

# Exception Safety, cont'd

## Allocation & Efficiency

---

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, March 31, 2008

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

[CHAPPELLG@member.ams.org](mailto: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
- 1/2✓ • Exception Safety
  - Allocation & Efficiency
  - Generic Containers
  - Node-Based Structures
  - Linked Lists
  - Sequences in Practice

# Review

## Data Structure Implementation

---

What data members should class `Sequence` have?

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

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`.

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

# Review

## Exception Safety — Review of 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.

# Review

## Exception Safety — The Three Standard Guarantees

---

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**”.

### **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

- 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.

## Review

# Exception Safety — Writing Exception-Safe Code

---

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.

A bad design can force us to be unsafe.

- Thus, good design is part of of exception safety.
- An often helpful idea is that **everything has exactly one purpose**. Code that follows this principle is **cohesive**.
  - In particular: A non-const member function should not return an object by value.

# Exception Safety, cont'd

## Writing Exception-Safe Code — Do It

---

### TO DO

- Figure out and comment the exception-safety guarantees made by all functions implemented so far in class `Sequence`.

*Done. See (the latest versions of) `sequence.h`, `sequence.cpp`, on the web page.*

- Can/should any of these be improved?
  - *No, all the constructors got the Strong Guarantee, which is as much as we can expect, since they allocate. Every other function that is written got the No-Throw Guarantee. These will need to be re-evaluated if the class is turned into a template (as you will do for Assignment 5).*

## Exception Safety, cont'd

### Commit Functions — The Need

---

Often it is tricky to offer the Strong Guarantee when modifying 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.
- In fact, it may be impossible, if the undo operation itself may result in an error.

### Solution

- 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 is nothing to roll back.
- If there is no error, **commit** to our changes using a non-throwing operation.

A good commit function is often a non-throwing **swap** function.

## Exception Safety, cont'd

### Commit Functions — Swap [1/3]

---

Swap member functions usually look like this:

```
void MyClass::swap(MyClass & other);
```

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

Swap functions can *usually* be written very easily.

- Just swap the data members.
- Ownership issues are easy to handle properly (right?).

In fact, it is usually easy to write a swap function that is:

- Non-throwing.
- *Very* fast.

## Exception Safety, cont'd

### Commit Functions — Swap [2/3]

---

```
class MyClass {  
private:  
    int x;  
    double y;  
public:  
    void swap(MyClass & other); // Does not throw
```

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

```
void MyClass::swap(MyClass & other) // Does not throw  
{  
    int tempi = x;  
    x = other.x;  
    other.x = tempi;  
  
    double tempd = y;  
    y = other.y;  
    other.y = tempd;  
}
```

## Exception Safety, cont'd

### Commit Functions — Swap [3/3]

---

Alternatively, we can use `std::swap`, in `<algorithm>`:

```
void MyClass::swap(MyClass & other)
{
    std::swap(x, other.x);
    std::swap(y, other.y);
}
```

If we need to swap members that are **objects**, we might want to avoid `std::swap`.

- Algorithm `std::swap` uses the copy ctor and copy assignment. These might throw.
- If the objects have their own non-throwing swap member function, we can use that:

```
void MyClass::swap(MyClass & other)
{
    ...
    z.swap(other.z); // z is a data member of class type
}
```

The moral: A non-throwing swap member function is a good thing to have.

- This is why many C++ Standard Library classes have such a member function.

