

COMP6771

Advanced C++ Programming

Week 5.1

Resource Management

In this lecture

Why?

- While we have ignored heap resources (malloc/free) to date, they are a critical part of many libraries and we need to understand best practices around usage.

What?

- new/delete
- copy and move semantics
- destructors
- lvalues and rvalues

Revision: Objects

- What is an object in C++?
 - An object is a region of memory associated with a type
 - Unlike some other languages (Java), basic types such as int and bool are objects
- For the most part, C++ objects are designed to be intuitive to use
- What special things can we do with objects
 - Create
 - Destroy
 - Copy
 - Move

Long lifetimes

- There are 3 ways you can try and make an object in C++ have a lifetime that outlives the scope it was defined in:
 - Returning it out of a function via copy (slow & expensive)
 - Returning it out of a function via references (bad, see slide below)
 - Returning it out of a function as a heap resource (today's lecture)

Long lifetime with references

- We need to be very careful when returning references.
- **The object must always outlive the reference.**
- This is undefined behaviour - if you're unlucky, the code might even work!
- **Moral of the story:** Do not return references to variables local to the function returning.
- For objects we create INSIDE a function, we're going to have to create heap memory and return that.

```
auto okay(int& i) -> int& {  
    return i;  
}  
  
auto okay(int& i) -> int const& {  
    return i;  
}
```

```
auto not_okay(int i) -> int& {  
    return i;  
}  
  
auto not_okay() -> int& {  
    auto i = 0;  
    return i;  
}
```

New and delete

- Objects are either stored on the **stack** or the **heap**
- In general, most times you've been creating objects of a type it has been on the stack
- We can create heap objects via **new** and free them via **delete** just like in C

```
1 #include <iostream>
2 #include <vector>
3
4 int main() {
5     int* a = new int{4};
6     std::vector<int>* b = new std::vector<int>{1,2,3};
7     std::cout << *a << "\n";
8     std::cout << (*b)[0] << "\n";
9     delete a;
10    delete b;
11    return 0;
12 }
```

New and delete

- Why do we need heap resources?
 - Heap object outlives the scope it was created in
 - More useful in contexts where we need more explicit control of memory (e.g. vector as a dynamically sized array)

```
1 #include <iostream>
2 #include <vector>
3
4 int* newInt(int i) {
5     int* a = new int{i};
6     return a;
7 }
8
9 int main() {
10     int* myInt = newInt();
11     std::cout << *a << "\n"; // a was defined in a scope that
12                               // no longer exists
13     delete a;
14     return 0;
15 }
```

std::vector<int> - under the hood

Let's speculate about how a vector is implemented. It's going to have to manage some form of heap memory, so maybe it looks like this? Is anything wrong with this?

```
1 class my_vec {
2     // Constructor
3     my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}
4
5     // Destructor
6     ~my_vec() {}
7
8     int* data_;
9     int size_;
10    int capacity_;
11 }
```


Destructors

- Called when the object goes out of scope
 - What might this be handy for?
 - Does not occur for reference objects
- Implicitly noexcept
 - What would the consequences be if this were not the case
- Why might destructors be handy?
 - Freeing pointers
 - Closing files
 - Unlocking mutexes (from multithreading)
 - Aborting database transactions

std::vector<int> - Destructors

- What happens when vec_short goes out of scope?
 - Destructors are called on each member.
 - Destructing a pointer type does nothing
- As it stands, this will result in a memory leak. How do we fix?

```
1 class my_vec {
2     // Constructor
3     my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}
4
5     // Destructor
6     ~my_vec() {}
7
8     int* data_;
9     int size_;
10    int capacity_;
11 }
```

```
1 my_vec::~~my_vec() {
2     delete[] data_;
3 }
```

Rule of 5

When writing a class, always consider the "rule of 5"

- Destructor
- Copy constructor
- Copy assignment
- Move assignment
- Move constructor

Rule of 5

When writing a class, always consider the "rule of 5"

- Destructor
- Copy constructor
- Copy assignment
- Move assignment
- Move constructor

