

Generic Containers

Node-Based Structures

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, April 2, 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

Exception Safety [1/8]

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 [2/8]

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 [3/8]

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 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.

Review

Exception Safety [4/8]

Often it is tricky to offer the Strong Guarantee when modifying multiple parts of a large object.

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.

Review

Exception Safety [5/8]

Swap member functions usually look like this:

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

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

If the data members are all built-in types (including pointers!), then we can usually just call `std::swap` on them.

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

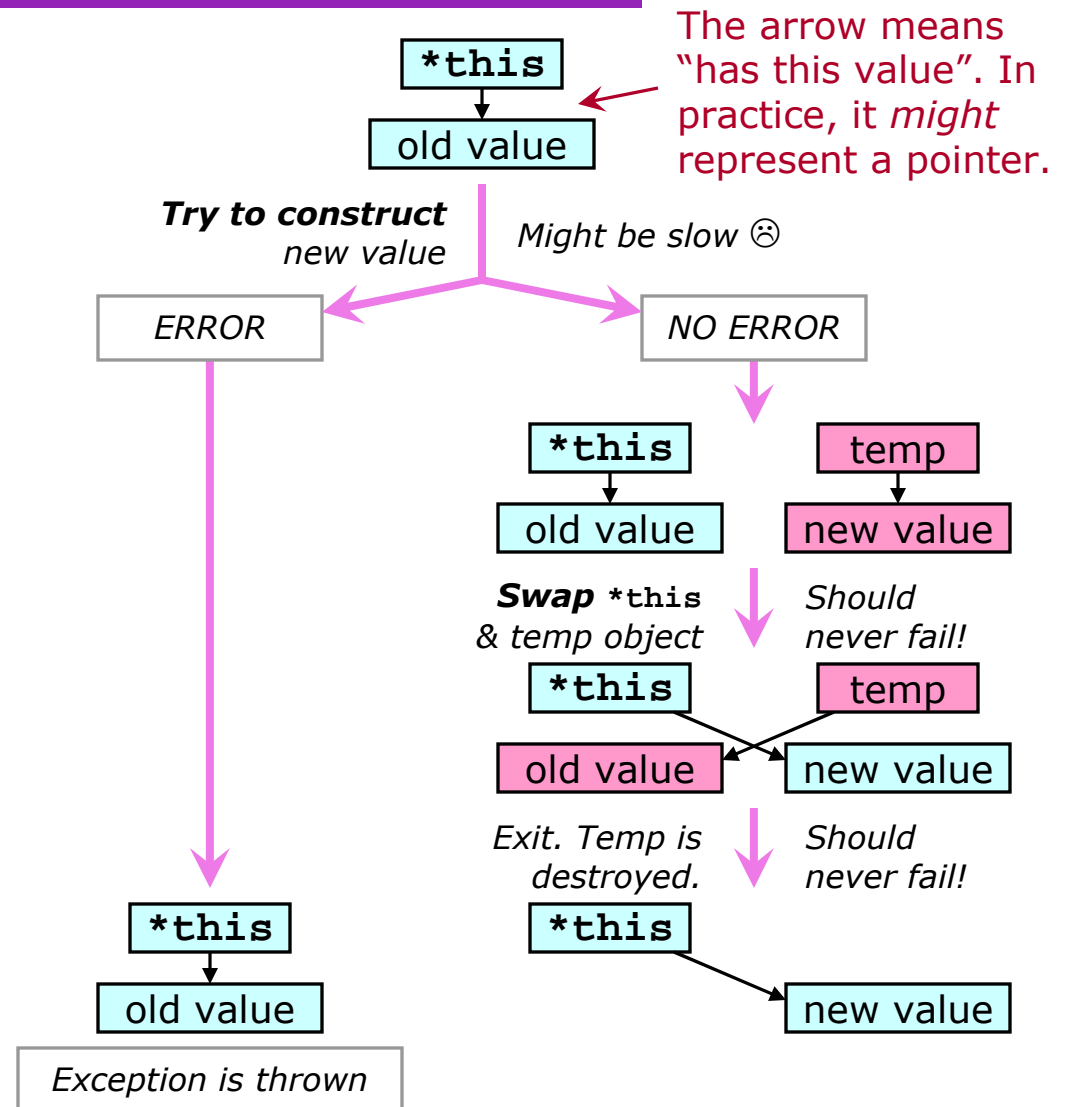
Review

Exception Safety [6/8]

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.



