PawLIB Documentation

Release 2.0.0

MousePaw Media

Aug 05, 2021

CONTENTS

The **PawLIB** static library provides various common utility functions, many of which are designed as high-performance, near-drop-in-replacements for common C++ Standard Library (`std::`) and Boost classes.

PawLIB focuses on minimizing CPU usage first, and memory second, and as such it is designed to run on older systems and those with more limited resources.

See PawLIB's `README.md`, `CHANGELOG.md`, `BUILDING.md`, and `LICENSE.md` for more information.

CHAPTER

ONE

CONTENTS

1.1 Setup and Compiling

1.1.1 Building

PawLIB is a static library which needs to be compiled before it can be used.

Note: Complete building instructions can be found in BUILDING.md

Environment and Dependencies

To build PawLIB, you must have the following installed on your system:

- CMake 3.1+ (cmake.org)
- Clang 3.4+ OR GCC 5.3+ (MSVC is not presently supported.)
- CPGF 1.60, from either:
  - libdeps 1.0 (GitHub) [recommended], *or*
  - CPGF (GitHub).

Build Configuration

If you use our `libdeps` repository, it is recommended that you clone it adjacent to the `pawlib` folder. In that scenario, you can skip this section, as the default configuration will work for you.

If you put `libdeps` somewhere else, or provide CPGF yourself, be sure to create a build configuration file.

To do this, make a copy of `build.config.txt` in the root of the PawLIB repository, and name it `build.config`. Optionally, you may replace "build" with any name, so long as the filename ends in `.config`.

Edit the file, and change the following section to point to the location of CPGF's *include* and *lib* directories.

```
set(CPGF_DIR
    ${CMAKE_HOME_DIRECTORY}/../../libdeps/libs
)
```

In that path, the variable `${CMAKE_HOME_DIRECTORY}` refers to either the `pawlib-source/` or `pawlib-tester/` directories in the PawLIB repository, since the same configuration file is used in compiling both.

Save and close.

Compiling PawLIB

Make sure that CPGF is built. If you're using `libdeps`, you can simply run `make cpgf` in that repository.

In the PawLIB repository, run `make ready` to build PawLIB as a static library.

Note: If you experience any errors in compiling, we invite you to contact us. See *Finding Support*.

1.1.2 Linking to PawLIB

To use PawLIB in your C++ project, you will need to statically link to both it and CPGF.

Important: Linking order is important in C++! You should link to PawLIB first, and then its dependencies (namely CPGF) second.

You can point your compiler and linker at the `pawlib/include/` and `pawlib/lib/` directories.

If you need help finding the paths for CPGF, refer to the path in the PawLIB build configuration (see *Build Configuration*). The `include/` and `lib/` directories should be in the path specified.

1.1.3 Using PawLIB

All `#include` commands follow the format `#include "pawlib/HEADER.hpp"`, where `HEADER.hpp` is the header file you wish to include.

1.1.4 Building Tester

If you want to use the PawLIB tester application, you can build that as well. Once you've confirmed that PawLIB itself can build without errors, run `make tester` to build the tester.

1.2 FlexArray

1.2.1 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.

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`.

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.

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`

1.2.2 Using FlexArray

Including FlexArray

To include FlexArray, use the following:

```
#include "pawlib/flex_array.hpp"
```

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.

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`.

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`.

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);

// 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);
```

(continues on next page)

(continued from previous page)

```
// 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`.

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.
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.

1.3 FlexQueue

1.3.1 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.

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`.

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.

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`

1.3.2 Using FlexQueue

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

Including FlexQueue

To include FlexQueue, use the following:

```
#include "pawlib/flex_queue.hpp"
```

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.

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`.

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
```

(continues on next page)

(continued from previous page)

```
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`.

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**).

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.
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.

1.4 FlexStack

1.4.1 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.

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`.

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.

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`

1.4.2 Using FlexStack

Including FlexStack

To include FlexStack, use the following:

```
#include "pawlib/flex_stack.hpp"
```

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.

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]
```

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");
```

(continues on next page)

(continued from previous page)

```
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"]
```

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"]

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.

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.

1.5 Trilean

1.5.1 What Is It?

A trilean is an atomic data type with three distinct states: "true", "maybe", and "false". It is designed to behave like a boolean, but with the additional feature of "uncertainty."

In order to understand trilean, one must remember that "true" and "false" are *certain* states, entirely distinct from and unequivalent to "maybe". All of trilean's behavior is based around this rule.

Why Use Trilean?

The logic of trilean has historically been achieved in C++ using either an enumeration or a pair of booleans. However, trilean offers a couple of distinct advantages:

- A trilean variable is exactly one byte in size.
- All three states can be represented in a single variable.
- Trilean is fully compatible with boolean and its constants.
- Conditional logic with trilean is simpler and cleaner.

1.5.2 Using Trilean

Including Trilean

To include Trilean, use the following:

```
#include "pawlib/core_types.hpp"
```

"Maybe" Contant

Along with the normal `true` and `false` constants provided by C++, trilean offers a `maybe` constant. This new constant is designed to be distinct from `true` and `false`, yet usable in the same manner for trileans.

Defining and Assigning

Defining a new trilean variable is simple.

```
tril foo = true;
tril bar = maybe
tril baz = false;
```

In addition to the three constants, you can assign other booleans and trileans.

```
bool foo = true;
tril bar = foo;
tril baz = bar;

// foo, bar, and baz are all 'true'
```

Conditionals

Obviously, one can compare a trilean against of its state constants directly.

```
tril foo = maybe;

if(foo == true)
    // code if TRUE...
else if(foo == maybe)
    // code if MAYBE...
else if(foo == false)
    // code if FALSE...
```

However, one can also test a trilean in the same manner as a boolean.

```
tril foo = maybe;

if(foo)
    // code if TRUE...
else if(~foo)
    // code if MAYBE...
else if(!foo)
    // code if FALSE...
```

You will notice that, in addition to the familiar tests for "true" (`if(foo)`) and "false" (`if(!foo)`), trilean has a third unary operator, `~`, for "maybe" (`if(~foo)`).

Important: Remember, neither the "true" or "false" conditions will ever match "maybe".

This basic behavior is what makes trilean so useful. For example, you may want to repeat a block of code until you encounter specific scenarios to cause it to either pass or fail, using something like `while(~foo)`.

Comparisons

Trilean can be compared to booleans and other trileans using the == and != operators.

```
tril foo = true;
tril bar = maybe;
bool baz = true;

if(foo == bar)
    // This fails.

if(foo != bar)
    // This passes.

if(foo == baz)
    // This passes.

if(baz == foo)
    // This passes.

if(baz == bar)
    // This fails.
```

What About Switch?

The idea of allowing a trilean to cast to an integer was discussed and debated in great deal. Finally, the decision was made to prevent casting a trilean to anything but a boolean (discussed later).

This means that trileans **are not compatible with switch statements**. While this may be initially disappointing to anyone used to using an enumeration for three-state logic, one will notice that an if-statement covering all three states of a trilean has at least 4 less lines of boilerplate.

```
tril foo;

/* This code demonstrates an if statement covering all three states
 * of a trilean. */

if(foo)
{
    // Some code.
}
else if(~foo)
{
    // Some code.
}
else if(!foo)
{
    // Some code.
}
```

1.5.3 Certainty

Because trilean stores its data in two bits, it is possible for a variable to track its last certain state. In other words, if a trilean is “true” or “false,” and then is set to “maybe”, that true/false value is still being stored behind the scenes.

To make this useful, trilean offers a `certain()` function, which returns the last certain state of the variable without actually modifying itself.

```
tril foo = true;
foo = maybe;

bool bar = foo.certain();

// bar is now 'true', while foo is still 'maybe'
```

This behavior can also be used to revert a trilean to its last certain state.

```
tril foo = true;
foo = maybe;
foo = foo.certain();

// foo is now 'true'
```

Implications

The concept of “certainty” technically allows one to recognize and use four trilean states:

- Certain true (`if( foo )`)
- Uncertain true (`if( ~foo && foo.certain() )`)
- Uncertain false (`if (~foo && !foo.certain() )`)
- Certain false (`if (!foo)`)

Uncertainty Variables

The “magical” behavior of assigning the constant “maybe” not affecting the previous certain state is achieved through “uncertainty” variables. Any time an uncertainty is assigned to a trilean, only the uncertainty of the trilean is affected.

The constant “maybe” is usually the only uncertainty object you will interact with. However, it is possible to create your own `certainty`. Be aware that this data type does not provide any mechanism for modifying it after creation.

```
uncertainty my_maybe(true);
uncertainty my_certain(false);
```

As is expected, an uncertainty can never match “true” or “false”, or be directly cast to a boolean. However, the ~ operator works as with trileans.

```
if(~my_maybe)
    // This passes.

if(~my_certain)
    // This fails.
```

(continues on next page)

(continued from previous page)

```
if(~my_certain == false)
    // This passes.
```

The usefulness of an uncertainty variable is, quite probably, limited to allowing manipulation of a trilean's certainty.

1.5.4 Gotchas

Casting to Bool

In order to preserve the core logic that "maybe != true" in statements like `if(foo)`, casting a trilean to a boolean causes "maybe" to be converted to "false".

```
tril foo = maybe;
bool bar = foo;

// bar is now 'false'
```

In most cases, it is recommended to use the `certain()` function.

Note: In case you were wondering, we ensured that "maybe != false" in comparisons and conditionals by separately overloading the `!`, `!=`, and `==` operators.

1.6 Goldilocks

1.6.1 What is Goldilocks?

Goldilocks is a complete testing and runtime-benchmark framework, based on MousePaw Media' LIT (Live-In Testing) Standard. Although LIT is inherently different from "unit testing" and TDD (test-driven development), Goldilocks may be used for either approach. It may also be used in conjunction with other testing systems.

The core idea of Goldilocks is that tests ship in the final code, and can be loaded and executed within normal program execution via a custom interface. A major advantage of this system is that benchmarks may be performed on many systems without the need for additional software.

The fatest way to run tests in Goldilocks is with the *GoldilocksShell*.

Including Goldilocks

To include Goldilocks, use the following:

```
#include "pawlib/goldilocks.hpp"
```

1.6.2 Setting Up Tests

Structure

Every Goldilocks test is derived from the Test abstract class, which has six functions that may be overloaded.

`get_title()`

Returns a string (of type `testdoc_t`) with the title of the test. This is a required function for any test.

Note: The title is separate from the ID (name) of the test used to register the test with the TestManager. You use the ID (name) to refer to the test; the title is displayed on the screen before running the test.

`get_docs()`

Returns a string (of type `testdoc_t`) with the documentation string for the test. This should describe what the test does. This is a required function for any test.

`pre()`

This is an optional function that sets up the test to be run. In cases where a test is run multiple consecutive times, it is only called once. Thus, it must be possible to call `pre()` once, and then successfully call `run()` any number of times.

The function must return true if setup was successful, and false otherwise, to make sure the appropriate actions are taken.

`prefail()`

This is an optional function that tears down the test after a failed call to `pre()`. It is the only function to be called in that situation, and it will not be called under any other circumstances. It has no fail handler itself, so `prefail()` must succeed in any reasonable circumstance.

