COMP6771
Advanced C++ Programming

Week 5.1
Resource Management

Revision: Objects

- What is an object in C++?

Revision: Objects

- What is an object in C++?
  - An object is a region of memory associated with a type

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

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

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

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

std::vector<int> - under the hood

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

std::vector<int> - under the hood

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

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

std::vector<int> - under the hood

- When writing a class, always consider the "rule of 5"
  - Copy constructor
  - Destructor
  - Move assignment
  - Move constructor
  - Copy assignment

```
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

std::vector<int> - under the hood

- When writing a class, always consider the "rule of 5"
  - Copy constructor
  - Destructor
  - Move assignment
  - Move constructor
  - Copy assignment
- Though you should always consider it, you should rarely have to write it

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

std::vector<int> - under the hood

- When writing a class, always consider the "rule of 5"
  - Copy constructor
  - Destructor
  - Move assignment
  - Move constructor
  - Copy assignment
- 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"

```
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

std::vector<int> - under the hood

- When writing a class, always consider the "rule of 5"
  - Copy constructor
  - Destructor
  - Move assignment
  - Move constructor
  - Copy assignment
- 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

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

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?

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?

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

std::vector<int> - Destructors

- What happens when vec_short goes out of scope?
  - Destructors are called on each member.

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

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

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

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
  - We have a memory leak

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

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
  - We have a memory leak
- How do we solve this?

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

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
  - We have a memory leak
- How do we solve this?

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
my_vec::~my_vec() {
  delete[] data_;
}
```

```cpp
// Call constructor.
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
// Doesn't do anything
auto& vec_ref = vec_long;
// Calls copy constructor.
auto vec_short2 = vec_short;
// Calls copy assignment.
vec_short2 = vec_long;
// Calls move constructor.
auto vec_long2 = std::move(vec_long);
// Calls move assignment
vec_long2 = std::move(vec_short);
```

std::vector<int> - Copy constructor

- What does it mean to copy a my_vec?

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

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

std::vector<int> - Copy constructor

- What does it mean to copy a my_vec?
- What does the default synthesized copy constructor do?

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

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

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
auto vec_short = my_vec(2);
auto vec_short2 = 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?

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
auto vec_short = my_vec(2);
auto vec_short2 = 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?

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
auto vec_short = my_vec(2);
auto vec_short2 = 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?

```cpp
my_vec::my_vec(my_vec const& orig): data_{new int[orig.size_]},
                                    size_{orig.size_},
                                    capacity_{orig.size_} {
  // Should work if we also define .begin() and .end(), an exercise for the reader.
  ranges::copy(orig, data_);
}
```

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
auto vec_short = my_vec(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

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
vec_long = 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

```
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

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

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

```cpp
auto vec_short = my_vec(2);
auto vec_long = my_vec(9);
vec_long = 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

```
my_vec& my_vec::operator=(my_vec const& orig) {
  return my_vec(orig).swap(*this);
}

my_vec& my_vec::swap(my_vec& other) {
  std::swap(data_, other.data_);
  std::swap(size_, other.size_);
  std::swap(capacity_, other.capacity_);
}

// Alternate implementation, may not be as performant.
my_vec& my_vec::operator=(my_vec const& orig) {
  my_vec copy = orig;
  std::swap(copy, *this);
  return *this;
}
```

```
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

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

Lvalue references

- There are multiple types of references

Lvalue references

- There are multiple types of references
  - Lvalue references look like T&
  - Lvalue references to const look like T& const

Lvalue references

- There are multiple types of references
  - Lvalue references look like T&
  - Lvalue references to const look like T& const
- A lvalue reference denotes an object whose resource cannot be reused
  - Most objects (eg. variable, variable[0])

Lvalue references

- There are multiple types of references
  - Lvalue references look like T&
  - Lvalue references to const look like T& const
- A lvalue reference denotes an object whose resource cannot be reused
  - Most objects (eg. variable, variable[0])
- Once the lvalue reference goes out of scope, it may still be needed

Rvalue references

- Rvalue references look like T&&

Rvalue references

- Rvalue references look like T&&
- An rvalue denotes an object whose resources can be reused

