

# Allocation & Efficiency

## Generic Containers

### Thoughts on Project 5

---

CS 311 Data Structures and Algorithms  
Lecture Slides  
Wednesday, October 21, 2020

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

---

# Review

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
  - Node-based structures
  - More on Linked Lists
  - Sequences in the C++ STL
  - Stacks
  - Queues
- 
- The diagram uses red dotted lines and curly braces to group topics. A brace on the right groups the first five items (Interface for a smart array, Basic array implementation, Exception safety, Allocation & efficiency, and Generic containers) under the label 'Smart Arrays'. Another brace on the right groups the next three items (Node-based structures, More on Linked Lists, and Sequences in the C++ STL) under the label 'Linked Lists'. The final two items, 'Stacks' and 'Queues', are not grouped.

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

---

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.

**Exception safety.** Does a function ever throw, and if so:

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

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

← Minimum standard

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

← Preferred; may not be offered, usually for efficiency reasons

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

← Required in some special situations

Each guarantee includes the earlier guarantees.



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.**

*Related changes were made  
in `fsarray.h`, `fsarray.cpp`.*

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

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

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.

Make the move ctor and move assignment operator `noexcept`, along with functions they call. This enables various optimizations.

*Related changes were made in `fsarray.h`, `fsarray.cpp`.*

The noexcept status of a function call or other expression can be tested at runtime, using the **noexcept operator**.

```
if (noexcept( foo() ))  
{  
    ...
```

This is what allows the move-related optimizations mentioned on the previous slide to be done.

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

If we need to alter data in a way that is difficult to undo, but we still want to offer the Strong Guarantee, then we may wish to write our code as follows:

- Attempt to construct the altered version of the data.
- If this fails, then exit, destroying the attempt (generally automatic).
- If the attempt succeeds, then use a commit function to **commit** to the new version of the data.

Some possible commit functions:

- Swap member function.
- Move constructor.
- Move assignment.

We are using this one.



*Related changes in `fsarray.h`,  
`fsarray.cpp` are left for you to make.*

---

# Allocation & Efficiency

# Allocation & Efficiency Problem

---

Consider how to write `FSArray::resize`.

## Method

- If we resize smaller than or equal to the current size, then just set the `_size` member to the new value. (We would need to change the class invariants: `_data` points to an array of *at least* `_size` ints.)
- If we resize larger, then allocate a new memory block that is large enough for the new array, copy the data there, and increase `_size` to the new value. We call this **reallocate-and-copy**.

## Problem

- Suppose we use `push_back` to add a large number of items to an `FSArray`. Calling `push_back` *always* does a `reallocate-and-copy`, so this way of adding item is very inefficient.
- With the current design, there is *no way* to write an efficient `insert-at-end` for `FSArray`.

This is the significant—  
but not obvious—flaw  
mentioned earlier.

## Allocation & Efficiency

### Amortized Constant Time [1/3]

---

For a smart (resizable) 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 a possible reallocate-and-copy.

We can speed this up *most of the time* if we reallocate very rarely.  
When we reallocate, get more memory than we need. Perhaps twice as much. Then do not reallocate again until we fill this up.

Q. Using this idea, suppose we do *many* insert-at-end operations.  
How much time is required for  $k$  insert-at-end operations?

A.  $O(k)$ .

- If reallocate-and-copy ups allocated memory by a constant *factor*.
- Even though a single operation is not  $O(1)$ .

## Allocation & Efficiency

### Amortized Constant Time [2/3]

---

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

This is our last  
efficiency-related  
terminology.

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.)

For each of the efficiency categories, it is a good idea to have in mind an algorithm or operation in that category.

- Constant-time example: look-up by index in an array.
- Logarithmic-time example: Binary Search.
- Linear-time example: lots of possibilities, e.g., finding a maximum.
- Log-linear-time example: a fast comparison sort (Merge Sort, Introsort, Heap Sort).
- Amortized constant-time example: insert-at-end for a well written resizable array.



## Allocation & Efficiency

### Amortized Constant Time [3/3]

---

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
$O(c^n)$ , for some $c > 1$	Exponential time

Q. Where does *amortized constant time* fit into the above list?

A. It does *not* fit into the list!

The above are all about the worst-case time required for a *single operation*; amortized constant-time is not.

Insert-at-end for a well written resizable array is amortized constant-time. It is also still linear time.

