Nimbly Documentation

Release 2.0.0

MousePaw Media

Jun 17, 2022

CONTENTS:

Nimbly provides performance-minded alternatives to many standard library classes. It focuses on minimizing CPU usage first, and memory second, while leaving you in control. Nimbly is designed to shine in low-resource environments.

CHAPTER

ONE

FLEXARRAY

1.1 What is FlexArray?

FlexArray is a flexibly-sized array similar to `std::vector`. Internally, it is implemented as a circular buffer deque, guaranteed to be stored in contiguous memory, thereby helping to avoid or minimize cache misses.

While we aim to create a high-performance data structure, our top priority is in giving the user easy *control* over the tradeoffs between CPU performance, memory, and cache misses.

1.1.1 Performance

FlexArray is usually as fast as, or faster than, `std::vector`. Unlike `std::deque`, FlexArray is guaranteed to be stored in contiguous memory.

Here's how FlexArray stacks up against the GCC implementation of `std::vector`...

- Inserting to end is as fast or faster.
- Inserting to the middle is *slower*. (We plan to improve this in a later release.)
- Inserting to the beginning is faster.
- Removing from any position is faster.
- Accessing any position is as fast.

If general performance is more important to you than contiguous memory, see `SpeedList`.

1.1.2 Functional Comparison to `std::vector`

FlexArray offers largely the same functionality as `std::vector`. However, it is not intended to feature-identical. Some functionality hasn't been implemented yet, and we may not include some other features to leave room for future optimization and experimentation.

- FlexArray does not offer iterators. This *may* be added in the future.
- You cannot change the underlying data structure. Our base class is where most of the heavy lifting occurs.
- Some advanced modifiers haven't been implemented yet.

1.1.3 Technical Limitations

FlexArray can store a maximum of 4,294,967,294 objects. This is because it uses 32-bit unsigned integers for internal indexing, with the largest value reserved as `INVALID_INDEX`. The limit is calculated as follows.

`2^{32} - 2 = 4,294,967,294`

1.2 Using FlexArray

1.2.1 Including FlexArray

To include FlexArray, use the following:

```
#include "nimbly/flexarray.hpp"
```

1.2.2 Creating a FlexArray

A `FlexArray` object is created by whatever means is convenient. It handles its own dynamic allocation for storing its elements.

When the FlexArray is first created, you must specify the type of its elements.

```
// Both of these methods are valid...
FlexArray<int> temps_high;

FlexArray<int>* temps_low = new FlexArray<int>;
```

Raw Copy

By default, FlexArray uses standard assignment for moving items when the internal data structure resizes. However, if you're storing atomic data types, such as integers, additional performance gains may be achieved by having FlexArray use raw memory copying (*memcpy*) instead.

To switch to Raw Copy Mode, include `true` as the second template parameter (`raw_copy`).

```
FlexArray<int, true> i_use_rawcopy;
```

Resize Factor

To minimize the number of CPU cycles used on reallocation, when we run out of space in the data structure, on the next insertion, we allocate more space than we immediately need. This *resize factor* is controllable.

By default, when the FlexArray resizes, it **doubles** its capacity (`n * 2`). This provides the best general performance. However, if you want to preserve memory at a small performance cost, you can switch to a resize factor of `n * 1.5` (internally implemented as `n + n / 2`).

To switch to the `1.5` factor, include `false` as the third template parameter (`factor_double`).

```
FlexArray<int, true, false> i_resize_slower;
```

Reserve Size

We can specify the initial size (in elements) of the FlexArray in the constructor.

```
FlexArray<int>* temps_high = new FlexArray<int>(100);
```

Note: The FlexArray will always have minimum capacity of 2.

1.2.3 Adding Elements

You can insert an element anywhere into a FlexArray. As with `std::vector`, the first element is considered the "front", and the last element the "back".

`insert()`

It is possible to insert an element anywhere in the array using `insert()`. This function has a worst-case performance of `O(n/2)`.

```
FlexArray<int> temps;

// We'll push a couple of values for our example.
temps.push(45);
temps.push(48);

// Insert the value "37" at index 1.
temps.insert(37, 1);
// Insert the value "35" at index 2.
temps.insert(35, 2);

// The FlexArray is now [48, 35, 37, 45]
```

If there is ever a problem adding a value, the function will return `false`. Otherwise, it will return `true`.