The function should return a boolean indicating whether the tear-down was successful or not.

Note: Goldilocks currently ignores `prefail()`'s return.

`run()`

This is a required function for any test. It contains all the code for the test run itself. After `pre()` is called once (optionally), `run()` must be able to handle any number of consecutive calls to itself.

There must always be a version of `run()` that accepts no arguments. However, it is not uncommon to overload `run()` to accept a scenario string (part of the LIT Standard) for generating a particular scenario, or prompting a random one to be generated.

The function should return true if the test succeeded, and false if it failed.

Important: `run()` (with no arguments) should be consistent in its success. Assuming pre() was successful, if the first consecutive call to `run()` is successful, all subsequent calls to run() must also be successful. This is vital to the benchmarker functions, as they can call a single test up to 10,000 times. One consideration, then, is that run() should only use one scenario in a single lifetime, unless explicitly instructed by its function arguments to do otherwise.

`janitor()`

This is called after each repeat of `run()` during benchmarking and comparative benchmarking. It is designed to perform cleanup in between `run()` calls, but not to perform first time setup (`pre()`) or end of testing (`post()`) cleanup. It returns a boolean indicating success.

`post()`

This is an optional function which is called at the end of a test's normal lifetime. It is the primary teardown function, generally responsible for cleaning up whatever was created in `pre()` and `run()`. It is normally only if `run()` returns true, although it will be called at the end of benchmarking regardless of `run()`'s success.

This function should return a boolean indicating success. It has no fail handler itself, so `post()` should succeed in all reasonable circumstances.

Note: Goldilocks currently ignores `post()`'s return.

`postmortem()`

This is an optional teardown function which is usually called if a test fails (`run()` returns false). It is responsible for cleaning up whatever was created in `pre()` and `run()`, much like `post()` is, but again only for those scenarios where `run()` fails.

This function should return a boolean indicating success. It has no fail handler itself, so `postmortem()` should succeed in all reasonable circumstances.

Creating a Test

Creating a test is as simple as creating a class that inherits from `Test (from goldilocks.hpp)`, which is a pure virtual base class.

Important: The constructor and destructor must obviously be defined, however, it is not recommended that they actually do anything - all setup and teardown tasks must be handled by the other functions in order to ensure proper functionality - a test instance is defined once when Goldilocks is set up, but it is highly likely to have multiple lifetimes.

Only bool `run()` must be defined in a test class. The rest of the functions are already defined (they do nothing other than return true), so you only need to define them if you require them to do something.

The following example exhibits a properly-defined, though overly simplistic, test. In reality, we could have skipped `pre()`, `prefail()`, `janitor()`, `postmortem()`, and `post()`, but they are defined to demonstrate their behavior.

```
#include <iochannel.hpp>
#include <goldilocks.hpp>

class TestFoo : public Test
{
public:
    TestFoo(){}

    testdoc_t get_title()
    {
        return "Example Test";
    }

    testdoc_t get_docs()
    {
        return "This is the docstring for our example test."
    }

    bool pre()
    {
        ioc << cat_testing << "Do Pre Stuff" << IOCtrl::endl;
        return true;
    }
    bool prefail()
    {
        ioc << cat_testing << "Do Prefail Stuff" << IOCtrl::endl;
        return true;
    }
    bool run()
    {
        ioc << cat_testing << "Do Test Stuff" << IOCtrl::endl;
        char str[5000] = {'\0'};
        for(int a=0;a<5000;a++)
        {
            str[a] = 'A';
        }
        return true;
    }
    bool janitor()
    {
        ioc << cat_testing << "Do Janitorial Stuff" << IOCtrl::endl;
        return true;
    }
    bool postmortem()
    {
        ioc << cat_testing << "Do Postmortem Stuff" << IOCtrl::endl;
        return true;
    }
    bool post()
    {
        ioc << cat_testing << "Do Post Stuff" << IOCtrl::endl;
        return true;
    }
```

(continues on next page)

(continued from previous page)

```
    ~TestFoo(){}
};
```

Registering a Test

Registering a test with Goldilocks is a trivial task, thanks to its `register_test()` function. Once a test class has been defined, as above, simply register it via…

```
//Assuming testmanager is our instance of the Goldilocks test manager.
testmanager.register_test("TestFoo", new TestFoo);
```

Goldilocks will now actually own the instance of `TestFoo`, and automatically handle its deletion at the proper time.

Warning: Goldilocks actually requires exclusive ownership of each test object registered to it - thus, you should always pass the new declaration as the second argument. If you create the object first, and then pass the pointer, you run a high risk of a segfault or other undefined behavior.

The test can now be called by name using Goldilocks' various functions. (See below.)

You can also optionally register a comparative test for benchmarking, which will be run against the main test in the benchmarker.

```
//Assuming testmanager is our instance of the Goldilocks test manager.
testmanager.register_test("TestFoo", new TestFoo, new TestBar);
```

Running a Test

Once a test is registered with Goldilocks, running it is quite easy.

```
//Run the test once.
testmanager.run_test("TestFoo");

//Benchmark TestFoo on 100 repetitions.
testmanager.run_benchmark("TestFoo", 100);

//Compare TestFoo and TestBar on 100 repetitions.
testmanager.run_compare("TestFoo", "TestBar", 100);
```

1.6.3 Setting Up Suites

A Suite is a collection of tests. In a typical use of Goldilocks, all tests are organized into Suites.

In addition to allowing on-demand loading groups of tests, a Suite can be "batch run", where all of its tests are run in succession. When one test fails, the batch run halts and returns false.

Structure

Every Goldilocks suite is derived from the `TestSuite` abstract class. This only has two functions to overload, but both are required.

`get_title()`

Returns a string (of type `testsuitedoc_t`) with the title of the suite. This is the a required function for any test.

Note: The title is separate from the ID (name) of the test used to register the test with the TestManager. You use the ID (name) to refer to the test; the title is displayed on the screen before running the test.

`load_tests()`

This function specifies which tests belong to the suite.

`TestSuite` provides a function `register_test()` which properly registers each test with both the suite and the TestManager itself. For convenience, it follows the same format as `TestManager::register_test()`, with the exception of an optional boolean argument for specifying a test which belongs to the suite, but should not be part of the Suite's batch run.

One reason to exclude a test from the batch run for the Suite is if the test is a stress test that takes a long time to run.

We can also register the comparative tests as an optional fourth argument.

Below is an example of a Suite's `load_tests`.

```
void TestSuite_MagicThing::load_tests()
{
    /* Register this test with both the suite and the test manager.
     * Also register the comparative form. */
    register_test("t101", new MagicThing_Poof(), true, new OtherThing_Poof());

    register_test("t102", new MagicThing_Vanish());

    register_test("t103", new MagicThing_Levitate());

    register_test("t104", new MagicThing_Telepathy());

    /* This test will be loaded by the suite, but will be excluded
     * from the batch run. */
    register_test("t105", new MagicThing_SawInHalf(), true);
}
```

We have registered five tests with this suite, not counting the comparative form of the one. Upon loading the suite, all five tests will be loaded into the test manager. However, if we were to batch run this suite, only four of those tests (t101, t102, t103, and t104) would be run.

Registering a Suite

Registering a suite with Goldilocks is as easy as registering a test. Simply use its `register_suite()` function. Once a suite class has been defined, as above, it is registered with…

```
//Assuming testmanager is our instance of the Goldilocks test manager.
testmanager.register_suite("TestSuiteFoo", new TestSuiteFoo());
```

As with tests, Goldilocks owns the instance of `TestSuiteFoo`, and automatically handles its deletion at the proper time.

Warning: Goldilocks requires exclusive ownership of each suite object registered to it, the same as it does tests.

Loading a Suite

One of the major advantages of using a suite is that you can load its tests on demand. This is especially useful if you have hundreds or thousands of tests.

```
//Load a particular suite.
testmanager.load_suite("TestSuiteFoo");
```

Of course, sometimes you don't want to have to load each suite manually. As a shortcut, you can just load all suites currently registered with the test manager by calling…

```
//Load a particular suite.
testmanager.load_suite();
```

Running a Suite

You can start a batch run of all the suite's tests using…

```
//Batch run all tests in a suite.
testmanager.run_suite("TestSuiteFoo");
```

1.6.4 Interfacing Functions

Goldilocks provides a number of convenience functions to aid in creating an interactive command-line interface for the system.

In most cases, you can probably just use the GoldilocksShell (see *goldilocksshell*).

Functions

list_suites()

You can display the names and titles of all the tests currently registered in the test manager using…

```
// List all registered suites with their names and titles.
testmanager.list_suites();

// List all registered suites with their name only (no title).
testmanager.list_suites(false);
```

list_tests()

You can display the names and titles of all the tests currently registered (loaded) in the test manager using…

```
// List all registered tests with their names and titles.
testmanager.list_tests();

// List all registered tests with their name only (no title).
testmanager.list_tests(false);
```

If a test is loaded via a suite, it will not appear in this list until its suite has actually been loaded during that session.

i_load_suite()

Identical usage to `load_suite()`, except it prompts the user for confirmation before loading a suite.

i_run_benchmark()

Identical usage to `run_benchmark()`, except it prompts the user for confirmation before running the benchmark.

i_run_compare()

Identical usage to `run_compare()`, except it prompts the user for confirmation before running the compare.

i_run_suite()

Identical usage to `run_suite()`, except it prompts the user for confirmation before running the suite.

`i_run_test()`

Identical usage to `run_test()`, except it prompts the user for confirmation before running the test.

1.6.5 Benchmarker Output

The Goldilocks benchmarker outputs a *lot* of information. This section describes how to read it.

Pass Types

To account for the effects of cache warming, Goldilocks makes three passes, each with a specific behavior:

- **Mama Bear** attempts to simulate a "cold cache," increasing the likelihood of cache misses. This is done by running tests A and B alternatingly.
- **Papa Bear** attempts to simulate a "hot cache," decreasing the likelihood of cache misses. This is done by running all repetitions of test A before running all repetitions of test B.
- **Baby Bear** attempts to simulate average (or "just right") cache warming effects, such as what might be seen in typical program executions. This is done by running eight repetitions of each test alternatingly - 8 As, 8 Bs, 8 As, etc.

After running all three passes, the benchmarker results are displayed.

Result Groups

At the top of the results, we see the `BASELINE MEASUREMENTS`. These are based on measuring the actual measurement function of our benchmarker.

These results are important, as this is an indicator of fluctuations in results from external factors. If either of the RSD (Relative Standard Deviation) numbers are high (>10%), the results of the benchmarker may be thrown off.

Next, we see the individual results for each test beneath each pass type. The verdict is displayed below both sets of results, indicating which test was faster, and by how much. The verdict is ultimately the difference between means, but if that difference is less than the standard deviation, it will indicate that the tests are "roughly equal."

Statistical Data

Let's break down the statistical data in our results.

Most lines show two sets of values, separated with a `/` character. The *left* side is the **RAW** value, accounting for each measurement taken. The *right* side is the **ADJUSTED** value, which is the value after outlier measurements have been removed from the data.

The **MEAN ()** is the average number of CPU cycles for a single run of the test.

The **MIN-MAX(RANGE)** shows the lowest and highest measurement in the set, as well as the difference between the two (the range).

