Allocation & Efficiency
Generic Containers
Notes on Assignment 5

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, October 30, 2009

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2009 Glenn G. Chappell

Unit Overview
Handling Data & Sequences

Major Topics

- ✓ Data abstraction
- ✓ Introduction to Sequences
- Smart arrays
  - ✓ Array interface
  - ✓ Basic array implementation
  - ✓ Exception safety
  - Allocation & efficiency
  - Generic containers
- Linked Lists
  - Node-based structures
  - More on Linked Lists
- Sequences in the C++ STL
- Stacks
- Queues

Review
Array Interface

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
Basic Array Implementation

What data members should class **SmArray** 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 **size_ int**s.

Note: This design has a serious (but not obvious) problem, as we will see.

Review
Exception Safety — Refresher [1/3]

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

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 {
    p = new Foo[mySize];  // May throw
}
catch (...) {
    fixThingsUp();
    throw;
}
```

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

Review
Exception Safety — Refresher [2/3]

The following can throw in C++:

- "**`throw`**" throws.
- "**`new`**" may throw **`std::bad_alloc`** if it cannot allocate.
- 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)

Review
Exception Safety — Refresher [3/3]

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

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

Review
Exception Safety — Introduction [1/2]

Issues: Does a function ever signal client code that an error has occurred, and if it does:

- Are the data left in a usable state?
- Do we know something about that state?
- Are resource leaks avoided?

These issues are collectively referred to as **safety**.

We consider these in the context of exceptions: **exception safety**.

However, **most of the ideas we will discuss apply to any kind of error signaling technique**.

Review
Exception Safety — Introduction [2/2]

There are a number of commonly used safety levels.

- These are stated in the form of **guarantees** that a function makes.

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

- When a function exits without satisfying its postconditions, we will say it has **failed**.

Thus, a function we write must do one of two things:

- Succeed (satisfy its postconditions), or
- Fail, throw an exception, and satisfy its safety guarantee.

Review
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 code.

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 [1/2]

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 every module has exactly one well defined responsibility (the **Single Responsibility Principle**).
- In particular: A non-const member function should not return an object by value.

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

TO DO

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

*Done. See the latest versions of **`smarray.h`**, **`smarray.cpp`**, on the web page.*

- Can/should any of these be improved?
  - *No. All the constructors offer the Strong Guarantee, which cannot be improved, since they do dynamic allocation, and so might fail. All other functions written so far offer the No-Throw Guarantee.*

Review
Exception Safety — Commit Functions [1/5]

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 a non-throwing **swap** function.

Review
Exception Safety — Commit Functions [2/5]

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);
}
```

