# Exception Safety continued

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, October 21, 2024

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

# Unit Overview
# Data Handling & Sequences

Topics

- ✓ Data abstraction
- ✓ Introduction to Sequences
- ✓ Interface for a smart array
- ✓ Basic array implementation
- (part) Exception safety
- Allocation & efficiency
- Generic containers
- Node-based structures
- More on Linked Lists
- Sequences in the C++ STL
- Stacks
- Queues

Smart Arrays

Linked Lists

# 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.
- Things we need to be able to do:
  - 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.

# Review
# Basic Array Implementation — Design Decisions

Class: `FSArray` (Frightfully Smart Array).

Value type: `int`, *for now*.

Iterators: pointers (`int *`, `const int *`).

We are implementing a Sequence as a smart array.

Data members:

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

Class invariants:

- Member `_size` is nonnegative.
- Member `_data` points to an `int` array, allocated with `new []`, owned by `*this`, holding `_size` `int`s.

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

# Exception Safety — Introduction

The following issues are collectively called "**safety**" (in the context of exceptions, "**exception 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?

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

In this class, we will follow the convention that each function that is called will do one of two things:

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

# Exception Safety — Three Standard Guarantees

A function's guarantee will usually be one of the following three.

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

- The minimum standard for all code.

**Strong Guarantee.** If the function throws an exception, then it makes no changes that are visible to the client code.

- The guarantee we generally prefer.

**No-Throw Guarantee.** The function never throws an exception.

- Required in some special situations.

Each guarantee includes the earlier guarantee(s).

# Exception Safety — Writing Exception-Safe Code [1/2]

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.

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.

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

# Exception Safety — Writing Exception-Safe Code [2/2]

DONE

- 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?
  - *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.*
  - *So all documented guarantees are as high as they can reasonably be.*
- 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

continued

# Exception Safety
## noexcept — Noexcept Specification

C++11 introduced the keyword `noexcept`, to enable the following:

- We can declare that a function will not throw—or will not throw except in certain circumstances.
- Code can test at runtime whether an expression is non-throwing.

Placing `noexcept` after a parameter list declares a function as throwing no exceptions. This is a **noexcept specification**.

```
void foo() noexcept;
```

If a noexcept function throws, then the program terminates.

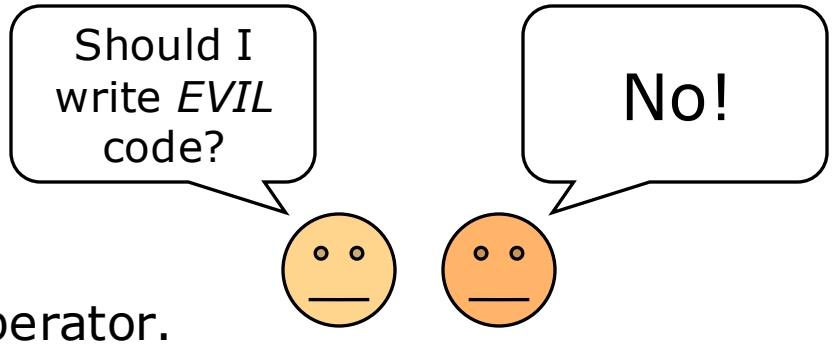
A destructor is implicitly marked noexcept, if the destructors of all data members—and base classes, if any—are noexcept, and you do not mark it otherwise.

Destructors will be noexcept, unless there is *EVIL* code lurking somewhere about.

# Exception Safety
# noexcept — When to Use It

Which functions should be noexcept?

- Destructor—but that is done for you, unless there is *EVIL* code.




- Move ctor and move assignment operator.
  - This enables a number of optimizations. For example, when a `vector` runs out of space, it does a reallocate-and-copy. If the value type has a noexcept move constructor, then the `vector` will **move** each data item; otherwise, it will **copy** them. (Consider why it does this.)
- Any function called by a noexcept function outside a try-block.
  - This is why we insisted on the `swap` member function being noexcept in Assignment 2—and will insist again in Assignment 5. The move assignment operator calls it, so it must be noexcept.
- *Optionally*, any function you are sure will never throw—even if that function is later rewritten.
  - Think of noexcept status as a *permanent* property of a function.

# Exception Safety
## noexcept — More Usage

`noexcept` is also an operator. Put a parenthesized expression after it. The result is `true` if the expression is noexcept.

```
if ( noexcept(bar()) )
    { … }  // Do this if bar() never throws
```

Code similar to this is how `vector` is able to check for a noexcept move ctor.

