and Queue.

We often strive for interfaces that are **minimal**.

- Avoid unnecessary functionality.

An interface should be **convenient**.

- Avoid making the interface a pain to use.

We want to facilitate **efficiency**.

- Allow the data to be dealt with efficiently.

We often want our interface to be **generic**.

- Avoid restricting possible implementations and internal data types.

- Some of the above goals can be at odds with each other. Designing a good, useful interface can be difficult.

These two often pull in opposite directions.

These two *can* pull in opposite directions.

Review

Interfaces to Data — An Interface for a Sequence [1/2]

Use iterators to handle positions, traversing.

ADT Operations

- CreateEmpty
 - Default ctor.
- CreateSized
 - Ctor given size.
- Destroy
 - Dctor.
- Copy
 - Copy ctor & copy assignment.
- LookUpByIndex
 - Bracket operator.
- Size
 - Member function `size`.
- Empty
 - Member function `empty`.
- Sort
 - Handle externally, using iterators. Use iterator-returning member functions `begin` and `end`.
- Resize
 - Member function `resize`.
- InsertByIter, InsertBeg, InsertEnd
 - Member function `insert` does InsertByIter.
 - Use in conjunction with iterator-returning functions to do InsertBeg, InsertEnd.
- RemoveByIter, RemoveBeg, RemoveEnd
 - As above, using `remove`.
- Splice
 - Call `resize`, then copy using `op[]`.
- Traverse
 - Use iterator-returning member functions `begin` and `end`.
- Swap
 - Member function `swap`.

Review

Interfaces to Data — An Interface for a Sequence [2/2]

Ctors & Dctor

- Default ctor
- Ctor given size
- Copy ctor
- Dctor

Member Operators

- Copy assignment
- Bracket

Global Operators

- *None.*

Associated Global Functions

- *None.*

Named Public Member Functions

- **size**
- **empty**
- **begin**
- **end**
- **resize**
- **insert**
- **remove**
- **swap**

Review

Data Structure Implementation [1/2]

Call our class "Sequence".

What type should a data item be?

- Use `int` for the value type (for now).
 - You will make it generic in Assignment 5.

What type should the size of a Sequence be?

- Use `std::size_t`.

How should we store the data?

- Use a dynamically allocated array of `ints`.
- Note: We could have used a separate RAII class, like `IntArray`.

How should we implement the iterators?

- Use pointers (`int *`, `const int *`).

Have member types, as in STL containers: `value_type`, `size_type`, `iterator`, `const_iterator`.

- This allows us to easily tell what a value is for.
- Also, we can easily change (say) the value type.

Review

Data Structure Implementation [2/2]

What data members should class `sequence` have?

- Size of the array: `size_type size_;`
- Pointer to the array: `value_type * data_;`

Note: There are serious (but non-obvious) problems in this design, as we will see.

What class invariants should it have?

- Member "`size_`" should be nonnegative.
- Member "`data_`" should point to an `int` array, allocated with `new []`, owned by `*this`, holding (at least) `size_ ints`.

What should `operator[]` return? Should it be `const` or not?

- We need two versions: non-`const` and `const`.
- The non-`const` version returns `value_type &`.
- The `const` version returns `const value_type &`.

What should `begin`, `end` return? Should they be `const` or not?

- As with `operator[]`, we need two versions.
- Non-`const` versions return `iterator`, `const` versions return `const_iterator`.

What about the Big Three? Can we use silently written functions?

- No. We are directly managing an owned resource.

Exception Safety

Review — Error Handling

An **error condition** (or “error”) is a condition occurring during runtime that cannot be handled by the normal flow of execution.

- Not the same as a bug.
- Example: Could not read file.

Three ways of dealing with an error condition:

- Prevention
 - Require the client (via a precondition?) to prevent an error condition.
- Containment
 - Fix the problem ourselves.
- **Signal the Client**
 - Rule of thumb: When we cannot fulfill our postconditions.

Three ways to signal an error condition to the client:

- Returning an error code.
- Setting a flag to be checked by a separate error-checking function.
- Throwing an exception.

Exception Safety

Review — Introduction to Exceptions [1/4]

Exceptions are objects that are “**thrown**”, generally to signal error conditions.

- We **catch** exceptions using a `try ... catch` construction.
- A **throw** causes repeated backing out of blocks & functions, until a matching **catch** is found.
- An uncaught exception terminates the program.

```
Foo * makeAFoo() // throw(std::bad_alloc)
{
    return new Foo(2, 3);
}

void myFunc() // throw()
{
    Foo * p;
    try {
        p = makeAFoo();
    }
    catch (std::bad_alloc & e) {
        allocationSuccessful = false;
        cout << "Oops! Message: " << e.what() << endl;
    }
    [more stuff here]
}
```

Exception Safety

Review — Introduction to Exceptions [2/4]

We can throw our own exceptions, using “`throw`”.

```
class Foo {
public:
    int & operator[](int index) // May throw std::range_error
    {
        if (index < 0 || index >= arraySize)
            throw std::range_error("Foo: index out of range");
        return theArray[index];
    }
private:
    int * theArray;
    std::size_t arraySize;
}
```

We do not do this very much. And we only do it when we need to signal the client that an error condition has occurred.

