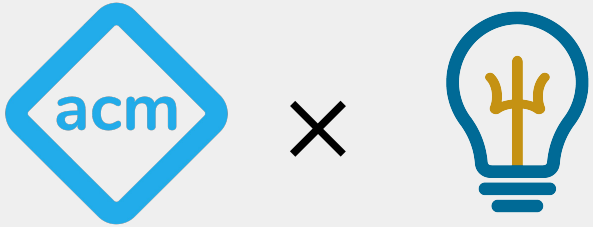

Advanced C++: Traditional

By David Hacker and Nicolas Nebel

History of C++

- Created by Bjarne Stroustrup in 1985, based off “C with classes”, which was based off the C programming language
- Today the standard is maintained by ISO (isocpp.org, https://github.com/cplusplus/draft).
- Standard-compliant cpp compilers implemented by GNU (g++ in GCC), LLVM Developer Group (clang), and Microsoft (Microsoft C/C++ in MSVC).
- No “official” compiler like Java or C#. C++11 marked a new era, changing c++ into a more modern language. New specs every 3 years since: c++14, 17, and 20 coming up.

Online References

- https://cppreference.com
  - Avoid rival www.cplusplus.com/ which is persistently out-of-date and in need of an overhaul
  - Preferably, scroll down to the cppreference links on the search results to train Google's algorithm better
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf
  - The latest & greatest working spec for C++
  - 1622 pages
  - May not always be feasible to search through so use cppreference as a default

C++ Package Management

- In short, absolutely terrible.
- No modern package manager built in (didn't exist back then)
- Heavily relies on Linux distribution's package manager
  - apt-get, yum, pacman install C++ libraries in various places
  - You could also download shared libraries yourself or download their source code, compile them, and install.
- Some attempts at rectifying it, but not official in any way.
  - Meson build system has the Wrap package manager
- Stems from how linking is done to find system libraries
  - LD_LIBRARY_PATH is searched exhaustively for a link
  - What happens if library path not in LD_LIBRARY_PATH?
  - What happens if you're on Windows?

Header-Only Files

- One way to approach package management is to avoid libraries altogether (avoid linking process)
- Instead, keep all library code inside a single header file.
  - This header file can be checked into your own source control and `#include`'d in whatever executables need it
  - Examples: Catch2 testing framework (what we are using)
- Pros: Avoids pain of linking, easy to transfer between projects
- Cons: Manual copying required, may not be kept continually up-to-date, header file can be large & messy, forces everything to be inlined, no clear separation of implementation and interface
- The vector class you will be implementing is in `my_vector.hpp`, which is … a header-only file!
  - All functions are inlined (because they are relatively small).

C++ Build Systems

- C++ build systems are different from package management mess
- Build systems allow a C++ project to be built on multiple environments, irrespective of operating system, directory structures.
- A build system, like a compiler, is usually a metaprogram
  - This means that it writes other programs
  - A compiler takes in source code and produces assembly
  - A build system takes in a configuration and produces Makefiles or environment-specific build tools
  - Those Makefiles can then be run to build the C++ source code on the operating system the build system has targeted
- Build systems usually handle linking for you. They may or may not do package management for you (usually not).

The Usefulness of CMake

- CMake is by far one of the most widely used build systems
- CMake takes in a CMakeLists.txt file and produces a Makefile (or whatever Windows uses).
- Inside the CMakeLists.txt file, there are commands to set up shared libraries, static libraries, executables, handle linking, find packages on your system, and more.
- Two steps involved with any project:
  - `cmake` (produces a Makefile)
  - `make` (which then compiles your code)
- CMake can be set up to use modules
  - A module allows CMake to configure how it searches for a library AND allows new commands to be added to CMake
  - We have a Catch2 module to generate testing commands

Out-of-Source Builds

- By default, running `cmake` will produce Makefiles and other build system residue within the current directory
  - This is not ideal because it pollutes our repository. All of the generated code needs to be ignored in source control.
- The solution is to what's called an "out-of-source build"
  - Idea: create a new directory usually called "build"
  - Change into the build directory
  - Run `cmake ..` to tell CMake where the CMakeLists.txt can be found
  - Result: Makefiles and residue are generated inside build directory. Source code remains outside of build directory.
  - Add build directory to .gitignore
  - Can have multiple build directories for debug, release, etc.