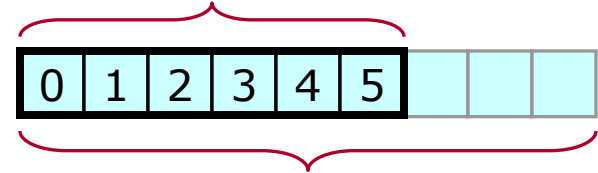
## Allocation & Efficiency

### Array Redesign — Internals

Q. How can we revise `FSArray`—internally—to allow for amortized constant-time `push_back`?

A. Track allocated space (**capacity**) along with **size** of client's dataset.

Client's data (size = 6)



Our allocated space (capacity = 9)

Finish the details of this revised design.

- Three data members: `_capacity`, `_size`, `_data`.
- Class invariant:  $0 \leq \_size \leq \_capacity$ .
- Class invariant: `_data` points to an array of ~~`_size`~~ `_capacity` ints; except: `_data` may be `nullptr` if ~~`_size`~~ `_capacity` is 0.
- A default-constructed `FSArray` will probably be resized larger. Set its *capacity* to something nonzero.
- When resizing to current *capacity* or smaller, just set `_size`.
- When resizing to larger than current capacity, reallocate-and-copy: *new\_capacity* is at least  $2 \times \text{old\_capacity}$ , and at least *new\_size*.

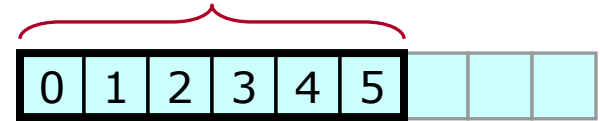
# Allocation & Efficiency

## Array Redesign — CODE

### Internal redesign:

- Three data members: `_capacity`, `_size`, `_data`.
- $0 \leq \text{\_size} \leq \text{\_capacity}$ .
- `_data` points to an array of `\_capacity` ints; except: `_data` may be `nullptr` if `_capacity` is 0.
- A default-constructed `FSArray` should have nonzero *capacity*.
- When resizing to current *capacity* or smaller, just set `_size`.
- When resizing to larger than current capacity, *new\_capacity* should be at least  $2 \times \text{old\_capacity}$ , and also at least *new\_size*.

Client's data (*size* = 6)



Our allocated space (*capacity* = 9)

### TO DO

- Make appropriate changes to the parts of the `FSArray` package that have been written.

*Done. See the latest versions of `fsarray.h` & `fsarray.cpp`. This is the last change I expect to make to this code.*

---

# Generic Containers

## Generic Containers

### Class Templates

---

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

- Examples: most STL containers, including `std::vector`.

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

When others write code that uses your template, they need to know your *requirements on types*.

- What member functions must exist, required safety levels.
- It is assumed that all functions offer at least the Basic Guarantee. You do not need to mention that requirement.
- If you need a member function to offer a stronger guarantee (e.g., destructor must not throw), then you *do* need to mention this.

Coming in the 2020 Standard: **concepts**, which allow you to tell the compiler (some of) your requirements on types.

## Generic Containers

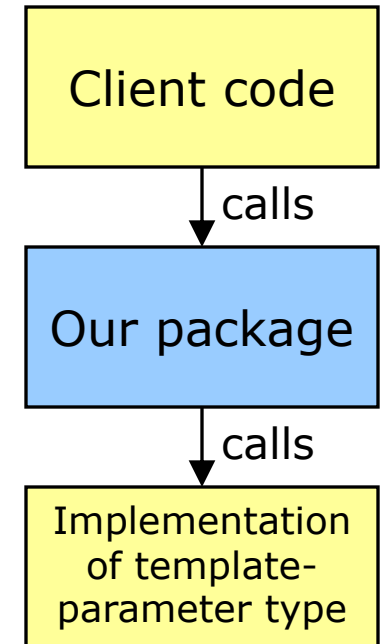
### Exception Safety [1/3]

---

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 more places that might generate exceptions.



↑  
This code  
might throw ...

## Generic Containers

### Exception Safety [2/3]

---

Since *every* member function of a template parameter type may throw (unless it is specifically prohibited from throwing), we need to check *every* use of such a member function, to make sure that we deal with them correctly.

Do not forget:

- Functions that are implicitly called (default ctor, copy ctor, etc.).
- Operators (in particular: assignment).
- STL algorithms. Those that modify a dataset (`std::copy`, `std::rotate`, etc.) generally use 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.