A noexcept *specification* optionally includes a parenthesized constant boolean expression. The function is noexcept if the expression is `true`.

```
void foo2() noexcept( noexcept(bar()) )
{
    bar();
}
```

`foo2` is noexcept if `bar` is noexcept.

This information is included for completeness, but you will probably not use it much.

# Exception Safety
# noexcept — CODE

## TO DO

- Write a noexcept move ctor for `FSArray`. If modifications to the class would help, then make those modifications.
  - *Member* `_data` *is now allowed to be a null pointer if* `_size` *is zero. We considered all the member functions, to make sure that they would operate properly with this change in the class invariants. The ctor from size and the copy ctor were changed.*
  - *Now the move ctor can copy the data members of its parameter, and then set the parameter's* `_size` *to* `0` *and* `_data` *to* `nullptr`*.*
- Make sure the exception-safety properties of the move ctor are correctly documented.
- If any other functions should be noexcept, then mark them as such.
  - *The move assignment operator and member function* `swap`*, neither written yet, have already been marked noexcept. We also marked as noexcept some simple functions that should never have to do anything that might throw:* `size`*,* `empty`*,* `begin`*,* `end`*.*

This deals with one of the flaws in our design.

*Done. See the latest versions of* `fsarray.h` *&* `fsarray.cpp`*. See* `fsarray_main2.cpp` *for another program that uses* `FSArray`*.*

# Commit Functions — The Need

It can be tricky to offer the Strong Guarantee when a single function modifies multiple parts of a large object.

- If we make several changes, and then we get an error, it can be difficult to undo the changes already made.
- What if the undo operation itself may result in an error?

An Idea That Often Works

- Create an entirely new object with the new value.
- If there is an error, destroy the new object. The old object has not changed, so there are no changes that are visible to the client.
- If there is no error, **commit** to our changes using a non-throwing function.

**Commit function:** a non-throwing function used to finalize the result of a computation.

*Swap* can be a useful commit function.

# Exception Safety
# Commit Functions — Swap [1/2]

A swap member function usually looks like this:

```
class MyClass {
    …
    void swap(MyClass & other) noexcept
    { … }
```

This should exchange the values of `*this` and `other`.

A swap member function can usually be written very easily: just swap the data members. Ownership issues are easy to handle properly (right?).

If we do it right, then we get a swap function that never throws and is very fast.

# Exception Safety
# Commit Functions — Swap [2/2]

```
class MyClass {
private:
    int _x;
    double * _y;
public:
    void swap(MyClass & other) noexcept;
```

We can implement `MyClass::swap` like this:

```
void MyClass::swap(MyClass & other) noexcept
{
    std::swap(_x, other._x);
    std::swap(_y, other._y);
}
```

This is the same as the `mswap` we discussed a few weeks ago in *Invisible Functions II*.

When we make such a member function public, we generally name it "`swap`". But it is not the same as `std::swap`!