Exception Safety

Review — Introduction to Exceptions [3/4]

We can catch **all** exceptions, using "...".

- In this case, we do not get to look at the exception, since we do not know what type it is.

```
try {  
    myFunc4(17);  
}  
catch (...) {  
    fixThingsUp();  
    throw;  
}
```

- Inside any `catch` block, we can **re-throw the same exception** using `throw` with no parameters.

Exception Safety

Review — Introduction to Exceptions [4/4]

The following can throw in C++:

- “`throw`” throws.
- “`new`” may throw `std::bad_alloc` if it cannot allocate.
 - There is a non-throwing version of `new`. See the applicable doc’s.
- A function that (1) calls a function that throws, and (2) does not catch the exception, will throw.
- Functions written by others may throw. See their doc’s.

The following do not throw:

- Built-in operations on built-in types.
 - Including the built-in `operator[]`.
- Deallocation done by the built-in version of “`delete`”.
 - Note: “`delete`” also calls destructors. These can throw.
- C++ Standard I/O Libraries (default behavior)
 - You *can* tell standard file streams to throw when an error occurs. However, they are non-throwing by default.

If a destructor is called between a throw and a catch, and that destructor throws, then the program terminates.

- Therefore, **destructors should not throw.**

Exception Safety

Introduction

When an error propagates to the caller, it is important that data are left in a usable state. In addition, we would like to know something about that state. It is also easy to get resource leaks in such situations; we wish to avoid these.

- These issues are collectively referred to as “**safety**”.

There are a number of commonly used safety levels.

- These are stated in the form of “guarantees” that a function makes.
- We usually think of them in terms of how they relate to member functions and class invariants, but they are applicable to any function.
- We will talk about these guarantees as they related to error handling via exceptions. However, **most of these ideas apply to any kind of error signaling technique.**

In this class, we will adopt the convention that a function throws when it cannot satisfy its postconditions.

- Thus, a function must either satisfy its postconditions, or else throw and satisfy its safety guarantee.

Exception Safety

The Three Standard Guarantees

Basic Guarantee

- Data remain in a usable state, and resources are never leaked, even in the presence of exceptions.

Strong Guarantee

- If the operation throws an exception, then it makes no changes that are visible to the client.

No-Throw Guarantee

- The operation never throws an exception.

Notes

- These guarantees were first set out by Dave Abrahams in the mid-1990s.
- Each guarantee includes the previous one.
- The Basic Guarantee is the minimum standard for all code.
- The Strong Guarantee is the one we generally prefer.
- The No-Throw Guarantee is required in some special situations.

Exception Safety

The Three Standard Guarantees — Basic Guarantee

Data remain in a usable state, and resources are never leaked, even in the presence of exceptions.

- When a member function throws, an object may end up in an **unknown** state, but it must be a **valid** state, with invariants maintained.

This is minimum standard that we expect well-written code to meet.

- What happens if this standard is not met, and an exception is thrown?

Exception Safety

The Three Standard Guarantees — Strong Guarantee

If the operation throws an exception, then it makes no changes that are visible to the client.

- Changes can be made, but the client must not see them.
- In practice, we exempt things like logging from these requirements.
- Generally, any work that has been done, must be undone.
- Thus, this is also called **commit or roll-back semantics**.

We like this level of safety, and we write code that meets it whenever it is reasonable to do so.

- But sometimes it is not reasonable, often due to efficiency concerns.

Exception Safety

The Three Standard Guarantees — No-Throw Guarantee

The operation never throws an exception.

- This is also known as the “No-Fail Guarantee”.

This is the **highest** level of exception safety, but it is not necessarily the **best** level.

- Exceptions are not “bad”. They are a tool that can help us deal with problematic situations. If we make the No-Throw Guarantee, then we have prohibited ourselves from using this tool.
- The No-Throw Guarantee does not say “errors do not occur”; rather, it says, “if an error occurs, then we are not allowed to signal the client; we must fix it ourselves”.
- Sometimes it is important to make the No-Throw Guarantee, often in situations in which we are “finishing something”.
 - One such situation: destructors.
 - We will discuss another situation shortly.

Exception Safety

Writing Exception-Safe Code — Ideas

To make sure code is exception-safe:

- Look at every place an exception might be thrown.
- For each, make sure that, if an exception is thrown, either
 - we terminate normally and meet our postconditions, or
 - we throw and meet our guarantees.

That's a lot of work, but ... modularity helps!

- Once we can certify a function as exception-safe, we can use it as such without re-examining it.

A bad design can force us to be unsafe.

- Thus, good design is part of exception safety.
- In particular, an often helpful idea is that **everything has exactly one purpose**. Code that follows this principle is **cohesive**.
 - Suppose that a function has two things to do, and the second thing produces an error.
 - Suppose that the second thing, above, is when the function returns a value.
 - Thus, the rule: A non-const member function should not return an object by value.