`size_type size() const;`

Returned by value. Copy ctor call?

Yes, but `size_type` is `std::size_t`, a built-in type; its operations will not throw.

## Generic Containers

### Exception Safety [3/3]


---

One tricky situation is copying the data in a dynamic array, since copy assignment of a class type might throw.

Suppose that, if an error occurs, we need to deallocate the dynamic array created below.

```
arr = new MyType[size];  
copy(begin(x), end(x), arr);
```

If `MyType` copy assignment throws, then we have a memory leak!



We will come back to this example shortly.



# Generic Containers

## Exception Neutrality [1/2]

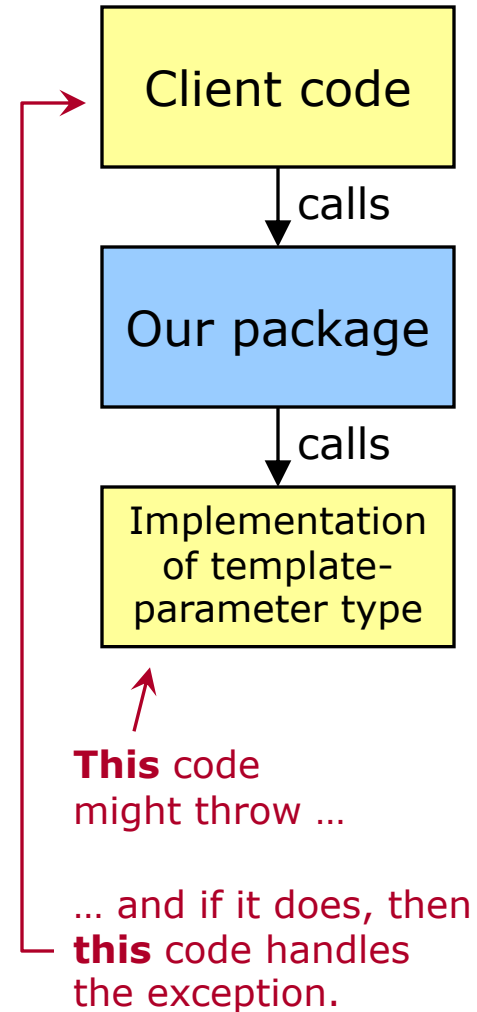
When we call client-provided functions, the client code generally needs to handle any exceptions.

Code is **exception-neutral** if it allows exceptions thrown by client-provided code to propagate unchanged to the caller.

This is our last exception-related terminology.

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.