Review

Exception Safety [7/8]

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.

Review

Exception Safety [8/8]

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.

Review

Allocation & Efficiency

An operation is **amortized-constant-time** if k operations require $O(k)$ time.

- Thus, over many consecutive operations, the operation averages constant time.
- *Not* the same as constant-time average case.
- Quintessential amortized-constant-time operation: insert-at-end for a well written (smart) array.

Fixing Class **Sequence**

- Our original design did not allow for efficient insert-at-end.
 - Reallocate-and-copy would happen every time.
- The revised design had three data members: size, capacity, and the array pointer.
- Having a “capacity” member allows us to keep a record of how much memory is allocated. Then we can allocate extra so that we do not need to reallocate every time.
- Result: amortized-constant-time insert-at-end.

Generic Containers

Introduction

A **generic container** is a container that can hold a client-specified data type.

Examples

- Arrays
- STL containers, including `std::vector`.

In C++, we usually implement a generic container using a **class template**.

Generic Containers

Class Templates — Recall ...

The C++ Standard does **not** require compilers to be able to do separate compilation of templates.

- Thus, you should define all member functions of a class template and all associated global function templates in your header file.
- With templates, you probably will not have a source (`.cpp`) file.

When you write a class template, indicate the **requirements on the types** it takes as template parameters.

- Typically: must have certain member functions and/or operators, and the dtor must not throw.
- It is assumed that member functions must all offer at least the Basic Guarantee. You do not need to mention this.

Generic Containers

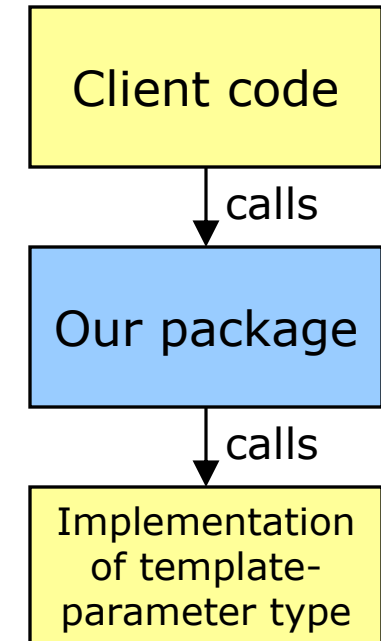
Exception Safety [1/2]

When we write a template, we deal with the type given to us using its own member functions. These client-provided functions **may throw**.

- Unless we require that they do not (in our “requirements on types”).

Exception safety gets harder.

- The same procedures apply, but now we have many more places that might generate exceptions.



↑
This code
might throw ...

Generic Containers

Exception Safety [2/2]

Since **every** member function of a template parameter type, that is not specifically prohibited from throwing, may throw, we need to check **every** use of such a member function, to make sure that we deal with them correctly.

- Do not forget silently called functions (default ctor & copy ctor) and operators.

One tricky situation is copying the data in a dynamic array. Copy assignment of a class type can throw, often requiring deallocation.

```
arr = new MyType[10];  
std::copy(a, a+10, arr); // Memory leak,  
                        // if copy assignment throws.
```

We will come back to this example shortly.