The presence or absence of these 5 operations are critical in managing resources

std::vector<int> - under the hood

- Though you should always consider it, you should rarely have to write it
 - If all data members have one of these defined, then the class should automatically define this for you
 - But this may not always be what you want
 - C++ follows the principle of "only pay for what you use"
 - Zeroing out the data for an int is extra work
 - Hence, moving an int actually just copies it
 - Same for other basic types

```
1 class my_vec {
2     // Constructor
3     my_vec(int size): data_{new int[size]}, size_{size}, capacity_{siz
4
5     // Copy constructor
6     my_vec(my_vec const&) = default;
7     // Copy assignment
8     my_vec& operator=(my_vec const&) = default;
9
10    // Move constructor
11    my_vec(my_vec&&) noexcept = default;
12    // Move assignment
13    my_vec& operator=(my_vec&&) noexcept = default;
14
15    // Destructor
16    ~my_vec() = default;
17
18    int* data_;
19    int size_;
20    int capacity_;
21 }
```

```
1 // Call constructor.
2 auto vec_short = my_vec(2);
3 auto vec_long = my_vec(9);
4 // Doesn't do anything
5 auto& vec_ref = vec_long;
6 // Calls copy constructor.
7 auto vec_short2 = vec_short;
8 // Calls copy assignment.
9 vec_short2 = vec_long;
10 // Calls move constructor.
11 auto vec_long2 = std::move(vec_long);
12 // Calls move assignment
13 vec_long2 = std::move(vec_short);
```

std::vector<int> - Copy constructor

- What does it mean to copy a my_vec?
- What does the default synthesized copy constructor do?
 - It does a memberwise copy
- What are the consequences?
 - Any modification to vec_short will also change vec_short2
 - We will perform a double free
- How can we fix this?

```
1 class my_vec {
2     // Constructor
3     my_vec(int size):
4         data_{new int[size]},
5         size_{size},
6         capacity_{size} {}
7
8     // Copy constructor
9     my_vec(my_vec const&) = default;
10    // Copy assignment
11    my_vec& operator=(my_vec const&) = default;
12
13    // Move constructor
14    my_vec(my_vec&&) noexcept = default;
15    // Move assignment
16    my_vec& operator=(my_vec&&) noexcept = default;
17
18    // Destructor
19    ~my_vec() = default;
20
21    int* data_;
22    int size_;
23    int capacity_;
24 }
```

```
1 my_vec::my_vec(my_vec const& orig): data_{new int[orig.size_]},
2                                     size_{orig.size_},
3                                     capacity_{orig.size_} {
4     std::copy(orig.data_, orig.data_ + orig.size_, data_);
5 }
```

```
1 auto vec_short = my_vec(2);
2 auto vec_short2 = vec_short;
```

std::vector<int> - Copy assignment

- Assignment is the same as construction, except that there is already a constructed object in your destination
- You need to clean up the destination first
- The copy-and-swap idiom makes this trivial

```
1 my_vec& my_vec::operator=(my_vec const& orig) {
2     return my_vec(orig).swap(*this);
3 }
4
5 my_vec& my_vec::swap(my_vec& other) {
6     std::swap(data_, other.data_);
7     std::swap(size_, other.size_);
8     std::swap(capacity_, other.capacity_);
9 }
10
11 // Alternate implementation, may not be as performant.
12 my_vec& my_vec::operator=(my_vec const& orig) {
13     my_vec copy = orig;
14     std::swap(copy, *this);
15     return *this;
16 }
```

```
1 auto vec_short = my_vec(2);
2 auto vec_long = my_vec(9);
3 vec_long = vec_short;
```

```
1 class my_vec {
2     // Constructor
3     my_vec(int size):
4         data_{new int[size]},
5         size_{size},
6         capacity_{size} {}
7
8     // Copy constructor
9     my_vec(my_vec const&) = default;
10    // Copy assignment
11    my_vec& operator=(my_vec const&) = default;
12
13    // Move constructor
14    my_vec(my_vec&&) noexcept = default;
15    // Move assignment
16    my_vec& operator=(my_vec&&) noexcept = default;
17
18    // Destructor
19    ~my_vec() = default;
20
21    int* data_;
22    int size_;
23    int capacity_;
24 }
```

