Advanced C++: Modern

By David Hacker and Nicolas Nebel

https://acmurl.com/cpp2

https://github.com/TritonSE/cpp-workshop-2

Check in code: cpp2

sudo apt install cmake (if not already installed)
git clone https://github.com/TritonSE/cpp-workshop-2
cd cpp-workshop-2
mkdir build
cd build
cmake ..
make

History of Modern C++ (1)

- C++11 specification came out in 2011, introducing many long-awaited standard library additions and language features
  - Small changes like auto, nullptr, alias declarations (using), scoped enums, delete/default functions, override, noexcept, constexpr
  - Larger concepts such as smart pointers, lambda expressions, rvalue references, and variadic templates
  - Concurrency api was also added and will be touched on in this workshop, but I would highly recommend reading a more in-depth overview (to be found).

History of Modern C++ (2)

- Language additions continued in the C++14 and C++17 specifications to refine the additions made in c++11 and add some small new features
- C++20 will be approved in February 2020, and scale of changes is on par with C++11
  - Notes from the last meeting of ISO here
  - Includes module system, coroutines, idea of “concepts” for template arguments, and ranges API.
- Networking & reflection system in the works for C++23/26

Smart Pointers

- One of the more notable changes brought by c++11 was the addition of “smart pointers,” or a wrapper class for a pointer that deallocates space referenced by the pointer at the end of the object’s scope.
- This way, you don’t have to manually free memory on the heap once you’re done with it.
- Let’s look at a sample implementation on the next slide

Example Smart Pointer

```
template<typename T>
class my_smart_pointer
{
public:
    my_smart_pointer(T* ptr) : d_ptr(ptr){}

    ~my_smart_ptr()
    {
        delete d_ptr; // fun fact: calling delete on a nullptr does nothing
    }

private:
    T* d_ptr;
};
```

More on Smart Pointers

- Wow that was easy! With the exception of a few utility methods we didn't add, this seems pretty simple
- Except what if you want multiple references to an object, where each reference has a different scope?
    - If one of the references ran out of scope, the rest of the smart pointers would be referencing deallocated memory so dereferencing would result in undefined behavior
- This problem can be solved in two ways, resulting in two different kinds of smart pointers.
    - `std::unique_ptr`
    - `std::shared_ptr`

`unique_ptr` and Move Semantics

- `unique_ptr` solves the problem of multiple pointers with multiple scopes the easy way: only one pointer is active at a time
  - When copying this pointer, the previous `unique_ptr` object is invalidated
  - How is this done?
- Rvalue references (*very* in-depth stack overflow post [here](#))
  - Before explaining this, it is helpful to know what an rvalue is.
  - In simple terms, an lvalue is the term on the left side of an assignment statement, while an rvalue is the term on the right side.

```
lvalue ||      || rvalue
 (a)   \/      \/ (not named)
string a = “hello”;
```

Rvalues Explained

- However, this general definition of an lvalue could be expanded to any object that has an identifiable location in memory. In the following case, one lvalue is being copied to another:

```
string a = “hello”;
string b = a; // lvalue a copied into b
```

- In this case, we have an rvalue being copied into an lvalue:

```
string c = “ there”;
string d(a+c); // a+c is an rvalue, since the temporary object
// it creates cannot be referenced in memory (it has no variable
// name like a,b, or c)
```

- In c++98, a temporary string object would be created and its contents would be copied over to d in the string’s copy constructor. However, this temporary is kind of unnecessary

Rvalue References Explained

- C++11 provided support for rvalue operations with a new reference type and copy/assignment operators.
  - The new reference looks similar to an lvalue reference, with the syntax `<type>&&`
  - Functions the same as an lvalue reference, the only difference is that it only refers to rvalues
- In methods that take rvalue reference parameters, we can simply do a shallow copy and invalidate the temporary since it won't be used anywhere else.

```
string(string&& other) // string&& is an rvalue reference to a string
{
    data = other.data; // shallow ptr copy
    other.data = nullptr; // invalidate other object's ptr
}
```