Generic Containers

Exception Neutrality

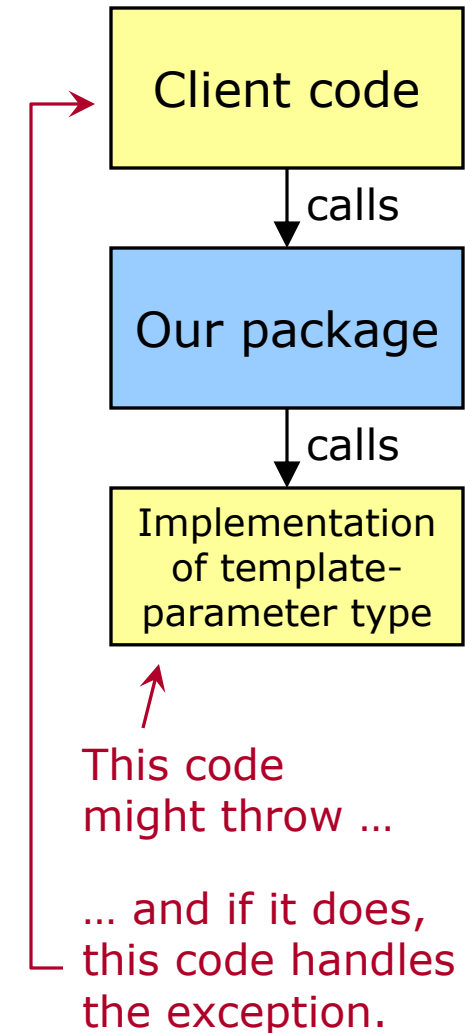
Another issue: When we use client-provided functions, the client may want to see exceptions the functions throw.

- A function that allows exceptions thrown by a client's code to propagate unchanged, is said to be **exception-neutral**.

When exception-neutral code calls a client-provided function that may throw, it does one of two things:

- Call the function outside a try block, so that any exceptions terminate our code immediately.
- Or, call the function inside a try block, then catch all exceptions, do any necessary clean-up, and re-throw:

```
try {
    x.func(); // x is var of client's type;
              // func may throw.
}
catch (...) { // Catch everything;
              // we don't know what func throws.
    [Do our own clean-up here]
    throw; // Re-throw same exception.
}
```



Generic Containers

Exception Safety & Neutrality Together

Putting it all together, we can use catch-all, clean-up, re-throw to get both exception safety and exception neutrality.

```
arr = new MyType[10];  
try  
{  
    std::copy(a, a+10, arr);  
}  
catch (...)  
{  
    delete [] arr;  
    throw;  
}
```

← Called outside any `try` block. If this fails, we exit immediately, throwing an exception.

← Called inside a `try` block. If this fails, we need to deallocate the array before exiting.

← This helps us meet the Basic Guarantee (also the Strong Guarantee if this function does nothing else).

← This makes our code exception-neutral.

Generic Containers

Thoughts on Assignment 5

Thoughts on writing some of the member functions for Assignment 5 in an exception-safe and exception-neutral manner.

- Function **swap**
 - Use `std::swap` on all data members. *Example is on an earlier slide.*
- Copy ctor
 - Allocate *outside* `try` block. Copy *inside* a `try` block. Catch-all, clean-up, re-throw. *Similar to the code on the previous slide.*
- Copy assignment
 - Write as discussed earlier, using `swap` (the member, *not* `std::swap!`). *Example is on an earlier slide.*
- Function **resize**
 - If resizing to \leq capacity: just set `size_`.
 - If resizing to $>$ capacity: use the swap trick: Create temp with the right size, `std::copy`, swap with temp.
- Functions **insert** & **remove**
 - Use `resize` and `std::rotate`.

Node-Based Structures

Introduction

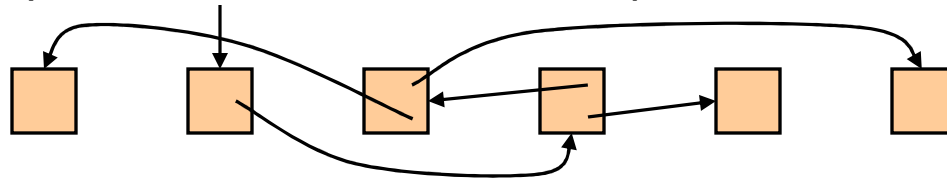
Our fundamental building block for data structures has been the **array**.