OUTLIERS shows how many values have been removed from the ADJUSTED set. Outliers are determined mathematically, and removing them allows us to account for external factors, such as other processes using the CPU during the benchmark.

SD () shows our standard deviation, which indicates how much fluctuation occurs between results. By itself, the standard deviation is not usually meaningful.

The **RSD**, or Relative Standard Deviation, is the percentage form of the standard deviation. This is perhaps the most important statistic! The lower the RSD, the more precise the benchmark results are. If the RSD is too high, it will actually be flagged as red.

The statistical data above can provide a useful indicator of the reliability of the benchmark results.

A high RSD may indicate that the results are "contaminated" by external factors. It is often helpful to run the comparative benchmark multiple times, and taking the pass with the lowest RSD.

However, higher RSDs may be the result of the tests themselves, as we'll see in the following example.

Other warning signs that the results may be contaminated or inaccurate include:

- The presence of outliers in BASELINE.
- RSDs > 10% in BASELINE.
- Red-flagged RSDs (> 25%) (unless the test has a technical reason to fluctuate in CPU cycle measurements between tests).
- Significantly different verdicts between passes.

The precision and accuracy of the results may be further validated by running the comparative benchmark multiple times, especially across computers, and directly comparing the RSDs and verdict outcomes. While actual CPU cycle measurements may vary greatly between similar systems, the relative outcomes should remain fairly consistent on most systems with the same processor architecture.

Statistical Data Example

Let's look at the comparison between the "shift" (insert at beginning) functionality of FlexArray and `std::vector`. You can run this yourself using the PawLIB tester, with the command `benchmark P-tB1002`.

We always start by screening the baseline:

```
BASELINE MEASUREMENTS
    MEAN (): 64 / 65
    MIN-MAX(RANGE): 58-75(17) / 58-75(17)
    OUTLIERS: 0 LOW, 0 HIGH
    SD (): 5.47 / 5.38
    RSD: 8% / 8%
```

We have no outliers and very low RSDs, so our results probably aren't contaminated. Of course, benchmarking is unpredictable, and external factors may change during the benchmarking itself. However, we have no reason here to throw out the results.

Had we seen an RSD greater than 10% for either result, it would have been wise to discard these results and rerun the benchmark altogether.

Now let's look at the first pass, MAMA BEAR, which is designed to demonstrate the effects of cache misses:

```
MAMA BEAR: [FlexArray: Shift 1000 Integers to Front (FlexArray)]
    MEAN (): 414650 / 401451
    MIN-MAX(RANGE): 262280-739036(476756) / 262280-323876(61596)
    OUTLIERS: 0 LOW, 5 HIGH
    SD (): 106700.22 / 76270.09
    RSD: 25% / 18%

MAMA BEAR: [FlexArray: Shift 1000 Integers to Front (std::vector)]
    MEAN (): 904723 / 876586
```

(continues on next page)

(continued from previous page)

```
    MIN-MAX(RANGE): 664354-1537966(873612) / 664354-714892(50538)
    OUTLIERS: 0 LOW, 5 HIGH
    SD (): 232960.59 / 169329.87
    RSD: 25% / 19%
```

Unsurprisingly, both results show some high outliers. The RSDs are roughly equal, however, so this is probably the result of those cache misses or other related factors.

Warning: How the two tests are structured matters! We are very careful to ensure both tests have the same structure and implementation, so the only difference between the two is the functions or algorithms we are directly comparing.

Looking at the result:

```
MAMA BEAR: VERDICT
         RAW: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪490073 cycles.
    ADJUSTED: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪398864.90807662549195 cycles.
```

FlexArray wins that round.

Now let's look at PAPA BEAR, which attempts to demonstrate cache warming:

```
PAPA BEAR: TEST [FlexArray: Shift 1000 Integers to Front (FlexArray)]
    MEAN (): 321917 / 325168
    MIN-MAX(RANGE): 305608-310824(5216) / 305608-310824(5216)
    OUTLIERS: 0 LOW, 0 HIGH
    SD (): 28252.27 / 28548.56
    RSD: 8% / 8%

PAPA BEAR: TEST [FlexArray: Shift 1000 Integers to Front (std::vector)]
    MEAN (): 654278 / 659817
    MIN-MAX(RANGE): 608020-765749(157729) / 608020-685548(77528)
    OUTLIERS: 0 LOW, 1 HIGH
    SD (): 53785.7 / 53494.46
    RSD: 8% / 8%
```

Unlike MAMA BEAR, these results have much lower RSDs - in fact, they are equal to the BENCHMARK RSDs (the ideal scenario) - and only one outlier between the two. This further lends itself to our theory that the higher RSDs in MAMA BEAR are the result of cache misses.

FlexArray wins this as well, albeit by a somewhat narrower margin:

```
PAPA BEAR: VERDICT
         RAW: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪332361 cycles.
    ADJUSTED: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪306100.43052620673552 cycles.
```

Finally, we look at BABY BEAR, which is intended to be the most similar to typical use scenarios:

```
BABY BEAR: TEST [FlexArray: Shift 1000 Integers to Front (FlexArray)]
    MEAN (): 317852 / 321814
    MIN-MAX(RANGE): 247433-323226(75793) / 306612-323226(16614)
    OUTLIERS: 1 LOW, 0 HIGH
    SD (): 33872.37 / 33610.86
    RSD: 10% / 10%

BABY BEAR: TEST [FlexArray: Shift 1000 Integers to Front (std::vector)]
    MEAN (): 648568 / 652663
    MIN-MAX(RANGE): 537774-780641(242867) / 537774-755231(217457)
    OUTLIERS: 0 LOW, 2 HIGH
    SD (): 60925.17 / 58541.29
    RSD: 9% / 8%
```

Our RSDs are slightly higher than with PAPA BEAR, but we still see relatively few outliers (a total of 3).

The BABY BEAR verdict indicates that FlexArray is the fastest, even in this scenario:

```
BABY BEAR: VERDICT
         RAW: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪330716 cycles.
    ADJUSTED: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪297238.13385525450576 cycles.
```

1.7 GoldilocksShell

1.7.1 What Is GoldilocksShell?

What good are tests without a way to run them? While you can certainly write your own class for interacting with Golidlocks, we wanted to provide a quick and easy way to run your tests and suites interactively.

Assuming you have one or more TestSuites configured (see *Setting Up Suites*), you can set up a GoldilocksShell relatively quickly!

1.7.2 Setting Up

We import GoldilocksShell with…

```
#include "pawlib/goldilocks_shell.hpp"
```

We start by defining a new `GoldilocksShell` object instaces somewhere in our code, preferably in our `main()` function.

```
GoldilocksShell* shell = new GoldilocksShell(">> ");
```

The part in quotes is the *prompt string*, which will appear at the start of every line where the user can type in the interactive terminal.

Now we need to register all of our Goldilocks `TestSuite` classes with the shell. Note that we don't need to create instances of these ourselves, but merely pass the class as a type. We also need to specify the name of the suite in quotes: this name will be what is used to idenfity it in the shell.

```
shell->register_suite<TestSuite_Brakes>("s-brakes");
shell->register_suite<TestSuite_Hologram>("s-hologram");
shell->register_suite<TestSuite_TimeRotor>("s-timerotor");
shell->register_suite<TestSuite_CloisterBell>("s-cloisterbell");
```

That is it! We are now ready to use GolidlocksShell.

1.7.3 Interactive Mode

As long as we're running in the command line, we can hand over control to the GoldilocksShell via a single line of code…

```
shell->interactive();
```

The GoldilocksShell will immediately launch and take over the terminal, using IOChannel.

Commands

In Interactive Mode, you are given a complete shell for executing Golilocks tests and suites.

Note: This initial version of GoldilocksTester does not offer support for the conventional shell commands, including history or arrow-key navigation. We'll be adding these in a later version.

To get help at any point, run `help`. To quit, type `exit`.

In this guide, we'll be using the default GoldilocksShell prompt symbol, `:`. This may be different for your project, depending on how you configured GoldilocksShell.

Listing and Loading Suites and Tests

When you first start GoldilocksShell, no tests have been loaded into memory. However, all the suites you specified are *ready* to be loaded.

To see all the available suites, run `listsuites`:

```
: listsuites
s-brakes: TARDIS Brakes Tests
s-hologram: TARDIS Holographic Interface Tests
s-timerotor: TARDIS Time Rotor Tests
s-cloisterbell: TARDIS Cloister Bell Tests
```

Thus, before you can run a test or suite, you must first **load** the suite containing it:

```
: load s-brakes
TARDIS Brakes Tests
Suite loaded.
```

Now you can run the `list` command to see all the loaded tests:

```
: list
t-brakes-engage: TARDIS Brakes: Engage Brakes
t-brakes-warn: TARDIS Brakes: No Brakes Warning
t-brakes-disengage: TARDIS Brakes: Disengage Brakes
t-brakes-fail: TARDIS Brakes: Brake Failure Protocol
t-brakes-pressure: TARDIS Brakes: Brake Pressure Test
```

Note: If `list` does not show any tests, be sure you've loaded at least one suite first.

If you just want to load *all* suites, simply run the `load` command without any arguments. It will ask you to confirm your choice:

```
: load
Load ALL test suites? (y/N) y
TARDIS Brakes Tests loaded.
TARDIS Holographic Interface Tests loaded.
TARDIS Time Rotor Tests loaded.
TARDIS Cloister Bell Tests loaded.
```

You can find out more information about any test using the `about` command:

```
: about t-brakes-engage
TARDIS Brakes: Engage Brakes
Ensures the controls are capable of engaging the brakes.
```

Running Tests and Suites

It is possible to run any test using the `run` command. This command always asks you to confirm before continuing:

```
: run t-brakes-engage
Run test TARDIS Brakes: Engage Brakes [t-brakes-engage]? (y/N) y
===== [TARDIS Brakes: Engage Brakes] =====
Pass 1 of 1
TEST COMPLETE
```

Optionally, you can repeat a test multiple times by specifying the number of times to repeat it.

> : run t-brakes-engage 5 Run test TARDIS Brakes: Engage Brakes [t-brakes-engage]? (y/N) y ===== [TARDIS Brakes: Engage Brakes] ===== Pass 1 of 5 Pass 2 of 5 Pass 3 of 5 Pass 4 of 5 Pass 5 of 5 TEST COMPLETE

You can also run an entire suite in one step:

```
: run s-brakes
Run test suite TARDIS Brakes Tests [s-brakes]? (y/N) y
===== [TARDIS Brakes Tests] =====
===== [TARDIS Brakes: Engage Brakes] =====
Pass 1 of 1
TEST COMPLETE
===== [TARDIS Brakes: No Brakes Warning] =====
Pass 1 of 1
```

(continues on next page)

(continued from previous page)

```
TEST COMPLETE
===== [TARDIS Brakes: Disengage Brakes] =====
Pass 1 of 1
TEST COMPLETE
===== [TARDIS Brakes: Brake Failure Protocol] =====
Pass 1 of 1
TEST COMPLETE
===== [TARDIS Brakes: Brake Pressure Test] =====
Pass 1 of 1
TEST COMPLETE

SUITE COMPLETE
```

Note: If you specify a repeat number for running a suite, it will be ignored.