Quick Note on `std::move`

- Functions that take rvalue reference parameters only work with rvalue inputs.
  - If the language implicitly allowed the use of these, the lvalue object would be implicitly invalidated and casuals would be very frustrated.
  - `unique_ptrs` take advantage of this by only allowing rvalue constructions and assignments which transfer ownership of the allocation to the new pointer and invalidate the old one
- We can explicitly cast an lvalue to an rvalue to invoke an rvalue operation by using the `std::move` function.

```
std::unique_ptr<Object> a(new Object());
std::unique_ptr<Object> b(a); // compile error
std::unique_ptr<Object> c(std::move(a)); // all good
```

Write

```
my_unique_ptr(my_unique_ptr&& other),
my_unique_ptr& operator=(my_unique_ptr&& other)
```

(10 min)

Hint: Look at the implementation for a string move constructor a few slides ago

Move Constructor Implementation

```
my_unique_ptr(my_unique_ptr&& other) : d_data_p(other.d_data_p)
{
     other.d_data_p = nullptr;
}
```

- Similarly to the string move constructor, we initialize our pointer to other’s pointer, then invalidate other’s pointer
- Fun fact: unlike with lvalue references, it’s pretty useless to have a const rvalue reference parameter because you always need to modify the object in order to invalidate it.

Move Assignment Implementation

```
my_unique_ptr& operator=(my_unique_ptr&& other)
{
      d_data_p = other.d_data_p;
      other.d_data_p = nullptr;
      return *this;
}
```

- Similarly to the move constructor earlier, we set our pointer to other’s pointer, then invalidate other’s pointer
- Fun fact: `unique_ptr` is a bit of a special case but in the case of other containers like `std::vector`, both the move and copy assignment operator can be implemented like this if both types of constructors are implemented.

```
vector& operator=(vector other)
{
      swap(other);
}
```

Write

`T* operator->(), T& operator*(),`

`T* release()`

(5 min)

Hint: how could you use std::swap in release()?

`->`, `*`, and `release()` Implementation

```
T* operator->() const
{
    return d_data_p;
}

T& operator*() const
{
    return *d_data_p;
}

T* release()
{
    T* result = d_data_p;
    d_data_p = nullptr;
    return result;
}
```

Notes on `unique_ptr`

- lvalue copy constructor and assignment operator are explicitly deleted using `= delete` because they are initially automatically generated by the compiler.
- The constructor syntax is a little messy so c++14 brought in the `make_unique`/`make_shared` functions to make things easier.
  - https://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique
- There is literally no reason to not use a `unique_ptr`. There is virtually no overhead, performance or space wise, and it instantly makes your code safer.
  - That is, unless you need multiple pointers to the same object...

Introducing `shared_ptr`

- Able to have multiple pointers to the same object, as soon as the last pointer object runs out of scope, the memory referenced is deallocated.
- Each shared pointer has two internal pointers, one pointer to the object and another pointing to a "control block," which holds a shared pointer count, weak pointer count (will cover later), and custom destructor.
  - The destructor of a `shared_ptr` checks the control block to see if it is the last strong reference and frees the memory if it is.

Introducing `weak_ptr`

- Implements a “weak” ptr to an object, which checks if the pointer is dangling.
    - It does not add to ref count and it can’t actually access or dereference ptr, used solely for above purpose.
- Can call `lock()` to get a `shared_ptr` to access object and fields
- Like a shared pointer, it also has a reference to the control block, which stays allocated as long as it’s weak pointer or shared pointer ref count is greater than zero.
- To see if the pointer is dangling, it checks to see if shared pointer ref count is greater than zero.