lvalue vs rvalue

- **lvalue**: An expression that is an object reference
 - E.G. Variable name, subscript reference
 - Always has a defined address in memory
- **rvalue**: Expression that is not an lvalue
 - E.G. Object literals, return results of functions
 - Generally has no storage associated with it

```
1 int main() {  
2     int i = 5; // 5 is rvalue, i is lvalue  
3     int j = i; // j is lvalue, i is lvalue  
4     int k = 4 + i; // 4 + i produces rvalue  
5                     // then stored in lvalue k  
6 }
```


Lvalue references

- There are multiple types of references
 - Lvalue references look like T&
 - Lvalue references to const look like T& const
- Once the lvalue reference goes out of scope, it may still be needed

Lvalue references

```
1 void f(my_vec& x);
```

- There are multiple types of references
 - Lvalue references look like T&
 - Lvalue references to const look like T& const
- Once the lvalue reference goes out of scope, it may still be needed

rvalue references

```
1 void f(my_vec&& x);
```

- Rvalue references look like T&&
- An rvalue reference formal parameter means that the value was disposable from the caller of the function
 - If outer modified value, who would notice / care?
 - The caller (main) has promised that it won't be used anymore
 - If inner modified value, who would notice / care?
 - The caller (outer) has never made such a promise.
 - An rvalue reference parameter is an lvalue inside the function

```
1 void inner(std::string&& value) {
2     value[0] = 'H';
3     std::cout << value << '\n';
4 }
5
6 void outer(std::string&& value) {
7     inner(value); // This fails? Why?
8     std::cout << value << '\n';
9 }
10
11 int main() {
12     f1("hello"); // This works fine.
13     auto s = std::string("hello");
14     f2(s); // This fails because i is an lvalue.
15 }
```

std::move

```
1 // Looks something like this.
2 T&& move(T& value) {
3     return static_cast<T&&>(value);
4 }
```

- A library function that converts an lvalue to an rvalue so that a "move constructor" (similar to copy constructor) can use it.
 - This says "I don't care about this anymore"
 - All this does is allow the compiler to use rvalue reference overloads

```
1 void inner(std::string&& value) {
2     value[0] = 'H';
3     std::cout << value << '\n';
4 }
5
6 void outer(std::string&& value) {
7     inner(std::move(value));
8     // Value is now in a valid but unspecified state.
9     // Although this isn't a compiler error, this is bad code.
10    // Don't access variables that were moved from, except to reconstruct them.
11    std::cout << value << '\n';
12 }
13
14 int main() {
15     f1("hello"); // This works fine.
16     auto s = std::string("hello");
17     f2(s); // This fails because s is an lvalue.
18 }
```

Moving objects

- Always declare your moves as noexcept
 - Failing to do so can make your code slower
 - Consider: push_back in a vector
- Unless otherwise specified, objects that have been moved from are in a valid but unspecified state
- Moving is an optimisation on copying
 - The only difference is that when moving, the moved-from object is mutable
 - Not all types can take advantage of this
 - If moving an int, mutating the moved-from int is extra work
 - If moving a vector, mutating the moved-from vector potentially saves a lot of work
- Moved from objects must be placed in a valid state
 - Moved-from containers **usually** contain the default-constructed value
 - Moved-from types that are cheap to copy are **usually** unmodified
 - Although this is the only requirement, individual types may add their own constraints
- Compiler-generated move constructor / assignment performs memberwise moves