Benchmarking

Golidlocks supports *comparative benchmarking*. There are two ways to run such a benchmark.

The first method requires a comparative test to be specified within a suite (see *load_tests()*). If you've done this, you can benchmark the test and its comparative, and output the complete benchmark stats:

```
: benchmark t-brakes-engage
Run comparative benchmark between TARDIS Brakes: Engage Brakes [t-brakes-engage] and␣
↪TARDIS Brakes: Handbrake? (y/N) at 100 repetitions? (y/N) y
======================
|     BENCHMARKER    |
======================
```

Upon completion it will display the complete benchmarker stats (see *Benchmarker Output*).

You can also specify the number of times to run the benchmarker (the default is 100):

```
: benchmark t-brakes-engage 1000
Run comparative benchmark between TARDIS Brakes: Engage Brakes [t-brakes-engage] and␣
↪TARDIS Brakes: Handbrake? (y/N) at 1000 repetitions? (y/N) y
======================
|     BENCHMARKER    |
======================
```

You can also run a comparative benchmark on any two tests using the `compare` function. It functions in much the same way, except that you specify *two* tests instead of one, and then the optional repetition count:

```
: compare t-brakes-engage t-brakes disengage 500
Run comparative benchmark between TARDIS Brakes: Engage Brakes [t-brakes-engage] and␣
↪TARDIS Brakes: Disengage [t-brakes-disengage]? (y/N) at 1000 repetitions? (y/N) y
======================
|     BENCHMARKER    |
======================
```

1.7.4 Command Line Argument Mode

Invocation

GoldilocksShell is also designed to handle the same input arguments as your typical `int main()`, which allows you to invoke the shell using command-line arguments.

This is especially useful for integrating Goldilocks into a Continuous Integration [CI] system, such as Jenkins. If the specified tests and suites are successful, the program will exit with code `0`; failures will cause the program to exit with code 1.

To use this feature, you must simply pass the argument count and argument array to the GoldilocksShell's `command()` function. It handles its own argument parsing.

```
int main(int argc, char* argv[])
{
    // ...setup code here...

    // If we got command-line arguments...
    if(argc > 1)
    {
        return shell->command(argc, argv);
    }

    return 0;
}
```

Skipping Arguments

If you accept other arguments via command-line, you may ask GoldilocksShell to skip those. Just specify the number of arguments to skip in the third argument.

Important: GoldilocksShell already knows to skip the first argument, which is the program invocation. You only need to tell it how many *extra* arguments to skip.

For example…

```
// myprogram --goldilocks --run sometest
int main(int argc, char* argv[])
{
    // ...setup code here...

    // If we're supposed to invoke Goldilocks.
    if(argc > 1 && strcmp(argv[1], "--goldilocks") == 0)
    {
        // Asking GoldilocksShell to skip one argument...
        return shell->command(argc, argv, 1);
        // Now it will only process arguments starting from ``--run``...
    }

    return 0;
}
```

Usage

GoldilocksShell's command line interface accepts multiple arguments, which are used to load and run tests, suites, and benchmarks. Commands are always run from left to right, in order.

The basic commands are as follows:

- `--help` displays help.
- `--listsuites` lists all available suites.
- `--load suite` loads the suite `suite`.
- `--list` lists all loaded tests.
- `--run item` runs the test or suite `item`.
- `--benchmark item` benchmarks the test `item`.

Important: The command line does not include the `compare` function, nor the ability to specify the number of test repetitions.

Ordinarily, to run a test, you must first load the suite containing it. However, for the sake of convenience, if you don't explicitly load any tests in the command, it will just load all suites. Thus…

```
$ tester --run t-brakes-engage
```

…will just load all the suites before attempting to run the test `t-brakes-engage`.

If you want to only load a single suite, perhaps to see what tests it contains, just include the `--load` argument. (Remember, if you don't explicitly load any suites, all the suites will be loaded.)

```
$ tester --load s-brakes --list
```

Warning: Each command only accepts one argument! If you want to load multiple suites, you must precede each suite ID with the `--load` argument.

Arguments are run in order, from left to right, and the program doesn't exit until all of them are finished. This means you can run multiple tests in one command; success will only be reported (exit code `0`) if all the tests pass.

```
$ tester --load s-brakes --run t-brakes-engage --run t-brakes-disengage
```

The above command, after loading only the specified suite, will run the requested tests. If they *both* succeed, the program will exit reporting success (exit code `0`).

Warning: Each command only accepts one argument! If you want to load multiple suites, you must precede each suite ID with the `--load` argument.

We can also run benchmarks from the command line. `--benchmark` bases its success/fail condition on the Baby Bear comparison; success means either (a) the main test is faster than its comparative, or (b) the two tests are roughly identical in performance ("dead heat").

```
$ tester --load s-brakes --benchmark t-brakes-engage
```

1.7.5 A Complete Example

Let's tie all this together. Here's an example of a complete `int main()` function set up to use GoldilocksShell, as outlined in the previous sections.

```
int main(int argc, char* argv[])
{
    GoldilocksShell* shell = new GoldilocksShell(">> ");

    shell->register_suite<TestSuite_Brakes>("s-brakes");
    shell->register_suite<TestSuite_Hologram>("s-hologram");
    shell->register_suite<TestSuite_TimeRotor>("s-timerotor");
    shell->register_suite<TestSuite_CloisterBell>("s-cloisterbell");

    // If we got command-line arguments...
    if(argc > 1)
    {
        return shell->command(argc, argv);
    }
    else
    {
        // Shift control to the interactive console.
        shell->interactive();
    }

    // Delete our GoldilocksShell.
    delete shell;
    shell = 0;

    return 0;
}
```

1.8 IOChannel

1.8.1 What is IOChannel?

IOChannel is designed as a wrapper and, depending on usage, a replacement for `std::cout` and `printf()`. Its sports a number of unique and useful features.

- Multiple asynchronous outputs.
- Message priorities (verbosity).
- Message categories.
- Built-in output formatting.
- Advanced memory tools.

1.8.2 Setting up IOChannel

Including IOChannel

To include IOChannel, use the following:

```
#include "pawlib/iochannel.hpp"
```

IOChannel Instance

For convenience, a single static global instance of IOChannel, `ioc`, exists in that header. It is suitable for most purposes, though a custom iochannel instance may be declared. All inputs and outputs that the developer wishes to interface with one another via this class must share the same instance.

Documentation Assumptions

For purposes of expediency, the default global static instance `ioc` will be used in this documentation. All namespaces outside of the scope of PawLIB will be stated explicitly.

1.8.3 Concepts

IOChannel uses two unique concepts, **Verbosity** and **Category**, to determine where and how a message is routed.

Category

The benefit to having categories on messages is that you can route different kinds of messages to different outputs. For example, you might send all errors and warnings to a debug terminal, and reserve "normal" messages for game notifications.

Category	Enum	Use
Normal	`IOCat::normal`	Regular use messages, especially those you want the user to see.
Warning	`IOCat::warning`	Warnings about potential problems.
Error	`IOCat::error`	Error messages.
Debug	`IOCat::debug`	Messages that might help you track down problems.
Testing	`IOCat::testing`	Messages related solely to testing.
All	`IOCat::all`	All of the above.

One of the advantages of this system is that you can actually leave messages in the code, and just control when and how they are processed and broadcast. This means you can actually ship with debugging statements still alive in the code, allowing you to diagnose problems on any machine.

You can control which of these categories messages are broadcast from using the echo settings (*Internal Broadcast Settings (Echo)*) and signals (*Category Signals (signal_c_...)*).

Verbosity

Some messages we need to see every time, and others only in special circumstances. This is what verbosity is for.

Verbosity	Enum	Use
Quiet	`IOVrb::quiet`	Only essential messages and errors. For normal end-use. Shipping default.
Normal	`IOVrb::normal`	Common messages and errors. For common and normal end-user testing.
Chatty	`IOVrb::chatty`	Most messages and errors. For detailed testing and debugging.
TMI	`IOVrb::tmi`	Absolutely everything. For intense testing, detailed debugging, and driving the developers crazy.

One example of verbosity in action would be in debugging messages. A notification about a rare and potentially problematic function being called might be `IOVrb::normal`, while the output of a loop iterator would probably be `IOVrb::tmi`.

You can control which of these categories messages are broadcast from using the echo settings (*Internal Broadcast Settings (Echo)*) and signals (*Verbosity Signals (signal_v_...)*).

1.8.4 Output

General

All output is done using the stream insertion (<<) operator, in the same manner as with `std::cout`. Before a message is broadcast, a stream control flags such as `IOCtrl::endl` must be passed.

`IOCtrl::endl` serves as an "end of transmission" [EoT] flag, clears any formatting set during the stream, and inserts a final newline character before flushing the stream. Thus, `\n` is not needed if the output should be displayed on a single line. This functionality also allows a single transmission to be split up over multiple lines, if necessary. Other stream control enumerations have different behaviors. (See *Stream Control*)

```
ioc << "This is the first part. ";
//Some more code here.
ioc << "This is the second part." << IOCtrl::endl;
```

Strings

IOChannel natively supports string literals, cstring (char arrays), `std::string`, and `onestring`.

These are passed in using the << operator, as with anything being output via IOChannel. The message will not be broadcast until an EoT (end-of-transmission) flag is passed.

```
ioc << "Hello, world!" << IOCtrl::endl;
//OUTPUT: "Hello, world!"

char* cstr = "I am a Cstring.\0";
ioc << cstr << IOCtrl::endl;
//OUTPUT: "I am a Cstring."

std::string stdstr = "I am a standard string.";
ioc << stdstr << IOCtrl::endl;
//OUTPUT: "I am a standard string."
```

Formatting

Cross-platform output formatting is built in to IOChannel. This means that formatting can be set using the IOFormat flags, and it will display correctly on each output and environment.

```
ioc << IOFormatTextAttr::bold << IOFormatTextFG::red << "This is bold, red text. "
    << IOFormatTextAttr::underline << IOFormatTextFG::blue << IOFormatTextBG::yellow <<
↪"This is bold, underline, blue text with a yellow background. "
    << IOFormatTextAttr::none << IOFormatTextFG::none << IOFormatTextBG::none << "This␣
↪is normal text."
    << IOCtrl::endl;
//The output is exactly what you'd expect.
```

Important: Currently, only ANSI is used. Formatting-removed and an easy-to-parse formatting flag system for custom outputs will be added soon.

Alternative, you can use the `IOFormat` object to store multiple flags. (See *Formatting Objects*)

Variable Input

IOChannel supports all basic C/C++ data types.

- Boolean (`bool`)
- Char (`char`)
- Integer (`int`) and its various forms.
- Float (`float`)
- Double (`double`)

Boolean

Output for boolean is pretty basic and boring.

```
bool foo = true;
ioc << foo << IOCtrl::endl;
//OUTPUT: "TRUE"
```

The output style can be adjusted, however, using the `IOFormatBool::` flags.

```
bool foo = true;
ioc << IOFormatBool::lower << foo << IOCtrl::endl;
//OUTPUT: "true"
ioc << IOFormatBool::upper << foo << IOCtrl::endl;
//OUTPUT: "True"
ioc << IOFormatBool::caps << foo << IOCtrl::endl;
//OUTPUT: "TRUE"
ioc << IOFormatBool::numeral << foo << IOCtrl::endl;
//OUTPUT: "1"
```

Char

Since char can represent both an integer and a character, IOChannel lets you display it as either. By default, IOChannel displays the char as a literal character. Using the `IOFormatCharValue::as_int` flag forces it to print as an integer.

```
char foo = 'A';
ioc << "Character " << foo << " has ASCII value "
    << IOFormatCharValue::as_int << foo << IOCtrl::endl;
//OUTPUT: Character A has ASCII value 65
```

When output as an integer, char can be used with all of the enumerations for int (see that section).