# Generic Containers

## Exception Neutrality [2/2]

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  
{  
    copy(begin(x), end(x), arr);  
}  
catch (...)  
{  
    delete [] arr;  
    throw;  
}
```

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

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

This helps us meet the Basic Guarantee—and also the Strong Guarantee, if this function does nothing else.

This makes our code exception-neutral.

---

# Thoughts on Project 5

## Thoughts on Project 5

### Introduction

---

This ends the material relevant to Project 5.

Next, we will look at node-based containers. You will *not* need to use those ideas in Project 5.

In Project 5, you will turn the smart array we have been writing in class into a generic container implemented as a class template.

Error handling will get trickier, since now things like assignment of the value type may throw.

- This means that STL algorithms like `std::copy` and `std::rotate` may throw.

# Thoughts on Project 5

## Important Ideas

---

### Things to pay attention to on Project 5:

- The Coding Standards
  - Document everything properly.
- Exception Safety
  - Are your member functions offering the proper guarantee?
    - All member functions must make *at least* the Basic Guarantee.
    - Constructors generally need to make the Strong Guarantee.
    - Destructors, move operations, and `swap` must make the No-Throw Guarantee.
    - Functions that do more complex modifications (`resize`, `insert`, `erase`) *might* not offer the Strong Guarantee, for the sake of efficiency.
  - Do member functions satisfy their guarantees?
    - Check *every* operation that might throw!
    - For a class template, this includes things like `std::copy`, `std::rotate`.
- Allocation & Efficiency
  - Are functions that might need to reallocate-and-copy (`resize`, `insert`, `push_back`) written to handle this efficiently?
- Generic Containers
  - Are all functions exception-neutral?

## Thoughts on Project 5

### Info on Slides

---

There is information on past slides that is relevant to writing member function `swap`, along with the copy ctor, the copy assignment operator, and the move assignment operator. See the slides for:

- *Invisible Functions II*
- *Exception Safety: Commit Functions*
- *Generic Containers: Exception Neutrality*

## Thoughts on Project 5

### Writing `resize`

---

Member function `resize` needs to allow for amortized constant-time insert-at-end.

One way to implement `resize`:

- If resizing to  $\leq$  capacity: just set `_size` to the new value.
- If resizing to  $>$  capacity:
  - We need to compute *newSize*, *newCapacity*, *newData*.
  - *newSize* is given.
  - *newCapacity* should be at least twice the old capacity, at least *newSize*, and at least the minimum capacity.
  - *newData* is allocated and then copied to, using `std::copy`.
  - What if `std::copy` fails? Be sure any necessary clean-up is done.
  - If the copy succeeds, swap each data member with its new value, using `std::swap`. Then be sure the old data is cleaned up properly.

Alternatively, instead of using three variables (*newSize*, *newCapacity*, *newData*), create a temporary object, and use member function `swap` to do the commit. If you do this, then be careful to set the size and capacity correctly!

## Thoughts on Project 5

### Writing insert & erase [1/4]

---

Member functions `insert` and `erase` both resize the client's array.

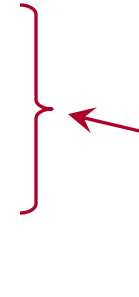
- `insert` resizes it one larger.
- `erase` resizes it one smaller.

Q. How do we do a resize operation?

A. Call member function `resize`.

For `insert`: resize larger, then move items up,  
putting the new item in its proper place.

For `erase`: move items down, then resize smaller.



Note the reversal in the order  
in which things are done.



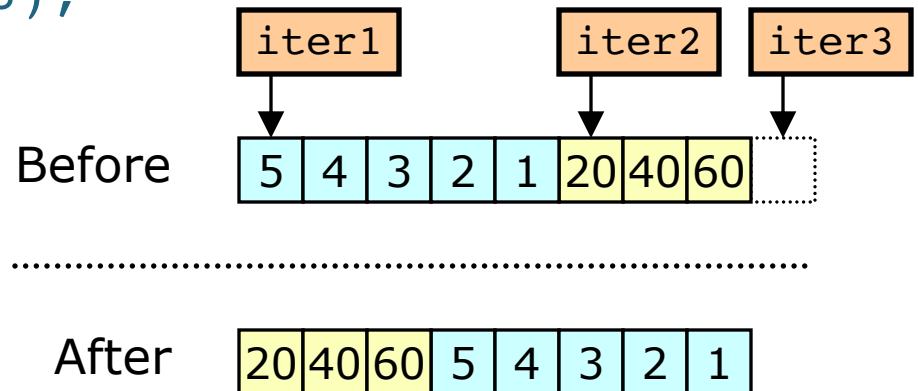
## Thoughts on Project 5

### Writing insert & erase [2/4]

Useful when writing insert and erase:

`std::rotate (<algorithm>)` takes three iterators, specifying two consecutive ranges. It interchanges the two ranges.

```
rotate(iter1, iter2, iter3);
```



Suppose one of the two ranges contains just one item. Then a call to `std::rotate` will move each item in the *other* range down or up one spot, which is what you want to do in insert and erase.

## Thoughts on Project 5

### Writing insert & erase [3/4]

---

Member functions `insert` and `erase` return iterators.

- `insert` returns an iterator to the item inserted.
- `erase` returns an iterator to the item just past the erased item.

For `erase`, this is easy: return the same iterator that was given.

But `insert` resizes the array to make it larger. So it *might* do a reallocate-and-copy. Be sure you return an iterator to the new array, not the old one.

- Hint. The given iterator and the returned iterator point to items with the same index. Save the *index* before resizing. After resizing, construct the new iterator from the saved index, and return it.
- An **iterator** is a pointer (in this project). An **index** is a number, of type `size_type`.

## Thoughts on Project 5

### Writing insert & erase [4/4]

---

Member functions `insert` and `erase` may make *small* changes to the container.

And we want them to be fast. In particular, `insert` must be amortized constant-time when inserting at the end.

This means that the commit-function idea is too slow to use when implementing these functions.

Consider *not* offering the Strong Guarantee in member functions `insert` and `erase` (and `resize`).