`push()`

The most common action is to "push" an element to the back using the `push()` function. The alias `push_back()` is also provided for convenience.

In FlexArray, `push()` has exactly the same performance as `shift()`; that is, `O(1)`.

```
FlexArray<int> temps_high;
temps_high.push(45);
temps_high.push(37);
temps_high.push(35);
temps_high.push_back(48); // we can also use push_back()
// The FlexArray is now [45, 37, 35, 48]
```

If there is ever a problem adding a value, the function will return `false`. Otherwise, it will return `true`.

shift()

You can also "shift" an element to the front using `shift()`. The alias `push_front()` is also provided.

In FlexArray, `shift()` has exactly the same performance as `push()`; that is, `O(1)`.

```
FlexArray<int> temps_low;
temps_low.shift(45);
temps_low.shift(37);
temps_low.shift(35);
temps_low.push_front(48); // we can also use push_front()
// The FlexArray is now [48, 35, 37, 45]
```

If there is ever a problem adding a value, the function will return `false`. Otherwise, it will return `true`.

1.2.4 Accessing Elements

at()

`at()` allows you to access the value at a given array index.

```
FlexArray<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(42);
apples.push(36);

apples.at(1);

// This output yields 42
```

Alternatively, you can use the `[]` operator to access a value.

```
// Using the array from above...
apples[2];

// The array is [23, 42, 36]
// This output yields 36
```

Warning: If the array is empty, or if the specified index is too large, this function/operator will throw the exception `std::out_of_range`.

peek()

peek() allows you to access the last element in the array without modifying the data structure. The alias peek_back() is also provided for convenience.

```
FlexArray<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(42);
apples.push(36);

apples.peek();
// This outputs 36.
// The array remains [23, 42, 36]
```

Warning: If the array is empty, this function will throw the exception std::out_of_range.

If you want to "peek" the first element, use peek_front().

peek_front()

peek_front() allows you to access the first element in the array without modifying the data structure.

```
FlexArray<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(42);
apples.push(36);

apples.peek_front();
// This outputs 23.
// The array remains [23, 42, 36]
```

Warning: If the array is empty, this function will throw the exception std::out_of_range.

1.2.5 Removing Elements

clear()

clear() removes all the elements in the FlexArray.

```
FlexArray<int> pie_sizes;

pie_sizes.push(18);
pie_sizes.push(18);
pie_sizes.push(15);
```

(continues on next page)

(continued from previous page)

```
// I ate everything...
pie_sizes.clear();
```

This function always returns true, and will never throw an exception (**no-throw guarantee**).

erase()

`erase()` allows you to delete elements in an array in a given range. Remaining values are shifted to fill in the empty slot. This function has a worst-case performance of `O(n/2)`.

```
FlexArray<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(42);
apples.push(36);

// The array is currently [23, 42, 36]

apples.erase(0,1);
// The first number in the function call is the lower bound
// The second number is the upper bound.
// The array is now simply [36]
```

If any of the indices are too large, this function will return `false`. Otherwise, it will return true. It never throws exceptions (**no-throw guarantee**).

pop()

`pop()` returns the last value in an array, and then removes it from the data set. The alias `pop_back()` is also provided. In FlexArray, `pop()` has exactly the same performance as `unshift()`; that is, `O(1)`.

```
FlexArray<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(42);
apples.push(36);

// The array is currently [23, 42, 36]

apples.pop(0,1);
// Returns 3. The array is now [23, 42]
```

Warning: If the array is empty, this function will throw the exception `std::out_of_range`.

unshift()

unshift() will return the first element in the array, and remove it. In FlexArray, unshift() has exactly the same performance as pop(); that is, O(1).

```
FlexArray<int> apples;

// We'll push some values for our example
apples.push(2);
apples.push(1);
apples.push(3);

// The array is currently [23, 42, 36]

apples.unshift();
// Returns 23.
// The array is now [42, 36]
```

Warning: If the array is empty, this function will throw the exception std::out_of_range.

yank()

yank() removes a value at a given index. Remaining values are shifted to fill in the empty slot. This function has a worst-case performance of O(n/2).

```
FlexArray<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(42);
apples.push(36);

// The array is currently [23, 42, 36]

apples.yank(1);
// Returns 42.
// The array is now [23, 36]
```

Warning: If the array is empty, or if the specified index is too large, this function will throw the exception std::out_of_range.