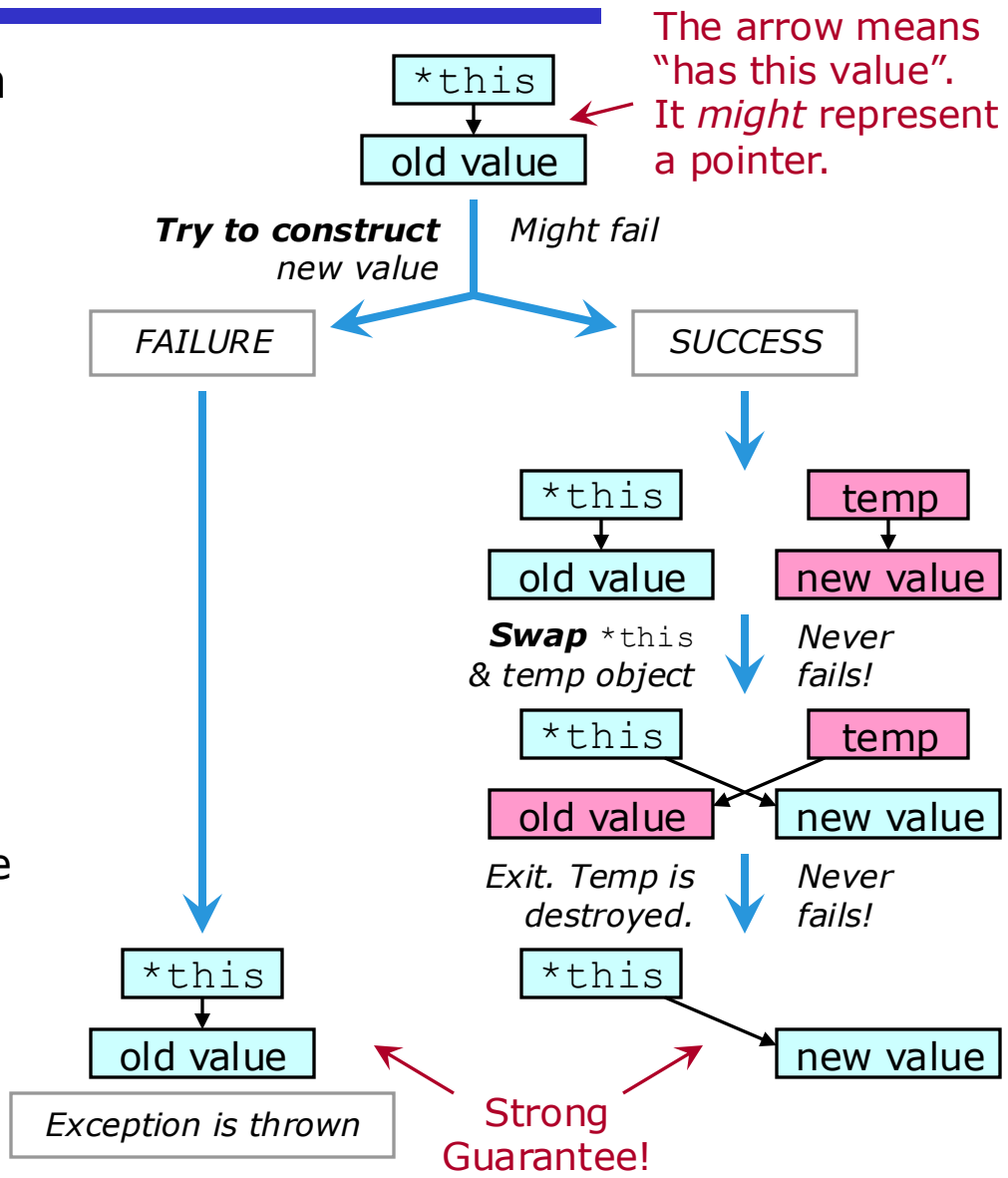
# Exception Safety
# Commit Functions — Usage [1/3]

Use a non-throwing swap function to get the Strong Guarantee.

To give our object a new value:

- **Try to construct** a temporary object holding this new value.
- If this fails, exit. No change.
  - Exiting is automatic, if the failing operation throws.
- If the construction succeeds, then **swap** our object with the temporary object holding the new value.
  - Swap never fails.
- Exit. The destructor of the temporary object cleans up the old value of our object.
  - Destruction is automatic.
  - It never fails (no *EVIL* code).

Above, boldface = code we write.



# Exception Safety
# Commit Functions — Usage [2/3]

We can set an object to a new value, while offering the Strong Guarantee, if we can construct the new value with the Strong Guarantee, and we have a non-throwing dctor and swap.

Procedure

- **Try to construct** a temporary object holding the new value.
- **Swap** with this temporary object.

Example: "clear" by swapping with a default-constructed temporary object.

```
void MyClass::clear()  // Strong Guarantee
{
    MyClass temp;
    swap(temp);
}
```

If there is a problem creating `temp`, then an exception is thrown, and "nothing" happens (Strong Guarantee).

Otherwise, the values are swapped. `*this` gets its new value. The old value of `*this` is cleaned up by `temp`'s destructor.

# Exception Safety
# Commit Functions — Usage [3/3]

Now we can write a copy assignment operator that makes the Strong Guarantee. We need:

- A copy ctor that makes the Strong Guarantee (usually possible).
- A swap member function that makes the No-Throw Guarantee (usually easy).
- A dctor that makes the No-Throw Guarantee (of course).

This is the same way we wrote copy assignment back in *Invisible Functions II*.

And this is *why* it was written that way.

```
// Strong Guarantee
MyClass & MyClass::operator=(const MyClass & rhs)
{
    MyClass temp(rhs);
    swap(temp);
    return *this;
}
```

Do the actual assignment:

1. **Try to construct** a temporary copy of `rhs`.
2. **Swap** with the temporary copy.

The old value is cleaned up by the destructor of `temp`, which should never throw.

Always end an assignment operator this way.