

# Allocation & Efficiency

## Generic Containers

### Thoughts on Assignment 5

---

CS 311 Data Structures and Algorithms  
Lecture Slides  
Wednesday, October 23, 2024

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

© 2005–2024 Glenn G. Chappell  
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
  - ✓ ■ 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 topics (from 'Interface for a smart array' to 'Exception safety') under the label 'Smart Arrays'. Another brace on the right groups the next three topics ('Node-based structures', 'More on Linked Lists', and 'Sequences in the C++ STL') under the label 'Linked Lists'. The topics 'Allocation & efficiency', 'Generic containers', 'Stacks', and 'Queues' are not grouped.

---

# Review

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.

Class: `FSArray` (Frightfully Smart Array).

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

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

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` ints. **Added:** ... UNLESS `_size == 0`, in which case `_data` *may be* `nullptr`.

We are  
implementing  
a Sequence as  
a smart array.

A flaw in this design has  
been found and fixed.  
But there is another flaw.

**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 function 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 function never throws an exception.

← Required in some special situations

Each guarantee includes the earlier guarantees.

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.hpp`, `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.

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



**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.

A non-throwing swap member function can be a useful commit function.

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

---

# Allocation & Efficiency

# Allocation & Efficiency Problem

Consider how `FSArray::resize` would work.

If the new size is larger than the current size, then we are out of space, so we must:

- Allocate a new memory block big enough to hold the resized array.
- Copy all array items to their new locations.
- Increase `_size` to the new value.

We refer to the above as **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`.

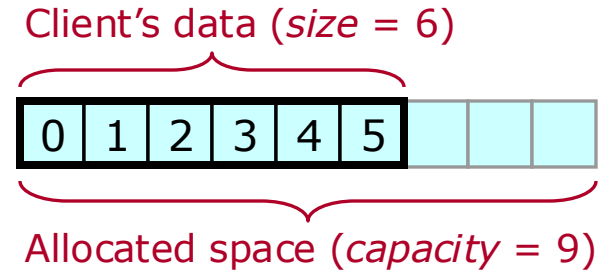
This is the other design flaw mentioned earlier.

## Allocation & Efficiency

### Amortized Constant Time [1/4]

A well designed smart array can allocate extra space beyond the client's data. The total allocated space is the **capacity**.

Insert-at-end is constant-time when more allocated space is available. It is still linear-time in general, due to the possible reallocate-and-copy.



**Idea.** Insert-at-end is fast *most of the time* if we reallocate rarely. When we reallocate, get more memory than we need—perhaps twice as much. 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. *See the next slide.*

# Allocation & Efficiency

## Amortized Constant Time [2/4]

Count the basic operations required for repeated insert-at-end.

Begin with size 1 and capacity 2.

Suppose that reallocate-and-copy always doubles the capacity.

To simplify things, our only basic operation will be copying an item.

Note that the total number of basic operations for all inserts is always at most 3 times the number of inserts.

This suggests (but does not prove):

Q. How much time is required for  $k$  insert-at-end operations?

A.  $O(k)$ —if reallocate-and-copy ups allocated space by a constant **factor**.

Even though a single insert is not  $O(1)$ .

Size	Capacity	Insert #BOs	Total #BOs
1	2	1	1
2	2	$2+1 = 3$	4
3	4	1	5
4	4	$4+1 = 5$	10
5	8	1	11
6	8	1	12
7	8	1	13
8	8	$8+1 = 9$	22
9	16	1	23
10	16	1	24
11	16	1	25
12	16	1	26
13	16	1	27
14	16	1	28
15	16	1	29
16	16	$16+1 = 17$	46

## Allocation & Efficiency

### Amortized Constant Time [3/4]

---

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 [4/4]

---

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.

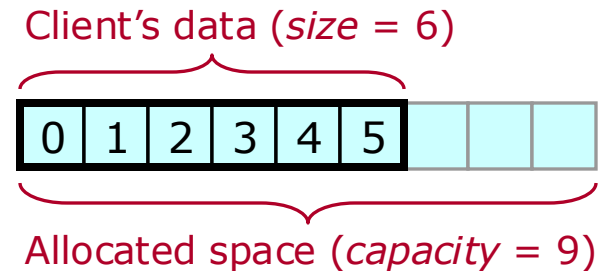
Insert-at-end for a well written resizable array is amortized constant-time. It is also still linear time.

# 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.



Finish the details of this revised design.

- ~~Two~~ 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: *newCapacity* is at least  $2 \times \text{capacity}$ , and at least *newSize*.



# Allocation & Efficiency

## Array Redesign — CODE

### Internal redesign:

- Three data members: `_capacity`, `_size`, `_data`.
- `0 ≤ _size ≤ _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, *newCapacity* should be at least  $2 \times \text{capacity}$ , and also at least *newSize*.