1.2.6 Size and Capacity Functions

`getCapacity()`

`getCapacity()` returns the total number of elements that can be stored in the FlexArray without resizing.

```
FlexArray<int> short_term_memory;

short_term_memory.getCapacity();
// Returns 8, the default size.
```

`length()`

`length()` allows you to check how many elements are currently in the FlexArray.

```
FlexArray<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(42);
apples.push(36);

apples.length();
// The function will return 3
```

`isEmpty()`

`isEmpty()` returns true if the FlexArray is empty, and false if it contains values.

```
FlexArray<int> answers;

answers.isEmpty();
// The function will return true

// We'll push some values for our example
answers.push(42);

answers.isEmpty();
// The function will return false
```

`isFull()`

`isFull()` returns true if the FlexArray is full to the current capacity (before resizing), and false otherwise.

```
FlexArray<int> answers;

answers.isFull();
// The function will return false

// Push values until we are full, using the isFull() function to check.
```

(continues on next page)

(continued from previous page)

```
while(!answers.isFull())
{
    answers.push(42);
}
```

reserve()

You can use `reserve()` to resize the FlexArray to be able to store the given number of elements. If the data structure is already equal to or larger than the requested capacity, nothing will happen, and the function will return `false`.

```
FlexArray<std::string> labors_of_hercules;

// Reserve space for all the elements we plan on storing.
labors_of_hercules.reserve(12);

labors_of_hercules.getCapacity();
// Returns 12, the requested capacity.
```

After reserving space in an existing FlexArray, it can continue to resize.

This function is effectively identical to specifying a size at instantiation.

shrink()

You can use `shrink()` function to resize the FlexArray to only be large enough to store the current number of elements in it. If the shrink is successful, it wil return `true`, otherwise it will return `false`.

```
FlexArray<int> marble_collection;

for(int i = 0; i < 100; ++i)
{
    marble_collection.push(i);
}

marble_collection.getCapacity();
// Returns 128, because FlexArray is leaving room for more elements.

// Shrink to only hold the current number of elements.
marble_collection.shrink();

marble_collection.getCapacity();
// Returns 100, the same as the number of elements.
```

After shrinking, we can continue to resize as new elements are added.

Note: It is not possible to shrink below a capacity of 2.

CHAPTER

TWO

FLEXQUEUE

2.1 What is FlexQueue?

FlexQueue is a flexibly-sized queue similar to `std::queue`. Internally, it is implemented as a circular buffer deque, guaranteed to be stored in contiguous memory, thereby helping to avoid or minimize cache misses.

While we aim to create a high-performance data structure, our top priority is in giving the user easy *control* over the tradeoffs between CPU performance, memory, and cache misses.

2.1.1 Performance

Because `std::queue` is based on `std::deque`, and thereby is not stored in contiguous memory, we instead must benchmark `FlexQueue` against `std::vector`.

FlexQueue is usually as fast as, or faster than, `std::vector`. Here's how FlexQueue ranks against the GCC implementation of `std::vector`…

- Pushing (to back) is faster.
- Popping (from front) is faster.
- Accessing is at least as fast.

If general performance is more important to you than contiguous memory, see `SpeedQueue`.

2.1.2 Comparison to `std::queue`

FlexQueue offers largely the same functionality as `std::queue`. However, it is not intended to feature-identical. Some functionality hasn't been implemented yet, and we may not include some other features to leave room for future optimization and experimentation.

- FlexQueue does not offer iterators. This *may* be added in the future.
- You cannot change the underlying data structure. Our base class is where most of the heavy lifting occurs.
- Some advanced modifiers haven't been implemented yet.