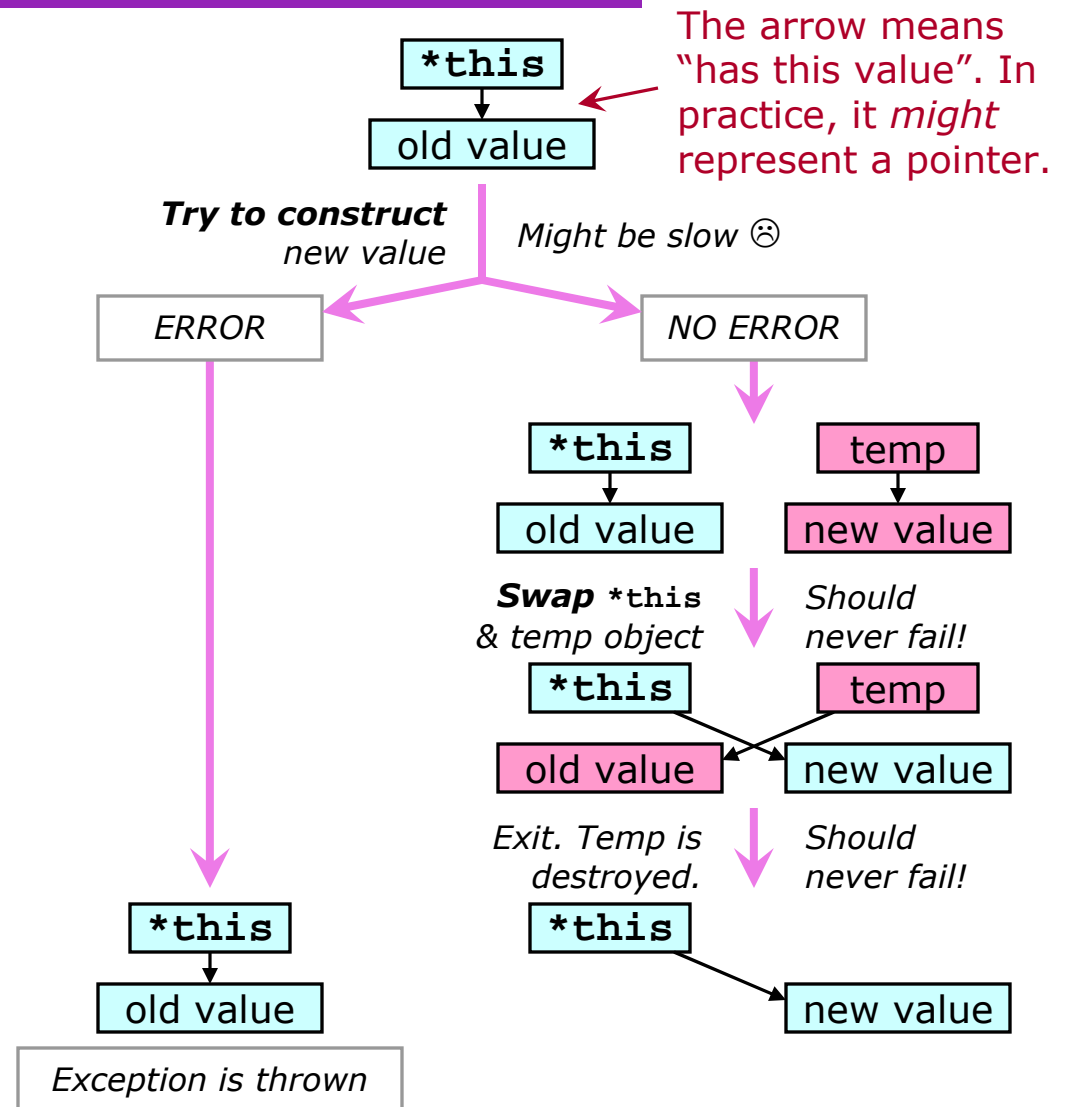
# Exception Safety, cont'd

## Commit Functions — Usage [1/3]

We can use a non-throwing swap function to get the Strong Guarantee:

- To give our object a new value, first **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.
- Exit. The destructor of the temporary object cleans up the old value of our object.
  - Destruction is automatic.
  - And it should never fail.

Note: boldface = code we write.



## Exception Safety, cont'd

### Commit Functions — Usage [2/3]

---

Thus, we can set an object to a new value, while offering the Strong Guarantee, as long as we have a way to construct the new value that offers the Strong Guarantee, along with a ctor and a swap function that offer the No-Throw Guarantee.

Procedure

- **Try to construct** a temporary object holding the new value.
- **Swap** with the above constructed 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, cont'd

## Commit Functions — Usage [3/3]

This idea lets us write a copy assignment operator that makes the Strong Guarantee. We need:

- A copy ctor that offers the Strong Guarantee (this is usually not too difficult).
- A swap member function that makes the No-Throw Guarantee (usually easy).
- A dtor that makes the No-Throw Guarantee (of course).

Code:

```
MyClass & MyClass::operator=(const MyClass & rhs) // Strong Guarantee
{
    if (this != &rhs) ← Check for self-assignment (standard).
    {
        MyClass temp(rhs); ← Do the actual assignment:
        swap(temp);         1. Try to construct a temporary copy of rhs.
                           2. Swap with the temporary copy.
    }
    return *this; ← The old value is cleaned up by the destructor
                  of temp (which does not throw).
} ← Always end an assignment operator this way.
```

Admittedly this is a bit mind-twisting. However, assuming the requirements are met, it is easy to write, and it always works.

# Allocation & Efficiency

## Do It?

---

### TO DO

- Consider how to write `Sequence::resize`.

*Discussed, but no code  
was written yet.*

### Ideas

- *If we are resizing smaller than (or equal to) the current size, just change the `size_` member to the new value.*
- *If we are resizing larger than the current size, then reallocate a large-enough chunk of memory for the array, copy the data there, and increase `size_` to the new value ("reallocate-and-copy").*
- *But the above method has a problem. For example, suppose we are using a `Sequence` object to implement a `Stack`. Pushing a new item on the end always requires a reallocate-and-copy, which will be very inefficient.*

# Allocation & Efficiency

## Amortized Constant Time

---

For a smart array, insert-at-end is linear time.

- It is constant time if space is available (already allocated).
- It is linear time in general, due to reallocate-and-copy.

We can speed this up much of the time if we reallocate very rarely.

- Idea: When we reallocate, get more memory than we need. Say twice as much. Then do not reallocate again until we fill this up.

Now, using this idea, suppose we do **many** insert-at-end operations. How much time is required by  $k$  insert-at-end operations?

- Answer:  $O(k)$ .
  - If, when we reallocate-and-copy, we increase the reserved memory by some constant factor.
- Even though a single operation is not  $O(1)$ .

If  $k$  consecutive operations require  $O(k)$  time, we say the operation is **amortized constant time**.

- Amortized constant time means constant time on average over a large number of consecutive operations.
- It does **not** mean constant time on average over all possible inputs.
- This is our last efficiency-related terminology.

# Allocation & Efficiency

## Do It Again

---

How can we redesign class `sequence` internally, so that we can write an amortized constant-time insert-at-end?

- A third data member can hold the amount of memory allocated. This is called the “capacity”.

### TO DO

- Finish the details of this new design. How does it work?
- Rewrite (most of) the existing member functions and invariants in `sequence` to use the new design.

*Done. See (the latest versions of)  
`sequence.h`, `sequence.cpp`,  
on the web page.*

*This is the last revision of these  
files that I will do.*