STL Refresher - Arrays & Lists (1)

`std::vector<T>`

- Dynamically resizable array
- Constant-time lookup of an index but linear-time insertion/removal (if not at the end)
- Resizes are costly; they force a linear-time reallocation and copy

`std::list<T>`

- Constant-time insertion and removal of elements
- Usually implemented as a doubly-linked list
- Poor cache performance due to linked list nature, can be alleviated slightly by choosing the right allocator

STL Refresher - Arrays & Lists (2)

`std::deque<T>`

- Double-ended queue; arguably the most useful STL container besides vectors and unordered maps
- Insertion/removal from beginning and end is in constant-time
- Lookups are constant-time; can simulate vector, stack, queue, etc.

`std::array<T>`

- A wrapper around fixed size arrays
- Intended to replace C-style array weirdness
- Same performance as regular array but adds iterators, assignment, equality checks, and benefits of an STL container

`vector` implementation

- We can learn a lot of common cpp design idioms by building a simple dynamic array container, similar to `std::vector`.
- There are a few things to keep in mind when writing our implementation:
  - Exception safety
    - No memory leaks or adverse consequences when an exception is thrown
    - Exception-safe, exception-neutral, no-throw/`noexcept`
  - Reducing copy operations and temporary objects
  - Minimize requirements on the object type stored
- We'll start by writing a simple constructor for a dynamic array with the following outline:

my_vector structure


```
template<typename T>
class my_vector
{
private:
     T* d_arr_p;
     size_t d_size;
     size_t d_capacity;
public:
     my_vector(size_t size) {}
     my_vector(const my_vector& other) {}
     ~my_vector() {}

     void swap(my_vector& other) {}
     void reserve(size_t size) {}
     void push_back(const T& elem){}

     my_vector& operator=(const my_vector& other){}
     T& operator[](int idx) {}
}
```

https://github.com/TritonSE/cpp-workshop-1

Write

`my_vector(size_t size)`,
`~my_vector()`

(< 5 min)

my_vector(size_t size)


```
my_vector(size_t size)
: d_arr_p(nullptr), d_capacity(size), d_size(0) // nothing used yet
{
     d_arr_p = new T[size]; // initial allocation
}
```

- `d_capacity` stores the total potential number of entries
- `d_size` stores the number of used slots
- The main line of interest is the initial allocation using **`new`**, which does two things:
  - Allocates enough memory for `d_capacity` elements. If this fails, a `bad_alloc` exception would be thrown
  - Constructs each `T` in every slot. If this fails, `operator delete[]` would be automatically called to free the memory

Alternate implementation


```
my_vector(size_t size)
: d_arr_p(static_cast<T*>(size == 0 ? nullptr : operator new(sizeof(T)*size))
), d_capacity(size), d_size(0) // nothing used yet
{}
```

- Here we aren't calling the default constructors at all, just allocating the space.
- This way, we wouldn't have to worry about the constructor throwing or worrying if the class even has a default constructor.
- However, the values wouldn't be initialized and using them would result in undefined behavior.
- We'll be using this one because it makes later implementations a little easier

~my_vector()


```
~my_vector()
{
     for(int i = 0; i < d_size; i++)
     {
          d_arr_p[i].~T();
     }

     operator delete(d_arr_p);
}
```

- One major flaw we can't get around: if one of the destructors throws, d_arr_p will never be freed.
- Therefore, we must impose a requirement on the class: the destructor must not throw
- This is a good rule for pretty much any class you make in C++, as there are similar problems in almost all containers