2.1.3 Technical Limitations

FlexQueue can store a maximum of 4,294,967,294 objects. This is because it uses 32-bit unsigned integers for internal indexing, with the largest value reserved as `INVALID_INDEX`. The limit is calculated as follows.

```
2^{32} - 2 = 4,294,967,294
```

2.2 Using FlexQueue

Queues are "First-In-First-Out"; you insert to the end (or "back"), and remove from the front.

2.2.1 Including FlexQueue

To include FlexQueue, use the following:

```
#include "nimbly/flexqueue.hpp"
```

2.2.2 Creating a FlexQueue

A `FlexQueue` object is created by whatever means is convenient. It handles its own dynamic allocation for storing its elements.

When the FlexQueue is first created, you must specify the type of its elements.

```
// Both of these methods are valid...

FlexQueue<int> dmvLine;

anotherQueue = new FlexQueue<int>;
```

Raw Copy

By default, FlexQueue uses standard assignment for moving items when the internal data structure resizes. However, if you're storing atomic data types, such as integers, additional performance gains may be achieved by having FlexQueue use raw memory copying (*memcpy*) instead.

To switch to Raw Copy Mode, include `true` as the second template parameter (`raw_copy`).

```
FlexQueue<int, true> i_use_rawcopy;
```

Resize Factor

When we run out of space in the data structure, we need to reallocate memory. To reduce the CPU cycles used on reallocation, we allocate more space than we immediately need. This *resize factor* is controllable.

By default, when the FlexQueue resizes, it **doubles** its capacity (`n * 2`). This provides the best general performance. However, if you want to preserve memory at a small performance cost, you can switch to a resize factor of `n * 1.5` (internally implemented as `n + n / 2`).

To switch to the `1.5` factor, include `false` as the third template parameter (`factor_double`).

```
FlexQueue<int, true, false> i_resize_slower;
```

Reserve Size

We can specify the initial size (in elements) of the FlexQueue in the constructor.

```
FlexQueue<int>* dmvLine = new FlexQueue<int>(250);
```

Note: The FlexQueue will always have minimum capacity of 2.

2.2.3 Adding Elements

enqueue()

`enqueue()` adds a value to the end of the queue. Aliases `push()` and `push_back()` are also provided. This function has the performance of `O(1)`.

```
FlexQueue<int> apples;

// We'll add some values
// using the three aliases
apples.enqueue(23);
apples.push(12);
apples.push_back(31);

// The queue is now [23, 12, 31]
```

If there is ever a problem adding a value, the function will return `false`. Otherwise, it will return `true`.

2.2.4 Accessing Elements

at()

`at()` allows you to access the value at a given index.

```
FlexQueue<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(12);
apples.push(31);

apples.at(1);

// The queue is [23, 12, 31]
// This output yields 12
```

Alternatively, you can use the `[]` operator to access a value.

```
// Using the queue from above...

apples[2];

// The queue is [23, 12, 31]
// This output yields 31
```

Warning: If the queue is empty, or if the specified index is too large, this function/operator will throw the exception `std::out_of_range`.

`peek()`

`peek()` allows you to access the next element in the queue without modifying the data structure.

```
FlexQueue<int> apples;

// We'll push some values for our example
apples.push(23);
apples.push(12);
apples.push(31);

std::cout << apples.peek();

// This output yields 23
// The queue remains [23, 12, 31]
```

Warning: If the queue is empty, this function will throw the exception `std::out_of_range`.

2.2.5 Removing Elements

In a queue, we typically remove and return elements from the beginning, or “front” of the queue. Imagine a line at a grocery store - you enter in the back and exit in the front.

`clear()`

`clear()` removes all the elements in the FlexQueue.

```
FlexQueue<int> pie_sizes;

pie_sizes.push(18);
pie_sizes.push(18);
pie_sizes.push(15);

// I ate everything...
pie_sizes.clear();

// The FlexQueue is now empty.
```

This function always returns true, and will never throw an exception (**no-throw guarantee**).

dequeue()

`dequeue()` will remove and return the first element in the queue. Aliases `pop()` and `pop_front()` are also provided. This function has the performance of `O(1)`.

```
FlexQueue<int> apples;

// We'll push some values
apples.push(23);
apples.push(12);
apples.push(31);
apples.push(40);

// The queue is now [23, 12, 31, 40]

// We'll now remove three elements
// with the three provided aliases
apples.dequeue();
apples.pop();
apples.pop_front();

// The queue is now simply [40]
```