std::vector<int> - Move constructor

Very similar to copy constructor, except we can use std::exchange instead.

```
1 my_vec::my_vec(my_vec&& orig) noexcept
2 : data_{std::exchange(orig.data_, nullptr)}
3 , size_{std::exchange(orig.size_, 0)}
4 , capacity_{std::exchange(orig.capacity_, 0)} {}
```

```
1 class my_vec {
2     // Constructor
3     my_vec(int size)
4     : data_{new int[size]}
5     , size_{size}
6     , capacity_{size} {}
7
8     // Copy constructor
9     my_vec(my_vec const&) = default;
10    // Copy assignment
11    my_vec& operator=(my_vec const&) = default;
12
13    // Move constructor
14    my_vec(my_vec&&) noexcept = default;
15    // Move assignment
16    my_vec& operator=(my_vec&&) noexcept = default;
17
18    // Destructor
19    ~my_vec() = default;
20
21    int* data_;
22    int size_;
23    int capacity_;
24 }
```

```
1 auto vec_short = my_vec(2);
2 auto vec_short2 = std::move(vec_short);
```

std::vector<int> - Move assignment

Like the move constructor, but the destination is already constructed

```
1 my_vec& my_vec::operator=(my_vec&& orig) noexcept {
2     // The easiest way to write a move assignment is generally to do
3     // memberwise swaps, then clean up the orig object.
4     // Doing so may mean some redundant code, but it means you don't
5     // need to deal with mixed state between objects.
6     std::swap(data_, orig.data_);
7     std::swap(size_, orig.size_);
8     std::swap(capacity_, orig.capacity_);
9
10    // The following line may or may not be necessary, depending on
11    // if you decide to add additional constraints to your moved-from
12    // object.
13    delete[] data_;
14    data_ = nullptr;
15    size_ = 0;
16    capacity = 0;
17    return *this;
18 }
```

```
1 class my_vec {
2     // Constructor
3     my_vec(int size): data_{new int[size]}, size_{size}, ca
4
5     // Copy constructor
6     my_vec(my_vec const&) = default;
7     // Copy assignment
8     my_vec& operator=(my_vec const&) = default;
9
10    // Move constructor
11    my_vec(my_vec&&) noexcept = default;
12    // Move assignment
13    my_vec& operator=(my_vec&&) noexcept = default;
14
15    // Destructor
16    ~my_vec() = default;
17
18    int* data_;
19    int size_;
20    int capacity_;
21 }
```

```
1 auto vec_short = my_vec(2);
2 auto vec_long = my_vec(9);
3 vec_long = std::move(vec_short);
```

Passing references to be copied

```
1 struct S {  
2     // modernize-pass-by-value error here  
3     S(std::string const& x) : x_{x} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
- When we construct "b"

Passing references to be copied

```
1 struct S {  
2     // modernize-pass-by-value error here  
3     S(std::string const& x) : x_{x} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a const reference
- When we construct "b"

Passing references to be copied

```
1 struct S {  
2     // modernize-pass-by-value error here  
3     S(std::string const& x) : x_{x} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a const reference
 - We copy it into x_
- When we construct "b"

Passing references to be copied

```
1 struct S {  
2     // modernize-pass-by-value error here  
3     S(std::string const& x) : x_{x} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a const reference
 - We copy it into x_
- When we construct "b"
 - We create a const reference

Passing references to be copied

```
1 struct S {  
2     // modernize-pass-by-value error here  
3     S(std::string const& x) : x_{x} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a const reference
 - We copy it into x_
- When we construct "b"
 - We create a const reference
 - Since we have a **const** reference, we cannot move from it

Passing references to be copied

```
1 struct S {  
2     // modernize-pass-by-value error here  
3     S(std::string const& x) : x_{x} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a const reference
 - We copy it into x_
- When we construct "b"
 - We create a const reference
 - Since we have a **const** reference, we cannot move from it
 - We copy it into x_

Passing references to be copied

Now consider the following

```
1 struct S {  
2     // modernize-pass-by-value error no longer here  
3     S(std::string x) : x_{std::move(x)} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
- When we construct "b"

Passing references to be copied

Now consider the following

```
1 struct S {  
2     // modernize-pass-by-value error no longer here  
3     S(std::string x) : x_{std::move(x)} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a temporary object by copying
- When we construct "b"