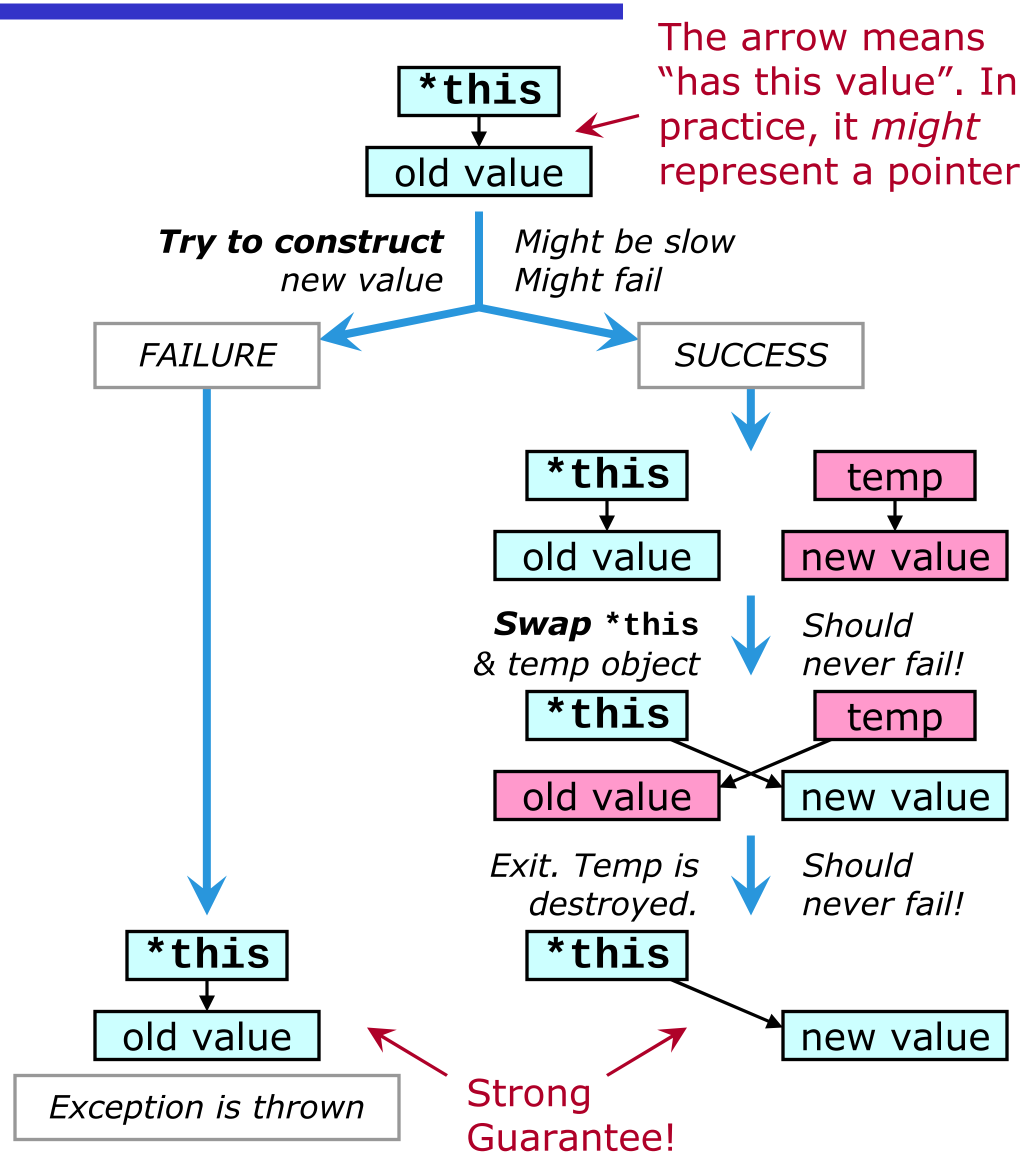
Review
Exception Safety — Commit Functions [3/5]

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 — Commit Functions [4/5]