Warning: If the queue is empty, this function will throw the exception `std::out_of_range`.

erase()

`erase()` allows you to delete elements in a queue in a given range. Remaining values are shifted to fill in the empty slot. This function has a worst-case performance of `O(n/2)`.

```
FlexQueue<std::string> coffeeshop_line;

// We'll push some values for our example
coffeeshop_line.enqueue("Bob");
coffeeshop_line.enqueue("Jane");
coffeeshop_line.enqueue("Alice");

// The queue is currently ["Bob", "Jane", "Alice"]

apples.erase(0,1);
// The first number in the function call is the lower bound
// The second number is the upper bound.
// The queue is now simply ["Alice"]
```

If any of the indices are too large, this function will return `false`. Otherwise, it will return true. It never throws exceptions (**no-throw guarantee**).

2.2.6 Size and Capacity Functions

`getCapacity()`

`getCapacity()` returns the total number of elements that can be stored in the FlexQueue without resizing.

```
FlexQueue<int> short_term_memory;

short_term_memory.getCapacity();
// Returns 8, the default size.
```

`length()`

`length()` allows you to check how many elements are currently in the FlexQueue.

```
FlexQueue<int> apples;

// We'll enqueue some values for our example
apples.enqueue(23);
apples.enqueue(42);
apples.enqueue(36);

apples.length();
// The function will return 3
```

`isEmpty()`

`isEmpty()` returns true if the FlexQueue is empty, and false if it contains values.

```
FlexQueue<int> answers;

answers.isEmpty();
// The function will return true

// We'll enqueue some values for our example
answers.enqueue(42);

answers.isEmpty();
// The function will return false
```

`isFull()`

`isFull()` returns true if the FlexQueue is full to the current capacity (before resizing), and false otherwise.

```
FlexQueue<int> answers;

answers.isFull();
// The function will return false

// Push values until we are full, using the isFull() function to check.
```

(continues on next page)

(continued from previous page)

```
while(!answers.isFull())
{
    answers.enqueue(42);
}
```

reserve()

You can use `reserve()` to resize the FlexQueue to be able to store the given number of elements. If the data structure is already equal to or larger than the requested capacity, nothing will happen, and the function will return `false`.

```
FlexQueue<std::string> labors_of_hercules;

// Reserve space for all the elements we plan on storing.
labors_of_hercules.reserve(12);

labors_of_hercules.getCapacity();
// Returns 12, the requested capacity.
```

After reserving space in an existing FlexQueue, it can continue to resize.

This function is effectively identical to specifying a size at instantiation.

shrink()

You can use `shrink()` function to resize the FlexQueue to only be large enough to store the current number of elements in it. If the shrink is successful, it wil return `true`, otherwise it will return `false`.

```
FlexQueue<int> marble_collection;

for(int i = 0; i < 100; ++i)
{
    marble_collection.enqueue(i);
}

marble_collection.getCapacity();
// Returns 128, because FlexQueue is leaving room for more elements.

// Shrink to only hold the current number of elements.
marble_collection.shrink();

marble_collection.getCapacity();
// Returns 100, the same as the number of elements.
```

After shrinking, we can continue to resize as new elements are added.

Note: It is not possible to shrink below a capacity of 2.

CHAPTER

THREE

FLEXSTACK

3.1 What is FlexStack?

FlexStack is a flexibly-sized stack similar to `std::stack<T, std::vector>`. Internally, it is implemented as a circular buffer deque, guaranteed to be stored in contiguous memory, thereby helping to avoid or minimize cache misses.

While we aim to create a high-performance data structure, our top priority is in giving the user easy *control* over the tradeoffs between CPU performance, memory, and cache misses.

3.1.1 Performance

FlexStack is slightly slower than the typical `std::stack`, but this is acceptable because of the inherent difference between Flex and `std::deque`; while Flex guarantees storage in contiguous memory, `std::deque` does not. As a result, we instead must compare against `std::stack<T, std::vector>`.