Drawbacks of Shared Pointers

- They have twice the size of a raw pointer
- Memory for the reference count must be dynamically allocated,
    - There is a small performance overhead in allocating on the heap rather than on the stack directly in the class field.
- Increments and decrements to ref count must be atomic, ie thread safe, causing slower performance
    - Many people don't know about/understand multi-threading in c++ and therefore forget about this, but this is the most important downside.
- Because of these drawbacks, raw pointers are used over shared pointers in performance-heavy applications like data structures.

Custom Destructs/Lambda Expressions

- Smart pointers also have a second optional argument in the constructor: custom destructors. Instead of the delete operator, you can specify another way to destroy the data controlled.
- Function parameters can be specified multiple ways in C++: as regular functions, functors (classes/structs with the () operator overridden), and lambdas.
    - Lambda expressions are an easy way to create inline functors

Lambda Expressions (1)

- Lambda expression syntax:

```
[ <captured vars> ] ( <params> ) { <body> }
auto callable = [&captured] (int param) { return param + captured; }
```

- Prefix captured variables with & to capture them by reference, or just put & to capture all used variables by reference
    - Likewise, name captured variables without & to capture them by value, or just put = to capture all used variables by value
- Only non-static local variables can be captured by value, members are captured by reference even if = is specified
- Be aware of dangling references after the function declaring the lambda exits, memory will be pulled out from underneath you.
- Some suggest to explicitly name the variables you capture in the [] section as to not forget what vars you are using by reference.

Lambda Expressions (2)

- Return type is auto-deduced (as of c++14), but can be specified with `-><type>` after params

```
[&captured_var] (const char* name) -> std::string
{
      std::cout << captured_var << “\n”;
      return std::strcat(“hello ”, name);
      // return of const char* is converted to std::string
}
```

- Init capture syntax allows you to initialize local lambda vars in [] section:

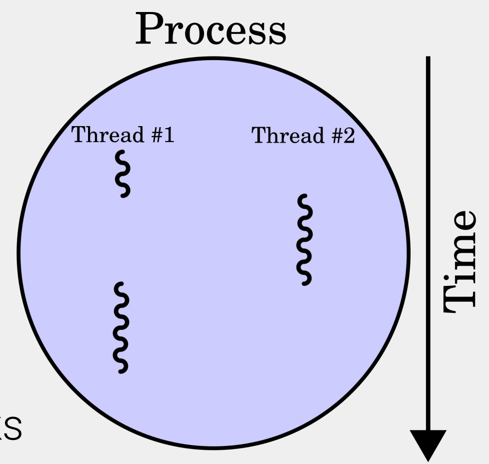
```
auto func = [pw = std::make_unique<Widget>()]
{
      return pw->isValidated() && pw->isArchived();
};
```

Lambda Expressions (3)

- Lambda expressions are a hidden type, but under the hood they are actually implemented as functors, with any captured variables stored in the object's fields.
- Because their type is hidden, they can be tricky to use. The standard library provides an `std::function` object to store a general callable object (ie function, functor, lambda),
  - Just use `auto` though, as `std::function` only has a fixed size for any given signature. If the signature is larger, it will allocate more memory on the heap and will end up using more storage than if just "`auto`" was used.
  - Also, due to some implementation complications, invoking a closure via a `std::function` object is almost certain to be slower than calling it via an auto-declared object.

Basics of Threading

- A thread is a "program within a program"
  - Represents some arbitrary sequence of instructions
  - Can be run concurrently with other threads
- Threads share process memory
  - Threads share global variables, heap, parent thread's stack
  - If thread A has pointer 0xDEAD and thread B has pointer 0xDEAD, then both thread A and B can read and write to the same memory address
- However, threads maintain their own stack.
  - Memory addresses in one stack can be shared across stacks
- Main advantage: can allow work to be parallelized across multiple cores (multithreading)



C++ Threads

- C++ thread object is pretty simple to create
- Takes in a void function with no arguments
  - Can pass in a std::bind<Fn, Args...> STL function
  - Can pass in a lambda expression capturing some values

```
std::string somestr = “test”;
std::thread child([&somestr]() {
    std::cerr << “test” << std::endl;
});
```

