Exception Safety

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, October 16, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu
© 2005–2020 Glenn G. Chappell
Some material contributed by Chris Hartman

Review

Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

Unit Overview
Data Handling & Sequences

Major Topics

- ✓ Data abstraction
- ✓ Introduction to Sequences
- ✓ Interface for a smart array
- ✓ Basic array implementation
- Exception safety
- Allocation & efficiency
- Generic containers

Smart Arrays

- Node-based structures
- More on Linked Lists

Linked Lists

- Sequences in the C++ STL
- Stacks
- Queues

Abstract data type (**ADT**):

- A **collection of data**, along with a **set of operations** on that data.
- Independent of implementation and programming language.
- Examples: Sequence, SortedSequence.

Data structure

- A construct within a programming language that stores a collection of data.
- Examples: Array, Linked List.

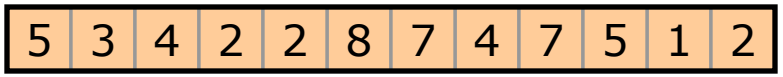
Class

- A feature in C++ and some other programming languages, aimed at facilitating OOP.
- In C++, we often implement a data structure using a class. However, we are not *required* to.
- Examples: `std::vector<int>`, `std::list<double>`.

Review
Introduction to Sequences

A **Sequence** is a collection of items that are in some order.

- We will restrict our attention to **finite** Sequences in which all items have the same type.



We defined an ADT **Sequence**.

- Data. An ordered list, all items the same type, indexed by 0, …, *size*–1.
- Operations. CreateEmpty, CreateSized, Destroy, Copy, LookUpByIndex, Size, Empty, Sort, Resize, InsertByPos, RemoveByPos, InsertBeg, RemoveBeg, InsertEnd, RemoveEnd, Splice, Traverse, Swap.

Review
Interface for a Smart Array [1/2]

We wish to implement a Sequence in C++ using a **smart array**. It will know its size, be able to copy itself, etc. It will also be able to *change* its size.

You will finish this implementation in Project 5.

Basic Ideas

- Use a C++ class. An object of the class implements a single Sequence.
- Use iterators, operators, ctors, and the dctor in conventional ways.
- *Every* function in the interface should exist in order to implement, or somehow make possible, an ADT operation.

Review
Interface for a Smart Array [2/2]

Ctors & Dctor

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

Member Operators

- Copy assignment
- Move assignment
- Bracket

Global Operators

None

Named Global Functions

None

Named Public Member Functions

- `size`
- `empty`
- `begin`
- `end`
- `resize`
- `insert`
- `erase`
- `push_back`
- `pop_back`
- `swap`

All design decisions so far have been made exactly the same as in `std::vector`— except that `vector` has other members, too.

Review
Basic Array Implementation — Design Decisions

Call our class `FSArray` (Frightfully Smart Array).

What type should an array item be?

- Use `int` for the value type.
- This is just for now. You will make it generic in Project 5.

How should we implement the iterators?

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

What data members should our array class have?

- Size of the array: `size_type _size;`
- Pointer to the array: `value_type * _data;`

Can we use automatically generated versions of the Big Five?

- No. We are directly managing an owned resource.

As we will see, this design actually has a significant flaw—which may not be obvious.

Basic Array Implementation — CODE

TO DO

- Write a skeleton form of class `FSArray`.
  - The package header & source files: `#ifndef`, `#include`, etc.
  - The class definition.
  - Definitions of all public types.
  - Prototypes and dummy definitions for all public functions.
- As time permits, begin implementing functionality.
  - Declarations of data members and comments indicating class invariants.
  - Definitions for functions that do not copy/move/swap or resize the array.
  - Definitions for member functions `push_back` & `pop_back`.

Done. See `fsarray.h` & `fsarray.cpp`. *Also see* `fsarray_main.cpp` *for a program to compile the package with.*