FlexStack is usually as fast as, or faster than, `std::stack<T, std::vector>`. Here's how FlexStack ranks against the GCC implementation of `std::stack`, with `std::vector` as its underlying container…

- Pushing (to back) is as fast.
- Popping (from back) is faster.
- Accessing is at least as fast.

If general performance is more important to you than contiguous memory, see `SpeedStack`.

3.1.2 Comparison to `std::stack`

FlexStack offers largely the same functionality as `std::stack`. However, it is not intended to feature-identical. Some functionality hasn't been implemented yet, and we may not include some other features to leave room for future optimization and experimentation.

- FlexStack does not offer iterators. This *may* be added in the future.
- You cannot change the underlying data structure. Our base class is where most of the heavy lifting occurs.
- Some advanced modifiers haven't been implemented yet.

3.1.3 Technical Limitations

FlexStack can store a maximum of 4,294,967,294 objects. This is because it uses 32-bit unsigned integers for internal indexing, with the largest value reserved as `INVALID_INDEX`. The limit is calculated as follows.

```
2^{32} - 2 = 4,294,967,294
```

3.2 Using FlexStack

3.2.1 Including FlexStack

To include FlexStack, use the following:

```
#include "nimbly/flexstack.hpp"
```

3.2.2 Creating a FlexStack

A `FlexStack` object is created by whatever means is convenient. It handles its own dynamic allocation for storing its elements.

When the FlexStack is first created, you must specify the type of its elements.

```
// Both of these methods are valid...
FlexStack<int> dish_sizes;

FlexStack<int>* dish_sizes = new FlexStack<int>;
```

Raw Copy

By default, FlexStack uses standard assignment for moving items when the internal data structure resizes. However, if you're storing atomic data types, such as integers, additional performance gains may be achieved by having FlexStack use raw memory copying (*memcpy*) instead.

To switch to Raw Copy Mode, include `true` as the second template parameter (`raw_copy`).

```
FlexStack<int, true> i_use_rawcopy;
```

Resize Factor

When we run out of space in the data structure, we need to reallocate memory. To reduce the CPU cycles used on reallocation, we allocate more space than we immediately need. This *resize factor* is controllable.

By default, when the FlexStack resizes, it **doubles** its capacity (`n * 2`). This provides the best general performance. However, if you want to preserve memory at a small performance cost, you can switch to a resize factor of `n * 1.5` (internally implemented as `n + n / 2`).

To switch to the `1.5` factor, include `false` as the third template parameter (`factor_double`).

```
FlexStack<int, true, false> i_resize_slower;
```

Reserve Size

We can specify the initial size (in elements) of the FlexStack in the constructor.

```
FlexStack<int>* temps_high = new FlexStack<int>(100);
```

Note: The FlexStack will always have minimum capacity of 2.

3.2.3 Adding Elements

Stacks are “Last-In-First-Out”; you insert to the end (or “back”), and remove from the back.

`push()`

We add new elements to the stack with a “push” to the back using the `push()` function. The alias `push_back()` is also provided for convenience. This function has a performance of `O(1)`.

```
FlexStack<int> dish_sizes;
dish_sizes.push(22);
dish_sizes.push(18);
dish_sizes.push(18);
dish_sizes.push_back(12); // we can also use push_back()
// The FlexStack is now [22, 18, 18, 12]
```

3.2.4 Accessing Elements

`at()`

`at()` allows you to access the value at a given stack index.

```
FlexStack<string> albums;

// We'll push some values for our example
albums.push("End Of Silence");
albums.push("Comatose");
albums.push("Fireproof");

albums.at(1);
// This output yields "Comatose"
```

Alternatively, you can use the `[]` operator to access a value.

```
// Using the stack from above...

albums[2];
// This output yields "Fireproof"
```

peek()

peek() allows you to access the next element in the stack without modifying the data structure.

```
FlexStack<string> albums;

// We'll push some values for our example
albums.push("End Of Silence");
albums.push("Comatose");
albums.push("Fireproof");

albums.peek();

// This output yields "Fireproof"
// The stack remains ["End of Silence", "Comatose", "Fireproof"]
```