Integer

An `int` can be represented in any base (radix) from binary (base 2) to base 35 using the `IOFormatBase::` flags.

```
int foo = 12345;
ioc << "Binary: " << IOFormatBase::bin << foo << IOCtrl::endl;
ioc << "Octal: " << IOFormatBase::oct << foo << IOCtrl::endl;
ioc << "Decimal: " << IOFormatBase::dec << foo << IOCtrl::endl;
ioc << "Dozenal: " << IOFormatBase::doz << foo << IOCtrl::endl;
ioc << "Hexadecimal: " << IOFormatBase::hex << foo << IOCtrl::endl;
ioc << "Base 31: " << IOFormatBase::b31 << foo << IOCtrl::endl;

/*OUTPUT:
Binary: 11000000111001
Octal: 30071
Decimal: 12345
Dozenal: 7189
Hexadecimal: 3039
Base 31: cq7
*/
```

In bases larger than decimal (10), the letter numerals can be output as lowercase or uppercase (default) using the `IOFormatNumCase::` flags.

```
int foo = 187254;
ioc << "Hexadecimal Lower: " << IOFormatBase::hex << foo << IOCtrl::endl;
ioc << "Hexadecimal Upper: " << IOFormatNumCase::upper
    << IOFormatBase::hex << foo << IOCtrl::endl;

/*OUTPUT:
Hexadecimal Lower: 2db76
Hexadecimal Upper: 2DB76
*/
```

Float and Double

Float and Double can only be output in base 10 directly. (Hexadecimal output is only possible through a pointer memory dump. See that section.) However, the significands (the number of digits after the decimal point) and use of scientific notation can be modified. By default, significands is 14, and use of scientific notation is automatic for very large and small numbers.

Significands can be modified using the `IOFormatSignificands(#)` flag. Scientific notation can be turned on with `IOFormatSciNotation::on`, and off using `IOFormatSciNotation::none`. It can also be reset to automatic with `IOFormatSciNotation::automatic`.

```
float foo = 12345.12345678912345;
ioc << "Significands 5, no sci: " << IOFormatSignificands(5) << foo << IOCtrl::endl;
ioc << "Significands 10, sci: " << IOFormatSignificands(10)
    << IOFormatSciNotation::on << foo << IOCtrl::endl;

/*OUTPUT:
Significands 5, no sci: 12345.12304
Significands 10, sci: 1.2345123046e+4
*/
```

Both types work the same.

Pointer Output

One of the most powerful features of IOChannel is its handling of pointers. In addition to printing the value at known pointer types, it can print the address or raw memory for ANY pointer, even for custom objects.

Pointer Value

By default, IOChannel will attempt to print the value at the pointers. This can also be forced using `IOFormatPtr::value`.

```
int foo = 12345;
int* fooptr = &foo;
ioc << "Value of foo: " << IOFormatPtr::value << fooptr << IOCtrl::endl;

char* bar = "My name is Bob, and I am a coder.\0";
ioc << "Value of bar: " << bar << IOCtrl::endl;

/*OUTPUT:
Value of foo: 12345
Value of bar: My name is Bob, and I am a coder.
*/
```

Pointer Address

IOChannel can print out the address of the pointer in hexadecimal using `IOFormatPtr::address`. It displays with lowercase letter numerals by default, though these can be displayed in uppercase using `IOFormatNumCase::upper`. It is capable of doing this with any pointer, even for custom objects.

```
int foo = 12345;
int* fooptr = &foo;
ioc << "Address of foo: " << IOFormatPtr::address << fooptr << IOCtrl::endl;

char* bar = "My name is Bob, and I am a coder.\0";
ioc << "Address of bar: " << IOFormatPtr::address << IOFormatNumCase::upper
    << bar << IOCtrl::endl;

/*OUTPUT:
Address of foo: 0x7ffc33518308
Address of bar: 0x405AF0
*/
```

Pointer Memory Dump

IOChannel is capable of dumping the raw memory at any pointer using `IOFormatPtr::memory`. The function is safe for pointers to most objects and atomic types, as the memory dump will automatically determine the size and will never overrun the size of the variable. With char pointers (cstring), the only danger is when the cstring is not null terminated.

Spacing can be added between bytes (`IOFormatMemSep::byte`) and bytewords (`IOFormatMemSep::word`), or both (`IOFormatMemSep::all`). By default, the memory dumps with no spacing (`IOFormatMemSep::none`).

```
int foo = 12345;
int* fooptr = &foo;
ioc << "Memory dump of foo: " << IOFormatPtr::memory << IOFormatMemSep::byte
    << fooptr << IOCtrl::endl;

char* bar = "My name is Bob, and I am a coder.\0";
ioc << "Memory dump of bar: " << IOFormatPtr::memory << IOFormatMemSep::all
    << bar << IOCtrl::endl;

/*OUTPUT:
Memory dump of foo: 39 30 00 00
Memory dump of bar: 4d 79 20 6e 61 6d 65 20 | 69 73 20 42 6f 62 2c 20 | 61 6e 64 20 49␣
↪20 61 6d | 20 61 20 63 6f 64 65 72 | 2e 00
*/
```

The following dumps the raw memory for a custom object.

```
//Let's define a struct as our custom object, and make an instance of it.
struct CustomStruct
{
    int foo = 12345;
    double bar = 123.987654321;
    char faz[15] = "Hello, world!\0";
    void increment(){foo++;bar++;}
```

(continues on next page)

(continued from previous page)

```
};
CustomStruct blah;

ioc << IOFormatPtr::memory << IOFormatMemSep::all << &blah << IOCtrl::endl;
/*OUTPUT:
39 30 00 00 00 00 00 00 | ad 1c 78 ba 35 ff 5e 40 | 48 65 6c 6c 6f 2c 20 77 | 6f 72 6c␣
↪64 21 00 00 00
*/
```

You can also read memory from a void pointer, though you must specify the number of bytes to read using `IOMemReadSize()`.

> **Warning:** This feature must be used with caution, as reading too many bytes can trigger segfaults or any number of memory errors. Use the sizeof operator in the read_bytes() argument to prevent these types of problems. (See code).

Bitset

IOChannel is able to intelligently output the contents of any bitset. It temporarily forces use of the `IOFormatPtr::memory` flag to ensure proper output.

One may use any of the `IOFormatMemSep::` flags to control the style of output. By default, `IOFormatMemSep::none` is used.

```
bitset<32> foo = bitset<32>(12345678);
ioc << IOFormatMemSep::all << foo << IOCtrl::endl;
/* OUTPUT:
4e 61 bc 00
*/
```

Formatting Objects

If you find yourself regularly using particular formatting flags (`IOFormat...::`), you can store them in an IOFormat object for reuse. Flags are passed into the `IOFormat` object with the stream insertion operator (<<), and then the `IOFormat` object itself can be passed to the IOChannel.

```
IOFormat fmt;
fmt << IOFormatTextAttr::bold << IOFormatTextFG::red << IOFormatTextBG::black;

ioc << fmt << "This is bold, red text on a black background." << IOCtrl::endl;

ioc << fmt << IOFormatBG::blue << "This is bold, red text on a blue background."
    << IOCtrl::endl;
```

As you can see, anything passed to the IOChannel *after* the `IOFormat` object overrides prior options.

IOFormat supports all the flags beginning with `IOFormat...`.

Stream Control

There are multiple enums for controlling IOChannel's output.

For example, one might want to display progress on the same line, and then move to a new line for a final message. This can be accomplished via…

```
ioc << "Let's Watch Progress!" << IOCtrl::endl;
ioc << fg_blue << ta_bold;
for(int i=0; i<100; i++)
{
    //Some long drawn out code here.
    ioc << i << "%" << IOCtrl::sendc;
}
ioc << io_endl;
ioc << "Wasn't that fun?" << io_endl;

/* FINAL OUTPUT:
Let's Watch Progress!
100%
Wasn't that fun?
*/
```

The complete list of stream controls is as follows. Some notes…

- EoM indicates "End of Message", meaning IOChannel will broadcast the message at this point.
- n is a newline.
- r is simply a carriage return (move to start of current line).
- Clear means all formatting flags are reset to their defaults.
- Flush forces stdout to refresh. This is generally necessary when overwriting a line or moving to a new line after overwriting a previous one.

Command	EoM	Clear	r	n	Flush
`IOCtrl::clear`		X			
`IOCtrl::flush`					X
`IOCtrl::end`	X	X			
`IOCtrl::endc`	X	X	X		X
`IOCtrl::endl`	X	X		X	X
`IOCtrl::send`	X				
`IOCtrl::sendc`	X		X		X
`IOCtrl::sendl`	X			X	X
`IOCtrl::r`			X		
`IOCtrl::n`				X	

Cursor Movement

IOChannel can move the cursor back and forth on ANSI-enabled terminals using the *IOCursor::left* and *IOCursor::right* flags.

```
std::string buffer;
ioc << "Hello, world!"
            << IOCursor::left
            << IOCursor::left
            << IOCtrl::end;
std::getline(std::cin, buffer);

/* Will now wait for user input, while displaying "Hello, world!"
 * with the cursor highlighting the 'd' character.
 */
```

Important: Currently, only ANSI is used. Windows support, formatting-removed, and an easy-to-parse formatting flag system for custom outputs will be added soon.

Internal Broadcast Settings (Echo)

IOChannel can internally output to either `printf()` or `std::cout` (or neither). By default, it uses printf(). However, as stated, this can be changed.

IOChannel's internal output also broadcasts all messages by default. This can also be changed.

These settings are modified by passing a `IOEchoMode::` flag to the `configure_echo()` member function.

```
//Set to use `std::cout`
ioc.configure_echo(IOEchoMode::cout);

//Set to use `printf` and show only error messages (any verbosity)
ioc.configure_echo(IOEchoMode::printf, IOVrb::tmi, IOCat::error);

//Set to use `cout` and show only "quiet" verbosity messages.
ioc.configure_echo(IOEchoMode::cout, IOVrb::quiet);

//Turn off internal output.
ioc.configure_echo(IOEchoMode::none);
```

External Broadcast with Signals

One of the primary features of IOChannel is that it can be connected to multiple outputs using signals. Examples of this might be if you want to output to a log file, or display messages in a console in your interface.

Main Signal (signal_all)

The main signal is `signal_all`, which requires a callback function of the form `void callback(std::string, IOVrb, IOCat)`, as seen in the following example.

```
//This is our callback function.
void print(std::string msg, IOVrb vrb, IOCat cat)
{
    //Handle the message however we want.
    std::cout << msg;
}

//We connect the callback function to `signal_all` so we get all messages.
ioc.signal_all.add(&print);
```

Category Signals (signal_c_...)