Write

`swap(my_vector& other)`,

`my_vector(const my_vector& other)`

(5-10 min)

Hint: just swap the pointers for `swap()` *but do a full copy for the copy constructor. Also,* `std::swap` *exists.*

swap(my_vector& other)


```
swap(my_vector& other) noexcept
{
     using std::swap; // explained later
     swap( d_arr_p, other.d_arr_p );
     swap( d_size, other.d_size );
     swap( d_capacity, other.d_capacity );
}
```

- If you didn't see it in the header or figure it out yourself, this is has a nothrow guarantee.
- Since `c++11` all standard implementations of `std::swap` have been `noexcept`
- Constant runtime, implemented this way in the standard as well
- Any iterators are still valid but not swapped

my_vector(const my_vector& other)

```
my_vector(const my_vector& other)
: my_vector(other.d_size)
// fyi, calling a ctor from another ctor is only legal after
// c++11
{
     while( d_size < other.d_size )
     {
          new (d_arr_p+d_size) T(other.d_arr_p[d_size]);
          d_size++;
     }
}
```

- The syntax `new (<pointer>) <class>(<params>)` is used to construct an object in-place in a specific memory location given by `<pointer>`. Also known as placement-new.

Rule of 3/5

- https://en.cppreference.com/w/cpp/language/rule_of_three
- “If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three”
- By default, these methods work by simply copying over the internal fields of the class. However, in the case of `std::vector` and other containers, this would only result in a shallow copy.
- Since one of these needs to be implemented, all of them should

Write

`reserve(size_t size)`,

`push_back(const T& elem)`

(5-10 min)

Hint: Can these functions use previous functions and each other as helpers?

swap(my_vector& other)


```
reserve(size_t size)
{
     if(size <= d_capacity) { return; }

     my_vector temp(size);
     while(temp.d_size < d_size)
     {
          temp.push_back( d_arr_p[temp.d_size] ); // hol up
     }
     swap(temp);
}
```

- These functions actually use each other as helpers
- Assumes you have the second implementation for `my_vector(size_t size)`

swap(my_vector& other)


```
void push_back(const T& elem)
{
     if(d_size == d_capacity) // grow if necessary
     {
          reserve(d_size * 2 + 1);
     }

     new (d_arr_p+d_size) T(elem);
     ++d_size;
}
```

- These functions actually use each other as helpers
- Not much to explain here

Write

`my_vector& operator=(const my_vector& other),`

`T& operator[](size_t idx)`

(5-10 min)

Hint: Can these functions use previous functions as helpers?

Assignment operator


```
my_vector& operator=(const my_vector& other)
{
     my_vector temp(other);
     swap(temp); // this can't throw
     return *this;
}
```

- This is known as the “copy and swap idiom”, it’s used in several other data structures as a simple way to implement an assignment operator
- Assignment operators are used after an object has been constructed, while copy constructors are used when an object has not been constructed yet

```
my_vector a(10), b(5);
my_vector c = a; // copy ctor
b = a; // assignment operator
```

T& operator[](size_t idx)


```
T& operator[](size_t idx)
{
     // no bounds checking code here
     return d_arr_p[idx];
}
```

- Fun fact: in the standard for **`std::vector`**, bounds checking is used in the **`.at()`** method but not for the `[]` operator
- Returns an lvalue reference so there doesn't have to be a copy into the return destination

Tips for overloading operators

- Prefer passing objects by `const&` instead of passing by value.
- The standard requires that operators `= () [] ->` and `->` must be members.
- For all other functions:
  - If the function is `operator>>` or `operator<<` for stream I/O, or if it needs type conversions on its leftmost argument, or if it can be implemented using the class's public interface alone, make it a nonmember (and friend if needed in the first two cases)
  - If it needs to behave virtually, add a virtual member function to provide the virtual behavior, and implement it in terms of that