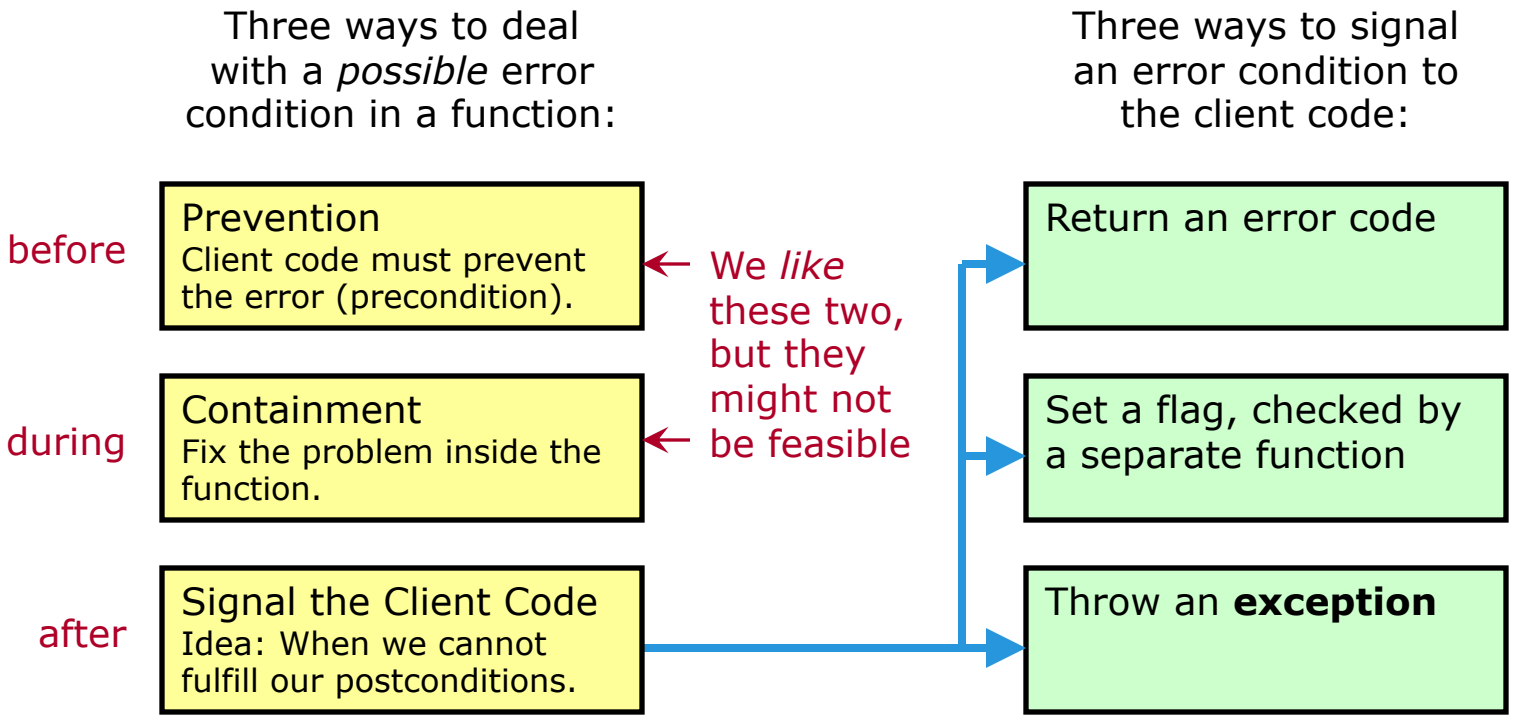
We will improve `FSArray` over the next few days. In Project 5 you will turn it into a generic container and finish it.

Exception Safety

Exception Safety
Refresher — Error Handling

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

- Not necessarily a bug or a user mistake.
- Example: Could not read file.



Exception Safety
Refresher — Using Exceptions [1/3]

Catch—when you can handle an error signaled by a function you call.

```
try { … }
catch (std::out_of_range & e) {
```

Catch exceptions by reference.

Throw—when your function is unable to fulfill its postconditions.

```
if (ix >= arrsize) throw std::out_of_range("bad index");
```

Catch all & re-throw—when you call a throwing function, and you cannot handle the error, but your function must clean up before exiting.

```
try { … }
catch (...) {
    [Clean up here]
    throw; }
```

The code contains three dots.

We generally **only write one** of the three: catch, throw, or catch all & re-throw.

Another might be written by someone else.

Refresher — Using Exceptions [2/3]

The following can throw:

- `throw` throws.
- `new` may throw `std::bad_alloc` or a derived class (default behavior).
- 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 documentation.

The following do *not* throw:

- Built-in operations, other than `new`, on built-in types.
  - Including `operator[]`.
- Deallocation done by the built-in version of `delete`.
- C++ Standard I/O Libraries (default behavior).