Almost all categories have a signal: `signal_c_normal`, `signal_c_warning`, `signal_c_error`, `signal_c_testing`, and `signal_c_debug`.

Note: `IOCat::all` is used internally, and does not have a signal. Use `signal_all` instead.

The callbacks for category signals require the form `void callback(std::string, IOVrb)`. Below is an example.

```
//This is our callback function.
void print_error(std::string msg, IOVrb vrb)
{

//Handle the message however we want.
std::cout << msg;

}

//We connect the callback function to signal_c_error to get only error messages.
ioc.signal_c_error.add(&print_error);
```

Verbosity Signals (signal_v_...)

Each verbosity has a signal: `signal_v_quiet`, `signal_v_normal`, `signal_v_chatty`, and `signal_v_tmi`. A signal is broadcast when any message of that verbosity or lower is transmitted.

The callbacks for verbosity signals require the form `void callback(std::string, IOCat)`. Below is an example inside the context of a class.

```
class TestClass
{
    public:
        TestClass(){}
        void output(std::string msg, IOCat cat)
        {
```

(continues on next page)

(continued from previous page)

```
            //Handle the message however we want.
            std::cout << msg;
        }
        ~TestClass(){}
};

TestClass testObject;
ioc.signal_v_normal.add(&testObject, TestClass::output)
```

1.8.5 Flag Lists

Category (`IOCat::`)

Flag	Use
`IOCat::none`	No category; **NEVER broadcasted**. Does not have a correlating signal.
`IOCat::normal`	The default value - anything that doesn't fit elsewhere.
`IOCat::warning`	Warnings, but not necessarily errors.
`IOCat::error`	Error messages.
`IOCat::debug`	Debug messages, such as variable outputs.
`IOCat::testing`	Messages in tests. (Goldilocks automatically suppresses these during benchmarking.)
`IOCat::all`	All message categories. Does not have a correlating signal.

Cursor Control (`IOCursor::`)

Flag	Use
`IOCursor::left`	Moves the cursor left one position.
`IOCursor::right`	Moves the cursor right one position.

Echo Mode (`IOEchoMode::`)

Note: These cannot be passed directly to IOChannel.

Flag	Use
`IOEchoMode::none`	No internal output.
`IOEchoMode::printf`	Internal output uses `printf()`.
`IOEchoMode::cout`	Internal output uses `std::cout`.

Base/Radix Format (`IOFormatBase::`)

Flag	Base
`IOFormatBase::bin`	2
`IOFormatBase::b2`	2
`IOFormatBase::ter`	3
`IOFormatBase::b3`	3
`IOFormatBase::quat`	4
`IOFormatBase::b4`	4
`IOFormatBase::quin`	5
`IOFormatBase::b5`	5
`IOFormatBase::sen`	6
`IOFormatBase::b6`	6
`IOFormatBase::sep`	7
`IOFormatBase::b7`	7
`IOFormatBase::oct`	8
`IOFormatBase::b8`	8
`IOFormatBase::b9`	9
`IOFormatBase::dec`	10
`IOFormatBase::b10`	10
`IOFormatBase::und`	11
`IOFormatBase::b11`	11
`IOFormatBase::duo`	12
`IOFormatBase::doz`	12
`IOFormatBase::b12`	12
`IOFormatBase::tri`	13
`IOFormatBase::b13`	13
`IOFormatBase::tetra`	14
`IOFormatBase::b14`	14
`IOFormatBase::pent`	15
`IOFormatBase::b15`	15
`IOFormatBase::hex`	16
`IOFormatBase::b16`	16
`IOFormatBase::b17`	17
`IOFormatBase::b18`	18
`IOFormatBase::b19`	19
`IOFormatBase::vig`	20
`IOFormatBase::b20`	20
`IOFormatBase::b21`	21
`IOFormatBase::b22`	22
`IOFormatBase::b23`	23
`IOFormatBase::b24`	24
`IOFormatBase::b25`	25
`IOFormatBase::b26`	26
`IOFormatBase::b27`	27
`IOFormatBase::b28`	28
`IOFormatBase::b29`	29
`IOFormatBase::b30`	30
`IOFormatBase::b31`	31
`IOFormatBase::b32`	32

continues on next page

Table 1 – continued from previous page

Flag	Base
`IOFormatBase::b33`	33
`IOFormatBase::b34`	34
`IOFormatBase::b35`	35
`IOFormatBase::b36`	36

Boolean Format (`IOFormatBool::`)

Flag	Use
`IOFormatBool::lower`	Lowercase - “true” or “false”
`IOFormatBool::upper`	Uppercase - “True” or “False”
`IOFormatBool::caps`	All caps - “TRUE” or “FALSE”
`IOFormatBool::num`	Binary numerals - “0” or “1”
`IOFormatBool::scott`	“Yea” or “Nay”

Char Value (`IOFormatCharValue::`)

Enum	Action
`IOFormatCharValue::as_char`	Output chars as ASCII characters.
`IOFormatCharValue::as_int`	Output chars as integers.

Memory Separators (`IOFormatMemSep::`)

Enum	Action
`IOFormatMemSep::no`	Output memory dump as one long string.
`IOFormatMemSep::byte`	Output memory dump with spaces between bytes.
`IOFormatMemSep::word`	Output memory dump with bars between words (8 bytes).
`IOFormatMemSep::all`	Output memory dump with spaces between bytes and bars between words.

Numeral Case (`IOFormatNumCase::`)

Enum	Action
`IOFormatNumCase::lower`	Print all letter digits as lowercase.
`IOFormatNumCase::upper`	Print all letter digits as uppercase.

Pointer Format (`IOFormatPtr::`)

Enum	Action
`IOFormatPtr::value`	Print the value at the address.
`IOFormatPtr::address`	Print the actual memory address.
`IOFormatPtr::memory`	Dump the hexadecimal representation of the memory at the address.

Scientific Notation Format (`IOFormatSciNotation::`)

Enum	Action
`IOFormatSciNotation::none`	No scientific notation.
`IOFormatSciNotation::auto`	Automatically select the best option.
`IOFormatSciNotation::on`	Force use of scientific notation.

Warning: `IOFormatSciNotation::none` has been known to cause truncation in very large and very small values, regardless of significands.

Significands(`IOFormatSignificands()`)

`IOFormatSignificands(n)` where `n` is the significands, as an integer representing the number of significands.

Text Attributes(`IOFormatTextAttr::`)

Enum	Action
`IOFormatTextAttr::none`	Turn off all attributes.
`IOFormatTextAttr::bold`	**Bold text**.
`IOFormatTextAttr::underline`	Underlined text.
`IOFormatTextAttr::invert`	Invert foreground and background colors.

Text Background Color(`IOFormatTextBG::`)

Enum	Action
`IOFormatTextBG::none`	Default text background.
`IOFormatTextBG::black`	Black text background.
`IOFormatTextBG::red`	Red text background.
`IOFormatTextBG::green`	Green text background.
`IOFormatTextBG::yellow`	Yellow text background.
`IOFormatTextBG::blue`	Blue text background.
`IOFormatTextBG::magenta`	Meganta text background.
`IOFormatTextBG::cyan`	Cyan text background.
`IOFormatTextBG::white`	White text background.

Text Foreground Color(`IOFormatTextFG::`)

Enum	Action
`IOFormatTextFG::none`	Default text foreground.
`IOFormatTextFG::black`	Black text foreground.
`IOFormatTextFG::red`	Red text foreground.
`IOFormatTextFG::green`	Green text foreground.
`IOFormatTextFG::yellow`	Yellow text foreground.
`IOFormatTextFG::blue`	Blue text foreground.
`IOFormatTextFG::magenta`	Meganta text foreground.
`IOFormatTextFG::cyan`	Cyan text foreground.
`IOFormatTextFG::white`	White text foreground.

Memory Dump Read Size (`IOMemReadSize()`)

`IOMemReadSize(n)` where `n` is the number of bytes to read and print, starting at the memory address. **Only used with void pointers.**

Warning: Misuse triggers undefined behavior, including SEGFAULT. Use with caution.

Verbosity (`IOVrb::`)

Enum	Use
`IOVrb::quiet`	Only essential messages and errors. For normal end-use. Shipping default.
`IOVrb::normal`	Common messages and errors. For common and normal end-user testing.
`IOVrb::chatty`	Most messages and errors. For detailed testing and debugging.
`IOVrb::tmi`	Absolutely everything. For intense testing, detailed debugging, and driving the developers crazy.

1.9 Onestring

1.9.1 What is Onestring?

`Onestring` is a multi-sized, Unicode-compatible replacement for `std::string`. Onestring contains all the basic functions found in `std::string` while optimizing the use of dynamic allocation wherever possible. To handle Unicode, each Onestring is made of Onechars, which are enhanced characters.

1.9.2 Using a Onestring

Creating a Onestring

You can create a Onestring with characters initialized with the = operator or leave it blank.

```
// Empty Onestring
Onestring blankString;

// Containing Onechars
Onestring fullString = "these are Unicode Characters";
```

You can also use the = operator to create a new Onestring with an existing Onestring.

```
// Create the first Onestring
Onestring firstString = "copy me";

// Containing Onechars
Onestring secondString = firstString;

// secondString now contains "copy me".
```

Adding to a Onestring

+, +=

The += and + operators add additional Unicode characters at the end of the existing `Onestring`.

Using +=

```
//Declare a Onestring
Onestring more = "apple";

// Use += to append an "s"
more += "s";
// `more` is now "apples"

// Use + to append "!"
more = more + "!"
// `more` is now "apples!"
```

`append()`, `push_back()`

Alternatively, you can use the functions `append()` and `push_back()` to add additional Unicode characters at the end of the existing `Onestring`.