Client's data (size = 6)



Allocated space (capacity = 9)

### TO DO

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

*Done. See `fsarray.hpp` & `fsarray.cpp`. This is the last change I expect to make to this code.*

---

# Generic Containers

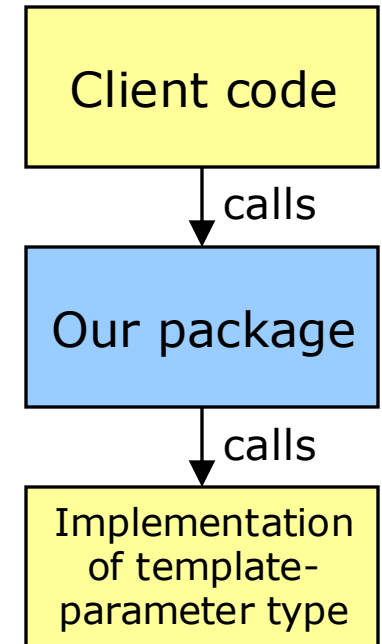
## Generic Containers

### 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.

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`, 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?

`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 exception is thrown, 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]

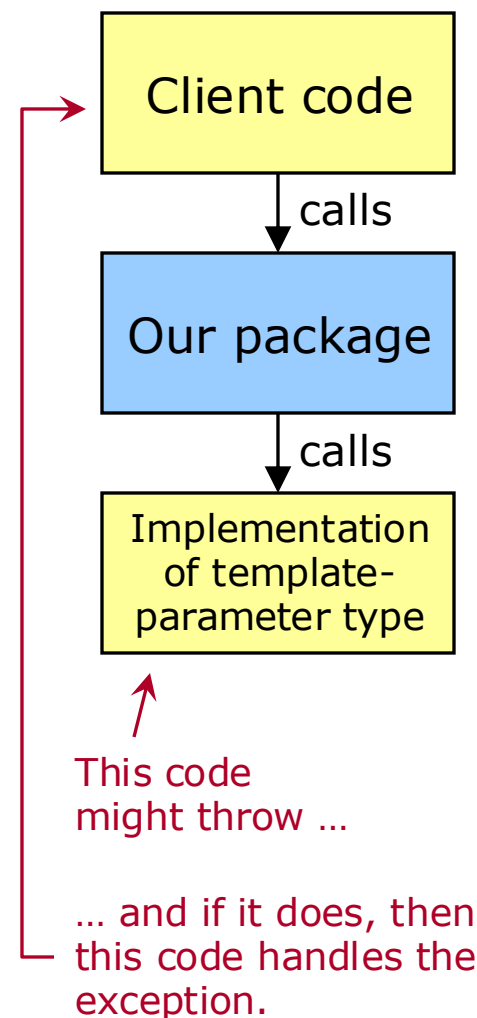
When we call client-provided functions, the client code generally needs to handle any exceptions.

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 Assignment 5



## Thoughts on Assignment 5

### Introduction

---

This ends the material relevant to Assignment 5.

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

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

In all future assignments in this class—including Assignment 5—you may *optionally* work in a group of two (but not three!). Each group should only turn in one copy of the assignment files. See the various assignment descriptions for details.

# Thoughts on Assignment 5

## Important Ideas

---

### Things to pay attention to on Assignment 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 functions, 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 Assignment 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 Assignment 5

### Writing `resize`

---

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

How I wrote `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, then deallocate the old data, and set each of the data members to its new value.

## Thoughts on Assignment 5

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

---

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

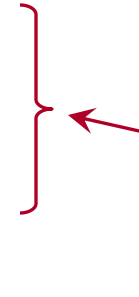
- `insert` resizes it one larger.
- `erase` resizes it one smaller.

Q. How do we resize the array?

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.

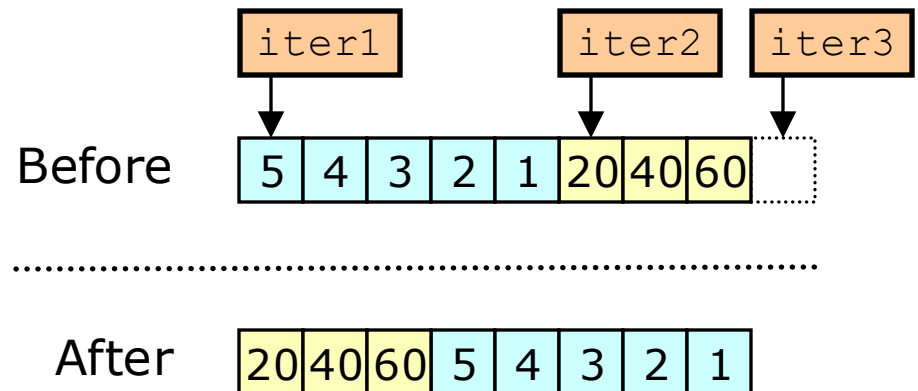
## Thoughts on Assignment 5

### 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 Assignment 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 assignment). An **index** is a number, of type `size_type`.

## Thoughts on Assignment 5

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

---

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

And we want small changes 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.

So do *not* offer the Strong Guarantee in member functions `insert` and `erase`.