Rvalue references

- Rvalue references look like T&&
- An rvalue denotes an object whose resources can be reused
  - eg. Temporaries (my_vec object in f(my_vec()))

Rvalue references

- Rvalue references look like T&&
- An rvalue denotes an object whose resources can be reused
  - eg. Temporaries (my_vec object in f(my_vec()))
  - When someone passes it to you, they don't care about it once you're done with it

Rvalue references

- Rvalue references look like T&&
- An rvalue denotes an object whose resources can be reused
  - eg. Temporaries (my_vec object in f(my_vec()))
  - When someone passes it to you, they don't care about it once you're done with it

- “The object that x binds to is YOURS. Do whatever you like with it, no one will care anyway”

Rvalue references

- Rvalue references look like T&&
- An rvalue denotes an object whose resources can be reused
  - eg. Temporaries (my_vec object in f(my_vec()))
  - When someone passes it to you, they don't care about it once you're done with it

- "The object that x binds to is YOURS. Do whatever you like with it, no one will care anyway"
- Like giving a copy to f… but without making a copy

Rvalue references

- Rvalue references look like T&&
- An rvalue denotes an object whose resources can be reused
  - eg. Temporaries (my_vec object in f(my_vec()))
  - When someone passes it to you, they don't care about it once you're done with it

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

- “The object that x binds to is YOURS. Do whatever you like with it, no one will care anyway”
- Like giving a copy to f… but without making a copy

Rvalue references

```cpp
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

Rvalue references

```cpp
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

Rvalue references

```
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

- An rvalue reference formal parameter means that the value was disposable from the caller of the function

Rvalue references

```cpp
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

- 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?

Rvalue references

```cpp
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

- 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

Rvalue references

```cpp
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

- 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?

Rvalue references

```cpp
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

- 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.

Rvalue references

```cpp
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

- 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

std::move

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

std::move

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

- Simply converts it to an rvalue
  - This says "I don't care about this anymore"
  - All this does is allow the compiler to use rvalue reference overloads

std::move

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

- Simply converts it to an rvalue
  - This says "I don't care about this anymore"
  - All this does is allow the compiler to use rvalue reference overloads

```cpp
void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string&& value) {
  inner(std::move(value));
  // Value is now in a valid but unspecified state.
  // Although this isn't a compiler error, this is bad code.
  // Don't access variables that were moved from, except to reconstruct them.
  std::cout << value << '\n';
}

int main() {
  f1("hello"); // This works fine.
  auto s = std::string("hello");
  f2(s); // This fails because i is an lvalue.
}
```

Moving objects

- Always declare your moves as noexcept

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

Moving objects

- Always declare your moves as noexcept
    - Failing to do so can make your code slower

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

Moving objects

- Always declare your moves as noexcept
  - Failing to do so can make your code slower
  - Consider: push_back in a vector

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```cpp
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```cpp
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```cpp
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

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

```
class T {
  T(T&&) noexcept;
  T& operator=(T&&) noexcept;
};
```

std::vector<int> - Move constructor

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

```
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

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

std::vector<int> - Move assignment

- Like the move constructor, but the destination is already constructed

```cpp
my_vec& my_vec::operator=(my_vec&& orig) noexcept {
  // The easiest way to write a move assignment is generally to do
  // memberwise swaps, then clean up the orig object.
  // Doing so may mean some redundant code, but it means you don't
  // need to deal with mixed state between objects.
  ranges::swap(data_, orig.data_);
  ranges::swap(size_, orig.size_);
  ranges::swap(capacity_, orig.capacity_);

  // The following line may or may not be nessecary, depending on
  // if you decide to add additional constraints to your moved-from object.
  orig.clear();
  return *this;
}

void my_vec::clear() noexcept {
  delete[] data_
  data_ = nullptr;
  size_ = 0;
  capacity = 0;
}
```

```cpp
class my_vec {
  // Constructor
  my_vec(int size): data_{new int[size]}, size_{size}, capacity_{size} {}

  // Copy constructor
  my_vec(my_vec const&) = default;
  // Copy assignment
  my_vec& operator=(my_vec const&) = default;

  // Move constructor
  my_vec(my_vec&&) noexcept = default;
  // Move assignment
  my_vec& operator=(my_vec&&) noexcept = default;