3.2.5 Removing Elements

In a stack, we typically remove and return elements from the end, or "back" of the stack. Imagine a stack of dishes - the last one added is the first one removed (ergo "last-in-first-out").

clear()

clear() removes all the elements in the FlexStack.

```
FlexStack<int> pie_sizes;

pie_sizes.push(18);
pie_sizes.push(18);
pie_sizes.push(15);

// I ate everything...
pie_sizes.clear();

// The FlexStack is now empty.
```

This function always returns true, and will never throw an exception (**no-throw guarantee**).

erase()

erase() allows you to delete elements in a stack in a given range. Remaining values are shifted to fill in the empty slot. This function has a worst-case performance of O(n/2).

```
FlexStack<string> albums;

// We'll push some values for our example
albums.push("End Of Silence");
albums.push("Comatose");
albums.push("Fireproof");

// The stack is currently ["End of Silence", "Comatose", "Fireproof"]
```

(continues on next page)

(continued from previous page)

```
albums.erase(0, 1);
// The first number in the function call is the lower bound
// The second number is the upper bound.
// The stack is now simply ["Fireproof"]
```

If any of the indices are too large, this function will return `false`. Otherwise, it will return true. It never throws exceptions (**no-throw guarantee**).

`pop()`

`pop()` returns the last value in an stack, and then removes it from the data set. The alias `pop_back()` is also provided. This function has a performance of `O(1)`.

```
FlexStack<int> dish_sizes;

// We'll push some values for our example
dish_sizes.push(22);
dish_sizes.push(18);
dish_sizes.push(12);

// The stack is currently [22, 18, 12]

dish_sizes.pop();
// Returns 12. The stack is now [22, 18]
```

Warning: If the stack is empty, this function will throw the exception `std::out_of_range`.

3.2.6 Size and Capacity Functions

`getCapacity()`

`getCapacity()` returns the total number of elements that can be stored in the FlexStack without resizing.

```
FlexStack<int> short_term_memory;

short_term_memory.getCapacity();
// Returns 8, the default size.
```

`length()`

`length()` allows you to check how many elements are currently in the FlexStack.

```
FlexStack<string> albums;

// We'll push some values for our example
albums.push("End Of Silence");
albums.push("Comatose");
albums.push("Fireproof");

albums.length();
// The function will return 3
```

`isEmpty()`

`isEmpty()` returns true if the FlexStack is empty, and false if it contains values.

```
FlexStack<string> albums;

albums.isEmpty();
// The function will return true

// We'll push some values for our example
albums.push("End Of Silence");
albums.push("Comatose");
albums.push("Fireproof");

albums.isEmpty();
// The function will return false
```

`isFull()`

`isFull()` returns true if the FlexStack is full to the current capacity (before resizing), and false otherwise.

```
FlexStack<int> answers;

answers.isFull();
// The function will return false

// Push values until we are full, using the isFull() function to check.
while(!answers.isFull())
{
    answers.push(42);
}
```

reserve()

You can use `reserve()` to resize the FlexStack to be able to store the given number of elements. If the data structure is already equal to or larger than the requested capacity, nothing will happen, and the function will return `false`.

```
FlexStack<std::string> labors_of_hercules;

// Reserve space for all the elements we plan on storing.
labors_of_hercules.reserve(12);

labors_of_hercules.getCapacity();
// Returns 12, the requested capacity.
```

After reserving space in an existing FlexStack, it can continue to resize.

This function is effectively identical to specifying a size at instantiation.

shrink()

You can use `shrink()` function to resize the FlexStack to only be large enough to store the current number of elements in it. If the shrink is successful, it wil return `true`, otherwise it will return `false`.

```
FlexStack<int> plate_collection;

for(int i = 0; i < 100; ++i)
{
    plate_collection.push(i);
}

plate_collection.getCapacity();
// Returns 128, because FlexStack is leaving room for more elements.

// Shrink to only hold the current number of elements.
plate_collection.shrink();

plate_collection.getCapacity();
// Returns 100, the same as the number of elements.
```

After shrinking, we can continue to resize as new elements are added.

Note: It is not possible to shrink below a capacity of 2.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search