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);
}
```

Review
Exception Safety — Commit Functions [5/5]

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 dctor that makes the No-Throw Guarantee (of course).

Code:

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

Check for self-assignment (standard). (→ `if (this != &rhs)`)

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 does not throw).

Always end an assignment operator this way. (→ `return *this;`)

Allocation & Efficiency
Write It?

TO DO

- Consider how to write **`SmArray::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 [1/2]

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
Amortized Constant Time [2/2]

Recall our time-efficiency categories.

Using Big-O	**In Words**
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(b^n)$, for some $b > 1$	Exponential time

Q: Where does “amortized constant time” fit in the above list?

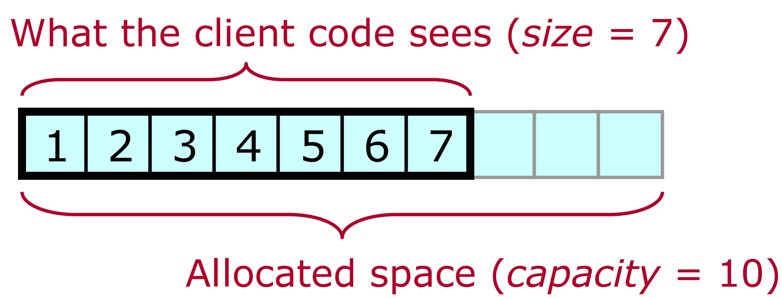
A: It doesn’t!

- The above are talking about the time taken by a single operation. “Amortized …” is not.
- Insert-at-end for a well written smart array is amortized constant time. It is also still linear time.

Allocation & Efficiency
Write It Again

How can we redesign class **`SmArray`** 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 **`SmArray`** to use the new design.

*Done. See the latest versions of **`smarray.h`**, **`smarray.cpp`**, on the web page.*

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 people write code that uses your template, they need to know what types it is usable with.

- In this class, when writing a template, include comments indicating **requirements on the types** it takes as template parameters.
  - Typically: must have certain member functions and/or operators, and the dctor must not throw.
- It is assumed that member functions must all offer at least the Basic Guarantee. You do not need to mention this.
  - If you need some member function to offer a stronger guarantee (e.g., destructor must not throw), then you *do* need to mention this.

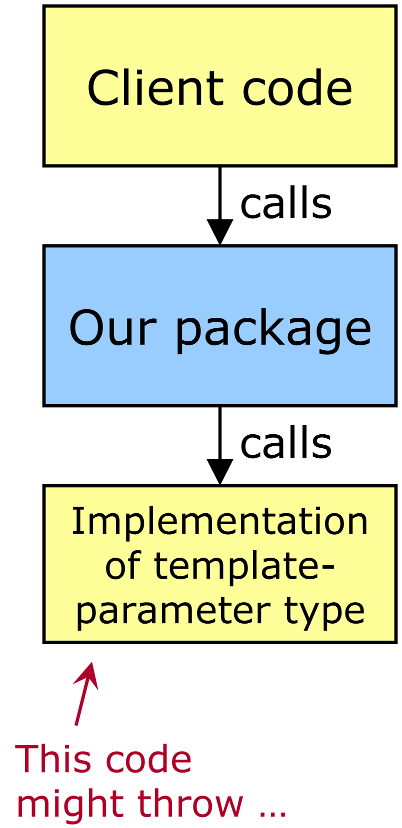
Generic Containers
Exception Safety [1/3]

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 trickier.

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



Generic Containers
Exception Safety [2/3]

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).
- Operators (in particular: assignment).
- STL algorithms. Those that modify a data set (**`std::copy`**, **`std::swap`**, **`std::rotate`**, etc.) generally do so using the assignment operator. If the assignment operator can throw, then these STL algorithms can throw.

Do *not* worry about these when they are called on built-in types.

```
void size(std::size_t theSize) const;
```

Passed by value. Copy ctor call? But **`std::size_t`** is a built-in type; this will not throw.

Generic Containers
Exception Safety [3/3]

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

We will come back to this example shortly.

Generic Containers
Exception Neutrality

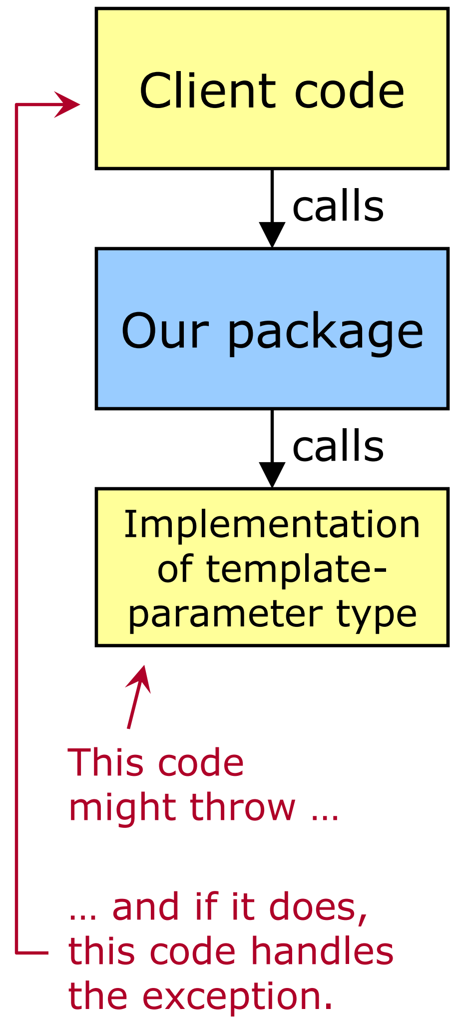
When we call client-provided functions, the client code generally handles any exceptions thrown.

Exception-neutral code allows exceptions thrown by client-provided code to propagate unchanged.

When such code calls a client-provided function that may throw, it must do 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, catch all exceptions, do necessary clean-up, and re-throw:

```
try {
    x.func();  // May throw
}
catch (...) {  // Exception not handled here
    [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[size];
try
{
    std::copy(a, a+size, arr);
}
catch (...)
{
    delete [] arr;
    throw;
}
```

- `arr = new MyType[size];` — Called outside any **try** block. If this fails, we exit immediately, throwing an exception.
- `std::copy(a, a+size, arr);` — Called inside a **try** block. If this fails, we need to deallocate the array before exiting.
- `delete [] arr;` — This helps us meet the Basic Guarantee (also the Strong Guarantee if this function does nothing else).
- `throw;` — This makes our code exception-neutral.

Notes on Assignment 5
Overview of Ideas

This ends the material that Assignment 5 covers.

Next, we will look at node-based structures. You do *not* need to worry about these on Assignment 5.

You *do* need to be concerned with:

- Invariants, Templates
  - Document everything properly.
- Exception Safety
  - Are your member functions offering the proper guarantee?
    - All functions must offer *at least* the Basic Guarantee.
    - Constructors generally need to offer a high level of exception safety.
    - Destructors and commit functions (**`swap`**) offer an even higher level.
    - Functions that do large-scale modifications (**`resize`**, **`insert`**, **`remove`**) will probably not offer a high level, for the sake of efficiency.
  - Are your member functions *satisfying* their guarantees?
    - Have you checked *every* place that might throw.
    - For a template, this includes things like **`std::copy`**, **`std::rotate`**.
- Allocation & Efficiency
  - Are functions that might need to do a reallocate-and-copy (**`resize`**, **`insert`**) written to handle this efficiently?

Notes on Assignment 5
Individual Functions [1/2]

Thoughts on making some Assignment 5 member functions exception-safe and exception-neutral.

- Function **`swap`**
  - Use **`std::swap`** on all data members. *Example is on earlier slide.*
- Copy ctor
  - Allocate *outside* **`try`** block. Copy *inside* a **`try`** block. Catch-all, clean-up, re-throw. *Same idea as the code two slides back.*
- Copy assignment
  - Write as discussed earlier, using the copy ctor and **`swap`** (the member, *not* **`std::swap`**!). *Example is on an earlier slide.*
- Function **`resize`**
  - If resizing to ≤ capacity: just set **`size`**.
  - If resizing to > capacity: create temp with the right size & capacity, **`std::copy`**, delete old & set members to new values.
    - The temp *could* be a separate object, and then you could use the swap trick. But if you do this, then make sure the temp's capacity is set correctly!
    - Alternatively, do not create a separate object. Have 3 variables that hold new values for the data members: **`newSize`**, **`newCapacity`**, **`newData`**. If this works, then delete the old data, and set all 3 data members to the new values.

Notes on Assignment 5
Individual Functions [2/2]

Thoughts (cont'd)

- Function **`remove`**
  - You need to resize the array. Function **`resize`** does this. Use it!
  - Suggestion: Do a **`std::rotate`**, and then call **`resize`**.
- Function **`insert`**
  - Again, call **`resize`** to do the resizing of the array. Do not duplicate code!
  - Suggestion: Call **`resize`**, put the new item in, and then **`std::rotate`**.
  - At the end, you need to return an iterator to the inserted item. This *would* be the same as the parameter, except that **`resize`** might have done a reallocate-and-copy. So: *Before* calling resize, save the *index* of the spot to insert, and then afterwards recreate the iterator from this index, and return it.