  // Destructor
  ~my_vec() = default;

  int* data_;
  int size_;
  int capacity_;
}
```

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

Passing references to be copied

Consider the following code

```
struct S {
  // modernize-pass-by-value error here
  S(std::string const& x) : x_{x} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
auto b = S(std::move(str));
```

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

Passing references to be copied

Consider the following code

```
struct S {
  // modernize-pass-by-value error here
  S(std::string const& x) : x_{x} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
auto b = S(std::move(str));
```

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

Passing references to be copied

Consider the following code

```
struct S {
  // modernize-pass-by-value error here
  S(std::string const& x) : x_{x} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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

Consider the following code

```
struct S {
  // modernize-pass-by-value error here
  S(std::string const& x) : x_{x} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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

Consider the following code

```
struct S {
  // modernize-pass-by-value error here
  S(std::string const& x) : x_{x} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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

Consider the following code

```
struct S {
  // modernize-pass-by-value error here
  S(std::string const& x) : x_{x} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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

```cpp
struct S {
  // modernize-pass-by-value error no longer here
  S(std::string x) : x_{std::move(x)} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
auto b = S(std::move(str));
```

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

Passing references to be copied

Now consider the following

```
struct S {
  // modernize-pass-by-value error no longer here
  S(std::string x) : x_{std::move(x)} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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

```cpp
struct S {
  // modernize-pass-by-value error no longer here
  S(std::string x) : x_{std::move(x)} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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

```
struct S {
  // modernize-pass-by-value error no longer here
  S(std::string x) : x_{std::move(x)} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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

```cpp
struct S {
  // modernize-pass-by-value error no longer here
  S(std::string x) : x_{std::move(x)} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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

```cpp
struct S {
  // modernize-pass-by-value error no longer here
  S(std::string x) : x_{std::move(x)} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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 optimise out

Passing references to be copied

Now consider the following

```cpp
struct S {
  // modernize-pass-by-value error no longer here
  S(std::string x) : x_{std::move(x)} {}
  std::string x_;
};

auto str = std::string("hello world");
auto a = S(str);
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 optimise out
- This should be the same performance for lvalues, but allow moving instead of copying for rvalues

Explicitly deleted copies and moves

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

```cpp
class T {
  T(const T&) = delete;
  T(T&&) = delete;
  T& operator=(const T&) = delete;
  T& operator=(T&&) = delete;
};
```

Implicitly deleted copies and moves

- Under certain conditions, the compiler will not generate copies and moves

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 memberwise

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 memberwise
    - 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

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 memberwise
  - 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

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 memberwise
  - 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 explictly 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"

RAII (Resource Acquisition Is Initialization)

- A concept where we encapsulate resources inside objects

RAII (Resource Acquisition Is Initialization)

- A concept where we encapsulate resources inside objects
  - Acquire the resource in the constructor

RAII (Resource Acquisition Is Initialization)

- A concept where we encapsulate resources inside objects
  - Acquire the resource in the constructor
  - Release the resource in the destructor

RAII (Resource Acquisition Is Initialization)

- A concept where we encapsulate resources inside objects
  - Acquire the resource in the constructor
  - Release the resource in the destructor
  - eg. Memory, locks, files

RAII (Resource Acquisition Is Initialization)

- 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)
  - 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

Object lifetimes

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

- Named objects:

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

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

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 <u>variable</u> in a <u>function</u> is tied to its scope
  - A <u>data member</u> is tied to the lifetime of the <u>class instance</u>
  - An <u>element in a std::vector</u> is tied to the lifetime of the vector

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 <u>variable</u> in a <u>function</u> is tied to its scope
  - A <u>data member</u> is tied to the lifetime of the <u>class instance</u>
  - An <u>element in a std::vector</u> is tied to the lifetime of the vector
- Unnamed objects:

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

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

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"

Object lifetime with references

- We need to be very careful when returning references.

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

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

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

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

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

Object lifetime with references

- We need to be very careful when returning references.
- **The object must always outlive the reference.**

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

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

```cpp
auto questionable(int const& x) -> int const& {
  return i;
}

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

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

Object 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!

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

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

```cpp
auto questionable(int const& x) -> int const& {
  return i;
}

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

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

Object 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

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

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

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

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

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