- To start thread, you either call `child.detach()` or `child.join()` from the parent thread
  - `join()` will cause the parent thread to block until the child finishes
  - `detach()` will cause child and parent to execute independently

Classic Synchronization Issues (1)

- An arbitrary number of threads can read from a memory location with no side effects
- What happens when one or more threads want to modify that memory location? Can we guarantee that data remains incorrupt?

```
int counter = 5;
std::thread child1([&counter]() {
     counter = counter - 1;
});
std::thread child2([&counter]() {
     counter = counter - 1;
});
child1.join();
child2.join();
```

- Expected output is `counter = 3`. Actual output is `counter = 4`?

Classic Synchronization Issues (2)

- The problem is that both threads can execute reads before one thread has a chance to write
  - child1 reads counter = 5, child2 reads counter = 5, child1 writes counter = 4, child2 writes counter = 4
- We want to ensure that the first thread that arrives reads AND writes counter before the second thread arrives
  - child1 reads counter = 5, child1 writes counter = 4, child2 reads counter = 4, child2 writes counter = 3
- Code that you want a sequential guarantee on is called a **critical section**
  - Prevents race conditions, two or more threads racing to get to a memory location or instruction first

Synchronization Solution: std::mutex

- C++ provides a data structure called a mutex.
    - Stands for mutual exclusion.
    - Also, known as a lock.
- Only one thread can “acquire” and hold on to the mutex at a time
    - Other threads must wait in line for the holding thread to “release” the mutex
- Thread acquires a mutex by calling `lock()`
- Thread releases a mutex by calling `unlock()`
- Race conditions can be avoided by …
    - Creating a shared mutex
    - Identifying <u>minimal</u> critical sections
    - Wrapping critical sections in mutex calls

```
int counter = 5;
std::mutex mtx;
std::thread child1([&counter, &mtx]() {
    mtx.lock();
    counter = counter - 1;
    mtx.unlock();
});
std::thread child2([&counter, &mtx]() {
    mtx.lock();
    counter = counter - 1;
    mtx.unlock();
});
child1.join();
child2.join();
```

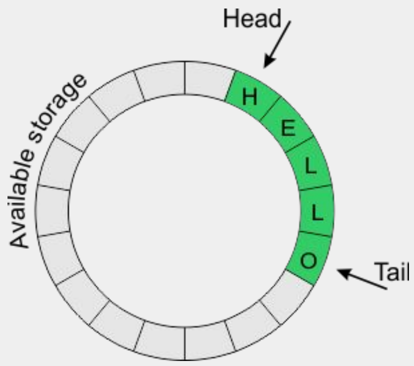
Never Use std::mutex Directly Ever

- This is going to seem contradictory but … never use `std::mutex` directly?
  - What! Didn't you just say `std::mutex` can be used to solve race conditions?
- What happens when a thread holding a mutex throws an exception and terminates?
  - Following threads may want the mutex
  - Deadlock occurs because mutex is never released
- Use `std::lock_guard<std::mutex>`
  - Give it a mutex through its constructor
  - Constructor calls `lock()`
  - Destructor calls `unlock()`

```cpp
int counter = 5;
std::mutex mtx;
std::thread child1([&counter, &mtx]() {
     std::lock_guard<std::mutex> lck(mtx);
     counter = counter - 1;
});
std::thread child2([&counter, &mtx]() {
     std::lock_guard<std::mutex> lck(mtx);
     counter = counter - 1;
});
child1.join();
child2.join();
```

Thread-Safe Ring Buffer

- A ring buffer is a compact data structure that emulates a FIFO queue with a maximal size N
    - Treats a fixed-length array as if the Nth element is the same as the 0th element.
- Has two pointers: a head pointer and a tail pointer
    - To remove an element from the front of the queue, increment the head pointer, wrapping to front of array when necessary
    - To add an element, write the element to the location pointed to by the tail and increment the tail pointer
- This is already written for you in `include/my_ring_buffer.hpp`
- Your job is to make it thread-safe
    - Multiple threads may add elements to the buffer
    - Multiple threads may remove elements from the buffer



Make methods in

`my_ring_buffer<T, N>`

thread-safe

(10 min)

Hint: could a buffer own a mutex?