Named Requirements

- These are just a few of the operators/functions that need to be overloaded. C++ containers don't have a rigid class hierarchy like Java, they just need to abide by certain "concepts" (or named requirements). See the specification for the `SequenceContainer` concept here:
- https://en.cppreference.com/w/cpp/named_req/SequenceContainer
- These concepts will be able to be explicitly defined in c++20
  - https://en.cppreference.com/w/cpp/language/constraints
- Although they kind of already can, see `std::enable_if`:
  - https://en.cppreference.com/w/cpp/types/enable_if

Customization Points

- Customization points in c++ are functions/operators/class methods you can implement to allow your class to be used in standard algorithms/containers.
- Operators are nice because they can work on both primitive and class template arguments.
    - However, they don't solve everything — for example, what if you needed to get an iterator to the beginning of a range for both both STL containers and c-style arrays?
- I like to classify customization points into two different types: operators/class methods and standalone methods. While the former is easy to implement as seen above, the latter is considered somewhat of an abomination.

std::swap (1)

- https://en.cppreference.com/w/cpp/algorithm/swap
- In our own library with its own namespace, it's pretty simple to write your own swap function:

```
void swap(Widget& a, Widget& b)
{
        a.swap(b);
}
```

- The problem comes when the library tries to call it. The library has to make an "unqualified" call to swap, invoking a process called ADL, or Argument Dependent Lookup.

```
using std::swap;  // pull `std::swap` into scope
swap(ta, tb);
```

`std::swap` (2)


```
using std::swap;  // pull `std::swap` into scope
swap(ta, tb);
```

- The first line pulls swap into scope, so calling a function named swap would be unqualified
  - It wouldn't be immediately clear which swap the call is referring to, your implementation or the standard implementation.
- ADL resolves this by running through a set of rules basically saying to try to use the user's implementation before the standard one.

`std::hash` (1)

- https://en.cppreference.com/w/cpp/utility/hash
- A different approach is taken in the std::hash function, but it's still pretty shit
- This time, hash actually isn't a function, but an object with an overload for the `()` operator. For those who don't know, this is called a functor
- To write our own implementation, we actually have to add (or inject) to the standard namespace:

std::hash (2)


```
namespace std
{
    template<>
    struct hash<Widget>
    {
        using argument_type = Widget;
        using result_type = std::size_t;
        result_type operator()(const argument_type& w) const noexcept
        {
            return <calc hash>;
        }
    };
}
```

- Then, the library can call the functor like normal:

```
std::hash<Widget>{}(w);
```

Stacks (1)

- We're going to finish this workshop by writing a very simple stack data structure with one method: **`pop()`**
- Stack implements its functionality *in terms of* another data structure, passed in by a template parameter.
  - This gives it the insertion/access properties of a list/vector/deque

```
template<typename T, class Container = my_vector>
class my_stack
{
public:
      <ctor shit>
      T pop();
      <other shit>
private:
      Container data_;
}
```

Stacks (2)

- Actually split into two methods, `top()` and `pop()`

```
T& top()
{
	// It is undefined behavior to call top on an empty stack
	return d_data.back();
}
void pop()
{
	d_data.pop_back();
}
```

- This makes it easier to get the top value without forcing a copy operation

Quick note on speed

- C++ has to cater to a multitude of different uses; as versatile as the STL is it can't cater to all of them.
- While functions like `std::sort` are generally pretty competitive, structures like `unordered_map` are pretty slow compared to some alternatives like Google's maps:
  - https://github.com/sparsehash/sparsehash
- As a result, a lot of companies have their own partial or full implementations of the standard library to make their software faster and to keep things consistent across platforms
  - Also true for open-source projects like OpenCV
- Case study: meshoptimizer
  - https://zeux.io/2019/01/17/is-c-fast/

Thanks for Coming!

Free Burritos Next Week: RSVP @ **acmurl.com/cppevent**