Using `append()`…

```
// Declare a Onestring
Onestring to_add = "apple";

// Add "s" to `to_add`
```

(continues on next page)

(continued from previous page)

```
to_add.append('s');

// `to_add` is now "apples"

// add "!" to `to_add`
to_add.push_back('!');

// `to_add` is now "apples!"
```

insert()

insert() allows you to insert a Unicode character into an existing Onestring at a given position.

```
// Declare a Onestring
Onestring alphabet = "abcdf";

// Insert a value into `alphabet`
// The first value in the function refers to the index to be inserted
// The second value refers to the value to be inserted
alphabet.insert(4, 'E');

// `alphabet` is now "abcdEf"
```

Removing from a Onestring

clear()

clear() erases the contents of a Onestring.

```
// Declare a Onestring
Onestring sleeve = "something";

// Clear the contents of `sleeve`
sleeve.clear();

// `sleeve` is now empty
```

pop_back()

pop_back() removes the last Unicode character in the Onestring

```
// Declare a Onestring
Onestring alphabet = "abcdef";

// Remove the last element from `alphabet`
alphabet.pop_back();

// `alphabet` is now "abcde"
```

Accessing Elements of A Onestring

[]

The [] operator acceses a Unicode character at a given location in a Onestring.

```
// Declare a Onestring
Onestring test = "hello";

// Check what character is at position 1 in `test`
test[1];

// The operator returns 'e'
```

at()

The at() function can be used as an alternative to [].

```
// Declare a Onestring
Onestring alphabet = "abcdef";

// Find the Onechar at position 3 in `alphabet`
alphabet.at(3);

// The function returns "d"
```

Comparing Onestrings

==

The == operator checks for equivalence between two strings and returns a boolean with the result.

```
// Declare two Onestring
Onestring dogs = "fun";
Onestring cats = "mean";

// Check for equivalence
if (dogs == cats)
{
  return dogs;
}
else
{
  return cats;
}

// This statement would return `cats`

// Reassign `cats`
cats = "fun";
```

(continues on next page)

(continued from previous page)

```
// Check for equivalence
if (dogs == cats)
{
  return dogs;
}
else
{
  return cats;
}

// The statement now returns `dogs`.
```

`equals()`

`equals()` can also be used to check for equality.

```
// Declare a Onestring
Onestring checker = "red";

// Compare with another Onestring
checker.equals("black");

// The function returns false

// Compare again
checker.equals("red");

// The function returns true
```

`<, >, <=, >=`

The <, >, <=, and >= operators compare string sizes, with the first relative to the second. < is less than, > is greater than, and <= and >= are less than or equal to, and greater than or equal to, respectively.

```
// Delcare three Onestrings
Onestring first = "one";
Onestring second = "two";
Onestring third = "three";

// Compare `first` to `second`
if (first < second)
{
  return first;
}
else
{
  return second;
}
```

(continues on next page)

(continued from previous page)

```
// The statement returns `second`

if (first <= second)
{
  return first;
}
else
{
  return second;
}

// The statement now returns `first`

if (third > second)
{
  return third;
}
else
{
  return second;
}

// Finally, this statement returns `third`
```

Other Functions

`empty()`

`empty()` checks to see if a Onestring is empty. The function returns true if it is empty, and false if it is not.

```
// Declare a Onestring
Onestring toyBox;

// Check to see if `toybox` empty
toyBox.empty();

// The function returns true

// Assign values to `toyBox`
toyBox = "basketball"

// Check again to see if its empty
toyBox.empty();

// This time, the function returns false.
```

getType()

getType() returns a boolean that represents either a Onestring or a QuickString.

size()

size() returns the number of characters that make up the Onestring.

```
// Declare a Onestring
Onestring checker = "red";

// check the size of `checker`
checker.size();

// The function will return 3
```

substr()

substr() creates a new substring based on a range of characters in an exisiting Onestring

```
// Declare a Onestring
Onestring full = "monochromatic"

// Declare a new Onestring
// Using `full`
// and substr
Onestring partial = full.substr(0,3);

// The new Onestring `partial` contains the word "mono".
// The numbers in the function call refer to
// the range to be copied into the new string.
```

swap()

swap() switches the contents of the current Onestring with another. The two Onestrings must be of the same size.

```
// Declare two Onestrings
Onestring first = "primary";
Onestring second = "secondary";

// Swap `primary` and `secondary`
first.swap(second);

// `first` now reads "secondary".
// `second` now reads "primary"
```

1.10 Pool

1.10.1 What is Pool?

Pool is a generic implementation of the object pool design pattern. It can store up to approximately 4 billion objects of the same type. All dynamic allocation is performed up front.

Performance Considerations

At present, Pool is actually a little *slower* than dynamic allocation on some modern systems, due to some overhead when deleting an object. While future versions may allevitate this, it is important to note that Pool's main advantage is in providing safer dynamic memory access.

It is possible that Pool may offer better performance in environments where dynamic allocation is extremely expensive. Running a comparative benchmark between Goldilocks tests `P-tB1601` and `P-tB1601*` will indicate whether there exist any mid-execution performance gains from using Pool in your particular environment.

Technical Limitations

Pool 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`

Including Pool

To include Pool, use the following:

```
#include "pawlib/pool.hpp"
```

1.10.2 Creating a Pool

A `Pool` object can be created by whatever means is convenient. It handles its own dynamic allocation internally when first created.

When the Pool is created, you must define its type and maximum size.

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

// Define a pool storing up to 500 Particle objects.
Pool<Particle> particles(500);

// Define a pool storing up to 100 Enemy objects.
Pool<Enemy>* baddies = new Pool<Enemy>(500);
```

Failsafe

By default, Pool will throw an `e_pool_full` exception if you try to add an object when the Pool is full. However, there are situations where you may not want this behavior. You can initialize Pool in **failsafe mode** to have Pool fail silently in this situation.

Note: Even in failsafe mode, Pool will still throw its other exceptions.

```
// Define a 500-object Particle pool in failsafe mode.
Pool<Particle> pool(500, true);
```

Failsafe mode does introduce another situation where an exception may be thrown. If the Pool is full, adding an object will return an invalid reference. One must either ensure that the reference is valid before using it, or catch the `e_invalid_ref` exception on `Pool::access()` and `Pool::destroy()`.

1.10.3 Using Pool

Pool References

One of the main features of Pool is its safety. When used properly, you don't run the risk of memory errors, segmentation faults, or other undefined behaviors generally associated with dynamic allocation and pointers.

Instead of pointers, Pool offers pool references (`pool_ref`). You can think of these as "keys" - you cannot access anything in a Pool without the appropriate `pool_ref`.

Each pool reference is associated with a particular object in the Pool. When the object is destroyed, all the references to that object are invalidated to prevent undefinied behavior.

Important: The most important thing to remember is that you should **never use pointers** to access objects within the Pool.

Each reference is also directly associated with a single Pool. You cannot use a reference associated with one Pool to access an object in another Pool, even if both Pools are initialized identically.

Invalid References

A reference is invalid if:

- It refers to a destroyed object.
- It has only been created with its default constructor, with no assignment to it from `Pool::create()`. (It will also be considered Foreign to every Pool.)
- It was returned from `Pool::create` in failsafe mode, and the pool was full.

You can check if a reference is invalid by using the `pool_ref::invalid()` function.

```
// This would be an invalid reference, since no object was created.
pool_ref<Foo> rf;

// This would return true.
rf.invalid();
```

Object Compatibility

To store an object in Pool, it **must** have a default constructor and a copy constructor. The copy constructor is used to provide indirect access to all the other constructors for the object.

Adding Objects

There are several ways to add a new object to the pool. In each one, the important thing is that you wind up with a `pool_ref` object. Watch this! It is not possible to access or destroy an object within the Pool without its reference.

All of the following methods are valid…

```
/* Foo contains a default constructor, a copy constructor, and a
 * constructor that accepts an integer. */
class Foo;
Pool<Foo> pool(10);

try
{
    pool_ref<Foo> rf1 = pool.create();
    pool_ref<Foo> rf2 = pool.create(Foo(5));
    pool_ref<Foo> rf3(pool);
    pool_ref<Foo> rf4(pool, Foo(42));
}
catch(e_pool_full)
{
    // Handle the exception.
}
```

Let's break those down further.

The first method is to define a `pool_ref` object, and assign the result of `Pool::create` function to it.

```
// Uses default constructor.
pool_ref<Foo> rf1 = pool.create();
// Uses copy constructor to indirectly access another constructor.
pool_ref<Foo> rf2 = pool.create(Foo(5));
```

You can also create the object by passing the Pool directly into the `pool_ref`'s constructor. This calls Pool.create() implicitly, so if the Pool is not in failsafe mode, you still need to watch out for the `e_pool_full` exception.

```
// Uses default constructor.
pool_ref<Foo> rf3(pool);
// Uses copy constructor to indirectly access another constructor.
pool_ref<Foo> rf4(pool, Foo(42));
```

Accessing Objects

Objects are accessed within a Pool using the `pool_ref` you got when creating the object.

```
// The class Foo has a function "say()"
class Foo;
Pool<Foo> pool(10);
pool_ref<Foo> rf1 = pool.create();

/* We use the pool reference to access the object. Then we can
 * interact with the object directly. */
pool.access(rf1).say();
```

The `Pool.access()` function can throw two different exceptions.

The most common is `e_pool_invalid_ref`. This is thrown when an invalid pool reference is passed.

```
Pool<Foo> pool(10);
pool_ref<Foo> emptyRef;
pool.access(emptyRef); // throws e_pool_invalid_ref
```

The other is `e_pool_foreign_ref`, which is thrown if a pool reference that belongs to another pool is passed.

```
Pool<Foo> pool(10);
Pool<Foo> otherPool(10);
pool_ref<Foo> foreignRef = otherPool.create();
pool.access(foreignRef); // throws e_pool_foreign_ref
```

Destroying Objects

When you're done with an object, you can remove it from the Pool. This frees up space for another object to be created in its place later. To destroy an object, simply pass a reference to it into `Pool::destroy()`.

It's important to note that if you have multiple references to the same object, they will all be invalidated when the object is destroyed.

```
Pool<Foo> pool(10);
pool_ref<Foo> thing(pool);
pool_ref<Foo> copyOfThing = thing;

// We destroy the object.
pool.destroy(thing);

pool.access(copyOfThing); // This will now throw e_pool_invalid_ref
```

`Pool::destroy()` can throw `e_pool_invalid_ref` or `e_pool_foreign_ref` under the same circumstances as with `Pool::access()`.

1.10.4 Exceptions

e_pool_full

Cause: The Pool is full.

Thrown By: `Pool::create()` (in non-failsafe mode)

e_pool_invalid_ref

Cause: An invalid reference was used.

Thrown By: `Pool::access()`, `Pool::destroy()`

e_pool_foreign_ref

Cause: A reference from another pool was used, or a reference created with its default constructor and not assigned to by `Pool::create()`.

Thrown By: `Pool::access()`, `Pool::destroy()`

e_pool_reinit

Cause: Attempting to reinitialize an an object that already exists.

Thrown By: Internal - shouldn't happen.

1.10.5 Examples

Enemy Pool

```
// Let's define an Enemy class for our example.
class Enemy
{
    Enemy();
    Enemy(const Enemy& cpy);
    Enemy(std::string);
    void attack();
    void hurt(int);
    void die();
    int health;
    ~Enemy();
};

// Create our pool.
Pool<Enemy> baddies(500);

// This function would return the damage from the player's move.
int getPlayerMove();

void fightSkeleton()
```

(continues on next page)

(continued from previous page)

```
{
    /* Since our pool is not in failsafe mode, we must be on the lookout
     * for the `e_pool_full` exception that create() can throw.*/
    try
    {
        /* Create a new Enemy object in the pool. This uses Enemy's copy
         * constructor to give access to the constructor accepting a string. */
        pool_ref<Enemy> skeleton = pool.create(Enemy("Skeleton"))
    }
    catch(e_pool_full)
    {
        // We couldn't create the enemy, so just quit.
        return;
    }

    while(baddies.access(skeleton).health > 0)
        // We order our skeleton to attack the player.
        baddies.access(skeleton).attack();
        // The player hurts the skeleton.
        baddies.access(skeleton).hurt(getPlayerMove());
    }

    // Make the skeleton character die.
    baddies.access(skeleton).die();
    // Destroy the skeleton object in the pool.
    baddies.destroy(skeleton);
}
```

Particle System

```
class Particle
{
    Particle();
    Particle(const Particle& cpy);
    Particle(int, int);
    emit();
};

/* For this example, we'll define a failsafe pool, so we don't have to
 * try/catch our creation of objects. */
Pool<Particle> particles(2000, true);