Final Thoughts on Thread Safety

- Correct solution for `my_ring_buffer` was to have each buffer own a mutex and then just add a guard for every thread safe method
  - This paradigm is pretty common - called a **monitor**
  - Every method in a monitor synchronizes over the same lock
- Performance improvement: try and remove locks
  - Use `std::atomic` for small integer values
  - Use implementations of “lock-free” data structures
  - Most implementations that avoid locks blow up horribly
- Performance improvement: separate data, lock groups individually
  - Java’s ConcurrentHashMap implementation
  - Stock exchange example - locking based on exchange groups
- More threading: semaphores, condition variables (take CSE 120!!!)

Misc. Modern C++ Tips (1)

- Use `nullptr` instead of NULL
  - NULL is literally `#define NULL 0` and it's a relic from C
- Prefer using over typedefs, as you can specify template parameters

```
template<typename T>
using vec3 = std::array<T, 3>;
using vec3f = vec3<float>;
```

- Prefer enum class to just enums

```
enum class ColorSpace { sRGB, HSV, XYZ };
ColorSpace cs = ColorSpace::sRGB; // now you have to access the enum
// type by prefixing it with the enum name (ColorSpace)
```

- Mark functions that support the nothrow guarantee as `noexcept`.
  - Checks if all things the function does are `noexcept`

Misc. Modern C++ Tips (2)

- Use default/delete to insert a default method implementation or delete a default method implementation
  - Can be used to be explicit even if the standard dictates that a default method be created/deleted
- Mark methods that override a parent's virtual functions as override
  - Doesn't really do anything but has the compiler check that it actually overrides something (i.e. the signature is right)
- Use `constexpr`, not `#define`.
  - `constexpr int a=1+2` will compute `a=3` at compile time and use that everywhere, while `#define a 1+2` will just copy+paste 1+2 everywhere
  - Is actually very powerful—can have constexpr objects, recursive functions, if statements, etc

Modern C++ Case Study (1)

- The Ranges API, included as a part of the standard library in the C++20 release, makes extensive use of modern c++ features.
  - https://github.com/ericniebler/range-v3
  - The library provides a framework to easily manipulate ranges (arbitrary iterable sequences like containers).
- The calendar example prints out the calendar for a given year to terminal using just a simple date iterator based off boost's date library
  - https://github.com/ericniebler/range-v3/blob/master/example/calendar.cpp
  - Lines 96-130 define the date range, 132-283 define several utility manipulators and views to help print the calendar, and finally the format_calendar function on 288 uses all the manipulators and views to return a range of strings to print with cout.

Modern C++ Case Study (2)

- However, some notable cppers have objected to this library and some of the new modern additions
  - https://aras-p.info/blog/2018/12/28/Modern-C-Lamentations/
- Objections include:
  - Code using ranges is kinda unreadable (as you may have found)
  - Code using the ranges api has very (150x slower) slow non-optimized build performance
  - Even though the ranges api provides several reusable components, they don't provide too much of an improvement over just using for loops.
  - Compilation times are drastically increased (mostly due to extensive template use). Even if you aren't using new features, library updates added more to even basic headers like <cmath>

Modern C++ Case Study (3)

- Everything in moderation—many companies have integrated modern features where they make sense, and for the most part the features introduced in C++11/14/17 don't have many drawbacks and are implemented reasonably (unless it's Microsoft's stl)
- Some companies have been pretty stubborn, for example Google's C++ style guide forbade usage of modern features up until recently
  - I think it was because they weren't well known to most coders and therefore code using them would be hard to understand
- Epic uses most features but discourages use of auto, mostly for readability reasons

Thanks for Coming!