Passing references to be copied

Now consider the following

```
1 struct S {  
2     // modernize-pass-by-value error no longer here  
3     S(std::string x) : x_{std::move(x)} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a temporary object by copying
 - We then move that temporary copy
- When we construct "b"

Passing references to be copied

Now consider the following

```
1 struct S {  
2     // modernize-pass-by-value error no longer here  
3     S(std::string x) : x_{std::move(x)} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a temporary object by copying
 - We then move that temporary copy
- When we construct "b"
 - We create a temporary object by moving (since the argument is an rvalue)

Passing references to be copied

Now consider the following

```
1 struct S {  
2     // modernize-pass-by-value error no longer here  
3     S(std::string x) : x_{std::move(x)} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a temporary object by copying
 - We then move that temporary copy
- When we construct "b"
 - We create a temporary object by moving (since the argument is an rvalue)
 - We then move that temporary copy

Passing references to be copied

Now consider the following

```
1 struct S {  
2     // modernize-pass-by-value error no longer here  
3     S(std::string x) : x_{std::move(x)} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a temporary object by copying
 - We then move that temporary copy
- When we construct "b"
 - We create a temporary object by moving (since the argument is an rvalue)
 - We then move that temporary copy
- It turns out that moving from temporary objects is something the compiler can pretty trivially optimize out

Passing references to be copied

```
1 struct S {  
2     // modernize-pass-by-value error no longer here  
3     S(std::string x) : x_{std::move(x)} {}  
4     std::string x_;  
5 };  
6  
7 auto str = std::string("hello world");  
8 auto a = S(str);  
9 auto b = S(std::move(str));
```

- When we construct "a"
 - We create a temporary object by copying
 - We then move that temporary copy
- When we construct "b"
 - We create a temporary object by moving (since the argument is an rvalue)
 - We then move that temporary copy
- It turns out that moving from temporary objects is something the compiler can pretty trivially optimize out

Explicitly deleted copies and moves

- We may not want a type to be copyable / moveable
- If so, we can declare `fn() = delete`

```
1 class T {  
2     T(const T&) = delete;  
3     T(T&&) = delete;  
4     T& operator=(const T&) = delete;  
5     T& operator=(T&&) = delete;  
6 };
```

Implicitly deleted copies and moves

- Under certain conditions, the compiler will not generate copies and moves
- The implicitly defined copy constructor calls the copy constructor member-wise
 - If one of its members doesn't have a copy constructor, the compiler can't generate one for you
 - Same applies for copy assignment, move constructor, and move assignment
- Under certain conditions, the compiler will not automatically generate copy / move assignment / constructors
 - eg. If you have manually defined a destructor, the copy constructor isn't generated
- If you define one of the rule of five, you should explicitly delete, default, or define all five
 - If the default behaviour isn't sufficient for one of them, it likely isn't sufficient for others
 - Explicitly doing this tells the reader of your code that you have carefully considered this
 - This also means you don't need to remember all of the rules about "if I write X, then is Y generated"

RAI (Resource Acquisition Is Initialization)

In summary, today is really about emphasising RAI

- Resource = heap object
- A concept where we encapsulate resources inside objects
 - Acquire the resource in the constructor
 - Release the resource in the destructor
 - eg. Memory, locks, files
- Every resource should be owned by either:
 - Another resource (eg. smart pointer, data member)
 - Named resource on the stack
 - A nameless temporary variable

Object lifetimes

To create safe object lifetimes in C++, we always attach the lifetime of one object to that of something else

- Named objects:
 - A variable in a function is tied to its scope
 - A data member is tied to the lifetime of the class instance
 - An element in a `std::vector` is tied to the lifetime of the vector
- Unnamed objects:
 - A heap object should be tied to the lifetime of whatever object created it
 - Examples of bad programming practice
 - An **owning raw pointer** is tied to nothing
 - A **C-style array** is tied to nothing
- **Strongly recommend** watching the first 44 minutes of Herb Sutter's cppcon talk "Leak freedom in C++... By Default"

Feedback