// Define a particle in the pool using its default constructor.
pool_ref<Particle> particle(particles);

void particleEffect(int type, int speed, int count)
{
    /* One design pattern might be to generate a lot of particles in a loop.
     * In this example, we'll store them in a FlexArray. */
    FlexArray<pool_ref<Particle>> smoke_effect;
```

(continues on next page)

(continued from previous page)

```
    for(int i=0; i<count; ++i)
    {
        /* Define a particle in the pool using its copy constructor, which
         * gives us access to the constructor that accepts an integer. */
        smoke_effect.push(pool_ref<Particle>(particles, Particle(type, speed));
    }

    /* Let's emit our particles. */
    for(int i=0; i<count; ++i)
    {
        // Ensure the particle exists before emitting it.
        if(!smoke_effect[i].invalid())
        {
            particles.access(smoke_effect[i]).emit();
        }
    }

    /* Destroy the particles when we're done. */
    for(int i=0; i<count; ++i)
    {
        // Ensure the particle exists before destroying it.
        if(!smoke_effect[i].invalid())
        {
            particles.destroy(smoke_effect[i]);
        }
    }
}
```

1.11 Standard Utilities

These are common utility functions which are used by many other classes in PawLIB.

Not all of the functions are documented below, only those we believe will be useful to PawLIB users.

1.11.1 Including Standard Utilities

To include Standard Utilities, use the following:

```
#include "pawlib/stdutils.hpp"
```

1.11.2 Integer to std::string [`itos()`]

We can convert any integer data type, signed or unsigned, to a std::string using `itos()`.

`itos()` converts the integer to a std::string. It accepts three arguments, two of which are required:

- the integer to convert,
- the base you're working in, represented as an integer (default=10),
- whether to represent digits greater than 9 as uppercase (default=false)

```
// The integer to convert.
int foo = -16753;

/* Convert the float to a std::string. We're passing all the arguments,
 * even though only the first two are required, for the sake of example.
 */
std::string foo_s = stdutils::itos(foo, 10, false);

// Print out the std::string.
ioc << foo_s << IOCtrl::endl;

// OUTPUT: -16753
```

Important: Enumerations are not implicitly cast to ints with this function. Therefore, you must `static_cast<int>()` any enumeration variables before passing them to this function.

1.11.3 Integer to C-String [`itoa()` & `intlen()`]

We can convert any integer data type, signed or unsigned, to a C-string using `itoa()` and `intlen()`.

`intlen()` returns the character count necessary to represent the integer as a string. It accepts three arguments, two of which are required:

- the integer being measured,
- the base you're working in, represented as an integer, and
- whether to include space for the sign (default=true).

`itoa()` converts the integer to a C-string. It accepts five arguments, two of which are required:

- the C-string to write to,
- the integer to convert,
- the base you're working in, represented as an integer (default=10),
- the number of characters in the integer (default=0, meaning it will be internally calculated), and

- whether to represent digits greater than 9 as uppercase (default=false)

Combining these functions allows us to flexibly convert any integer to a C-string, without having to know anything in advance.

```
// The integer to convert.
int foo = -16753;

/* We use the intlen function to determine the size of our C-string
 * Note that we are adding one to leave room for our null terminator. */
char foo_a[stdutils::intlen(foo, 10, true) + 1];

/* Convert the integer to a C-string. We're passing all the arguments,
 * even though only the first two are required, for the sake of example.
 * 0 for the fourth argument (size) causes the function to internally
 * calculate the size of the integer again, which is another call to
 * intlen(). You might save some execution time by specifying this instead.
 */
stdutils::itoa(foo_a, foo, 10, 0, false);

// Print out the C-string.
ioc << foo_a << IOCtrl::endl;

// OUTPUT: -16753
```

Note: It is generally going to be more practical to use `itos()` instead.

Important: Enumerations are not implicitly cast to ints with this function. Therefore, you must `static_cast<int>()` any enumeration variables before passing them to this function.

1.11.4 Float to String [`ftos()`]

We can convert any floating-point number data type (float, double, or long double) to a std::string using *ftos()*.

We need to specify the number of significands - in our case, the number of digits after the decimal point - to work with. Because of the nature of floating point numbers, the conversion is *not* perfect, as we'll see shortly.

`ftos()` converts the number into a C-string. It accepts three arguments, one of which are required:

- the number to convert,
- the number of significands (default=14), and
- whether to use scientific notation - 0=none, 1=automatic, and 2=force scientific notation (default=1).

```
// The integer to convert.
float foo = -65.78325;

/* Convert the float to a std::string. */
std::string foo_s = stdutils::ftos(foo, 5, 1);

// Print out the std::string.
```

(continues on next page)

(continued from previous page)

```
ioc << foo_s << IOCtrl::endl;

// OUTPUT: -65.78324
```

As you can see, the output is off by 0.00001. Again, this is because of how floating point numbers work, and the number of significands we specified. If we were to raise the significands to the default 14, our output would actually have been "-65.78324891505623".

1.11.5 Float to C-String [`ftoa()` & `floatlen()`]

We can convert any floating-point data type (float, double, or long double) to a C-string using `ftoa()` and `floatlen()`.

In both functions, we need to specify the number of significands - in our case, the number of digits after the decimal point - to work with. Because of the nature of floating point numbers, the conversion is *not* perfect, as we'll see shortly.

`floatlen()` returns the character count necessary to represent the floating-point number as a string. It accepts three arguments, only one of which is required:

- the number to count the characters in,
- the number of significands (default=14), and
- whether to count the symbols (default=true)

`ftoa()` converts the number into a C-string. It accepts four arguments, two of which are required:

- the C-string to write to,
- the number to convert,
- the number of significands (default=14), and
- whether to use scientific notation - 0=none, 1=automatic, and 2=force scientific notation (default=1).

```
// The integer to convert.
float foo = -65.78325;

/* Convert the float to a std::string. */
std::string foo_s = stdutils::ftos(foo, 5, 1);

// Print out the std::string.
ioc << foo_s << IOCtrl::endl;

// OUTPUT: -65.78324
```

As you can see, the output is off by 0.00001. Again, this is because of how floating point numbers work, and the number of significands we specified. If we were to raise the significands to the default 14, our output would actually have been "-65.78324891505623".

Note: It is generally going to be more practical to use `ftos()` instead.

1.11.6 Split String By Tokens [stdsplit]

This will split a `std::string` by a given token and store it in a `std::vector`. The token will be stripped out in the process.

Later versions of this will support Onestring and FlexArray.

```
std::string splitMe = "What if we:Want to split:A string:By colons?";
std::vector<std::string> result;

stdutils::stdsplit(splitMe, ":", result);
// result now contains "What if we", "Want to split", "A string", "By colons?"
```

1.11.7 Reverse C-String [strrev()]

This will reverse a given C-string in place, overriding the string.

```
char foo[14] = "Hello, world!";
stdutils::strrev(foo);
ioc << foo << IOCtrl::endl;
```

1.12 PawLIB Tester Console

PawLIB also provides a tester application, which will allow you to run any of Goldilocks tests and benchmarks using GoldilocksShell.

In addition to the various tests, we've provided comparative tests in our suites, to facilitate performance comparisons of PawLIB against the standard alternatives. See *Benchmarker Output* for details on reading the benchmarker statistics and results.

See *Using PawLIB* for instructions on how to build the PawLIB Tester.

Once it's built, you can run the tester from within the PawLIB repository via `./tester`.

For PawLIB test and suite ID naming conventions, see *PawLIB Tests*.

1.12.1 Interactive Mode

We can start Interactive Mode by running the tester application without arguments, via `./tester`. Type commands at the >> prompt.

All commands are detailed under *Interactive Mode*.

1.12.2 Command-Line Mode

We can run tests and suites by passing arguments to our `./tester` application. This is especially useful if you want to run tests in a automated manner, such as with a continuous integration system.

You can get help via `./tester --help`.

Multiple commands may be run in a single line. They will be executed in order.

All commands are detailed under *Command Line Argument Mode*.

1.13 PawLIB Tests

For instructions on using the PawLIB Tester, see *PawLIB Tester Console*.

1.13.1 Test Namespaces

Because we use Goldilocks for multiple projects at MousePaw Media, we follow certain conventions for test and suite IDs.

The Live-In Testing Standard defines the first part of the ID. For example,

- `P-` refers to the PawLIB project.
- `s` is a suite, while `t` is a test.
- `B` is a "behavior" test, and `S` is a "stress" test, etc.

The first digit indicate the major sector of PawLIB the suite and its tests are related to. The second digit is the specific sector, usually a single class.

The current sectors are delineated exhaustively in the table below. Not all sectors have tests implemented yet. `x` indicates that another number is needed.

ID	Sector
0x	Data Types
01	Trilean
1x	Data Structures
10	FlexArray
11	FlexMap
12	FlexQueue
13	FlexStack
14	SpeedList
15	FlexBit
16	Pool
20	IOChannel
30	PawSort
4x	Onestring (Sector)
40	Onestring
41	Onechar
5x	Blueshell
6x	Utilities

Any subsequent digits indicate the test number. A number may be shared between behavior and stress tests; both use the same implementation, but vary in their variables (such as iterations).

1.14 Finding Support

If you have any trouble with MousePaw Media projects, we invite you to contact us!

1.14.1 Supported Topics

We officially offer support for the following:

- Compiling and linking to PawLIB.
- Using PawLIB.
- Using the PawLIB Tester.
- Building and linking to MousePaw Media's `libdeps` repository with supported compilers.

We only support use of PawLIB using the supported compilers and environment (see *Environment and Dependencies*).

Note: Due to the complexities of running GCC and Clang on Microsoft Windows, we do not necessary offer support for that operating system. If you are certain that you are running a supported environment on Microsoft correctly, you ARE still welcome to contact us.

Questions about use of CPGF should be directed to that project instead (cpgf.org).

Community support is available for all C++-related questions via the ##c++-friendly chatroom on Freenode IRC.

1.14.2 Contacting Support

- Email: `support@mousepawmedia.com`
- Freenode IRC: #mousepawgames
- Phabricator Ponder on DevNet (available 7am-10pm PST / 1400-0500 UTC).

1.14.3 Bug Reports and Feature Requests

If you encounter a bug in PawLIB, or would like to see a feature added, we encourage you to file a report on DevNet Phabricator Maniphest.

Warning: We do **not** monitor pull requests or issues on GitHub!

To file a bug report or feature request:

1. Go to DevNet During hours (7am-10pm PST / 1400-0500 UTC), click *Connect Now*.
2. Click *Phabricator* from the main menu.
3. Sign in using your GitHub account. If this is your first time...
   - Authorize the *DevNet [MousePaw Media]* OAuth App.
   - Thoroughly read and agree to the *Community Rules*. We kept those concise, to make them easier to read and understand.
4. On the Phabricator menu at left, select *Maniphest*.

5. In the upper-right corner, select *Create Task* and choose either *Bug Report* or *Feature Request*.
6. See the link at the top of that form for instructions on how to craft a useful bug report or feature request.

CHAPTER TWO

INDICES AND TABLES

Note: The index is still a work in progress. If you'd like to help with this, please see our Contribution Guide.

- genindex
- search

INDEX