Refresher — Using Exceptions [3/3]

Marking a function noexcept is a promise that it will not throw.

```
void foo() noexcept
{ … }
```

If a noexcept function throws, then the program terminates.

If a destructor called during exception handling throws, then the program terminates. So **destructors should not throw**.

Because of this, generally the destructors in your classes are implicitly marked noexcept, unless you specify otherwise.

Exception Safety
Introduction [1/2]

The following issues are collectively called "**safety**":

- Does a function ever signal client code that an error has occurred, and if it does …
- Are resource leaks avoided?
- Are data left in a usable state?
- If so, do we know anything about that state?

In the context of exceptions, we use the term "**exception safety**".

A function's **guarantee** states the safety assurances it makes.

Exception Safety
Introduction [2/2]

When a function exits, having satisfied its postconditions, we say it has **succeeded**.

When a function exits without having satisfied its postconditions, we say it has **failed**.

We will follow the convention that a function throws an exception when it is unable to succeed.

So each function we call must do one of two things:

- Succeed and terminate normally (return), or
- Fail, throw an exception, and adhere to its safety guarantee.

Next we look at three standard safety guarantees. Then we discuss methods for writing functions that make helpful guarantees.

Exception Safety
Three Standard Guarantees — Overview

Basic Guarantee

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

These are the **Abrahams Guarantees**, formulated by C++ Standards Committee member Dave Abrahams in the 1990s. They can be applied to code in other programming languages as well.

Strong Guarantee

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

No-Throw Guarantee

- The operation never throws an exception.

Notes

- Each guarantee includes the earlier guarantees.
- 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.

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 all code to meet.

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

Three Standard Guarantees — Strong Guarantee

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

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

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

Sometimes it is not reasonable, typically due to efficiency issues.

The operation never throws an exception.

- In the context of error-handling methods other than exceptions, this may become the **No-Fail Guarantee**.

This is the highest level of safety, but not always the best level.

- Exceptions are not *bad*. They are a tool for dealing with problematic situations. The No-Throw Guarantee prohibits the use of this tool.
- This guarantee does not say "error conditions never occur"; it says that we handle them internally, never throwing to signal client code. Some kinds of errors make it difficult to offer this guarantee.

Sometimes it is necessary to make the No-Throw Guarantee—typically in situations in which we are *finishing something*.

- One such situation: destructors.
- We will cover another situation soon: commit functions.

Writing Exception-Safe Code — How

To ensure that code is exception-safe, look at *every* place an exception might be thrown. For each, make sure that, if an exception is thrown, then either

- the exception is caught and handled internally, or
- the function throws and adheres to its guarantees.

That can be a lot of work, but there are ideas that make it easier:

- **Modularity**—Once we certify a function as exception-safe, we can use it as such without re-examining it—as long as we do not alter it. A modular design, with lots of little functions, is one in which a previously certified function will rarely need to be rewritten.
- **RAII**—If a resource is owned by an object whose destructor is sure to be called, then we do not have to worry about leaking that resource.

Writing Exception-Safe Code — Design

A bad design may make exception safety impossible.

- Good design is part of exception safety.
- The **Single Responsibility Principle** (**SRP**)—every software component should have exactly one well defined responsibility—can be helpful here.

Consider:

- Suppose a function has two things to do, and the second thing fails.
- Suppose the second thing is returning a value.

Rule. **A non-const member function should not return an object by value.**

Story time …

See "Exception Handling: A False Sense of Security", linked on the class webpage.

Writing Exception-Safe Code — CODE

TO DO

- Do any improvements in class `FSArray` come to mind, now or in the process of doing the following steps? If so, consider making them.
- Figure out and document the exception-safety guarantees made by all functions implemented so far in class `FSArray`.
- Should any of these guarantees be changed? Perhaps a higher safety level can be achieved via a redesign/rewrite?
  - *No, all documented guarantees are as high as they can reasonably be.*
  - *The ctor from size offers the Strong Guarantee. We cannot raise its level of safety, because it does dynamic allocation, and so may fail.*
  - *All other functions written so far offer the No-Throw Guarantee.*
- Write an exception-safe copy ctor for class `FSArray`, and document its safety guarantee.
  - *The copy ctor offers the Strong Guarantee. Again, we cannot raise its level, as it does dynamic allocation.*

Done. See the latest versions of `fsarray.h` & `fsarray.cpp`.

Exception Safety
TO BE CONTINUED …

Exception Safety will be continued next time.