- Items are stored in contiguous memory locations.
- There is little memory-management overhead.
- Look-up operations are usually very fast.
- Operations that require rearrangement (insert, remove, etc.) can be slow.

Now we begin looking at data structures built out of **nodes**.

- A *node* is generally a small block of memory that is referenced via a pointer, and which may reference other nodes via pointers.



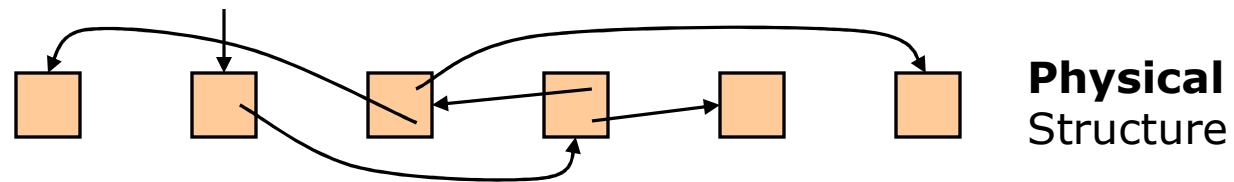
- Node-based structures do not necessarily store data in contiguous memory.
- Memory-management overhead becomes significant.
- To find a node, we follow a chain of pointers. Look-up can be slow.
- Operations that require rearrangement can be very fast.

Node-Based Structures

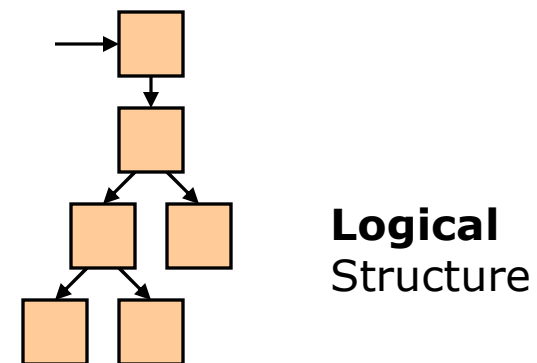
Pictures, Physical vs. Logical

When we draw pictures of node-based data structures, the positions of nodes in the picture have nothing to do with their positions in memory.

For example, if a structure is stored like this ...



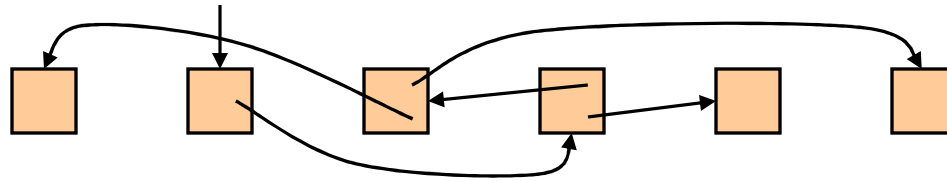
... then we might draw it like this:



Node-Based Structures

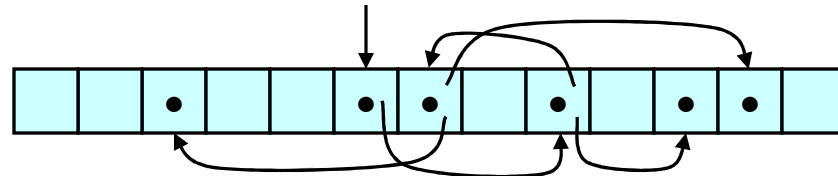
Storing Nodes in an Array

We normally store nodes in separately allocated blocks of memory.



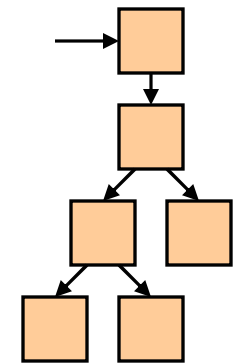
However, we might allocate an **array of nodes**.

- Replace pointers with array indices, if desired.



This is still a node-based data structure.

- It still has most of the pro's & con's of node-based structures.
- The main differences involve **memory management**: who does it & when it gets done.



Both of these have the above **logical** structure.