CMPT 373
Software Development Methods

A Crash Course in (Some of) Modern C++

Nick Sumner
wsumner@sfu.ca

With material from Bjarne Stroustrup & Herb Sutter

C++ *was* complicated/intimidating

- Pointers
  - Arithmetic & indexing
  - dangling
  - when to **`new`** and **`delete`**

C++ *was* complicated/intimidating

- Pointers
  - Arithmetic & indexing
  - dangling
  - when to `new` and `delete`
- Nontrivial types
  - inheritance
  - long names & scoping (iterators)
  - templates

C++ *was* complicated/intimidating

- Pointers
  - Arithmetic & indexing
  - dangling
  - when to `new` and `delete`
- Nontrivial types
  - inheritance
  - long names & scoping (iterators)
  - templates
- Many proposed rules (of varying validity)
  - Rule of 3
  - Don't pass/return objects to/from functions by value
  - …

Modern C++

- Significant effort has gone into revising C++ since C++03
  - Identifying & simplifying unnecessary complexity
  - Adopting features that help reduce complexity in large scale projects.

Modern C++

- Significant effort has gone into revising C++ since C++03
  - Identifying & simplifying unnecessary complexity
  - Adopting features that help reduce complexity in large scale projects.
- Safety
  - types, bounds, lifetimes

Modern C++

- Significant effort has gone into revising C++ since C++03
  - Identifying & simplifying unnecessary complexity
  - Adopting features that help reduce complexity in large scale projects.
- Safety
  - types, bounds, lifetimes
- Syntactic sugar (with safety benefits)

Modern C++

- Significant effort has gone into revising C++ since C++03
  - Identifying & simplifying unnecessary complexity
  - Adopting features that help reduce complexity in large scale projects.
- Safety
  - types, bounds, lifetimes
- Syntactic sugar (with safety benefits)
- Now developed under a lightweight process with new revisions every ~3 years.

Modern C++

- Significant effort has gone into revising C++ since C++03
  - Identifying & simplifying unnecessary complexity
  - Adopting features that help reduce complexity in large scale projects.
- Safety
  - types, bounds, lifetimes
- Syntactic sugar (with safety benefits)
- Now de… …ions
  every ~…

To get you (re)acquainted,
we will explore *some* of modern C++ for now.

I will assume familiarity with older C++,
constructors, destructors, etc.

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

```
Widget w{0, "fritter"};
```

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

```
Widget w{0, "fritter"};
```

Brace initialization was new in C++11

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

```
Widget w{0, "fritter"};
```

Where does **`w`** live in memory?
Is that good/bad?

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

```
Widget w{0, "fritter"};
```

  - Automatic variables/management should be the default.

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

  ```
  Widget w{0, "fritter"};
  ```

  - Automatic variables/management should be the default.
- What about creating one on the heap?

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

```
Widget w{0, "fritter"};
```

  - Automatic variables/management should be the default.

- What about creating one on the heap?

Old:

```
Widget* w = new Widget{0, "fritter"};
```

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

```
Widget w{0, "fritter"};
```

  - Automatic variables/management should be the default.

- What about creating one on the heap?

Old:

```
Widget* w = new Widget{0, "fritter"};
```

What problems does this create?

Managing Object Lifetimes

Suppose I have a **`Widget`** class constructed from an **`int`** and a string.

- How might I create one?

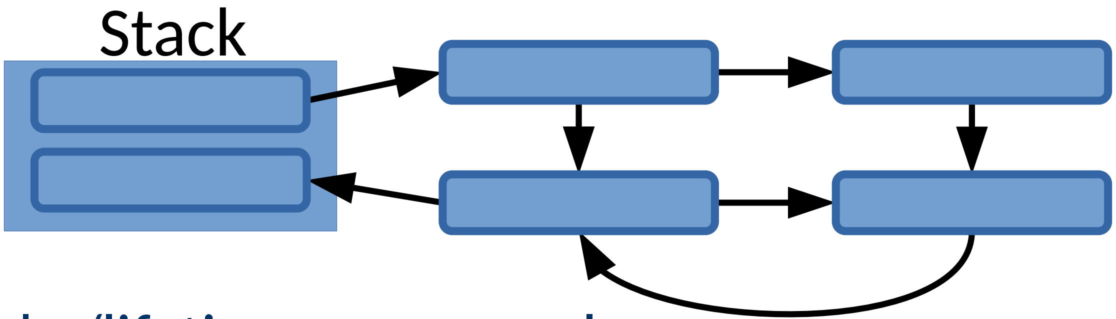
  ```
  Widget w{0, "fritter"};
  ```

  - Automatic variables/management should be the default.
- What about creating one on the heap?

Old:
```
Widget* w = new Widget{0, "fritter"};
```

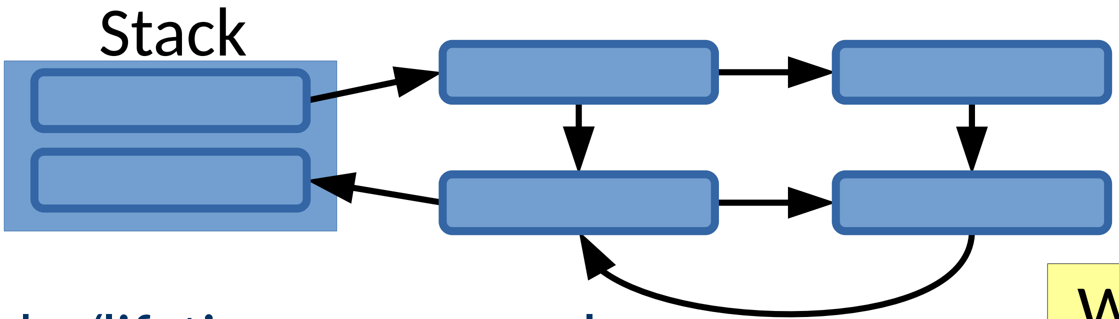
  - Need to delete everything.
  - Need to delete everything only once.
  - Complex object graphs make this harder

Managing Object Lifetimes



Object graphs/lifetimes are complex

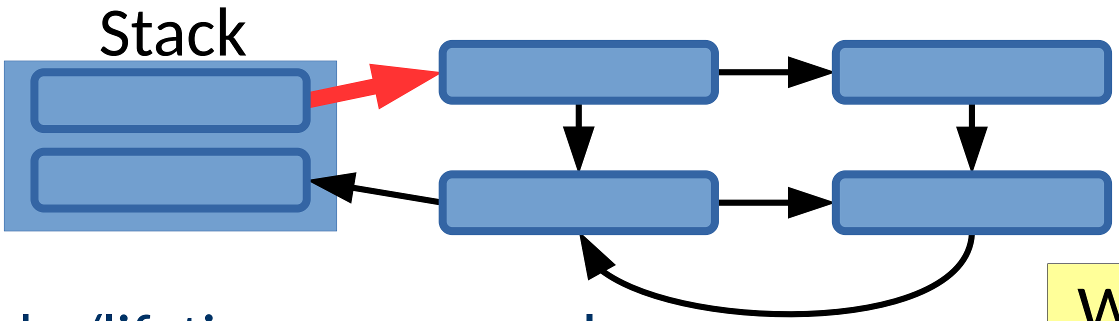
Managing Object Lifetimes



Object graphs/lifetimes are complex

Which pointers can I `delete` & when?

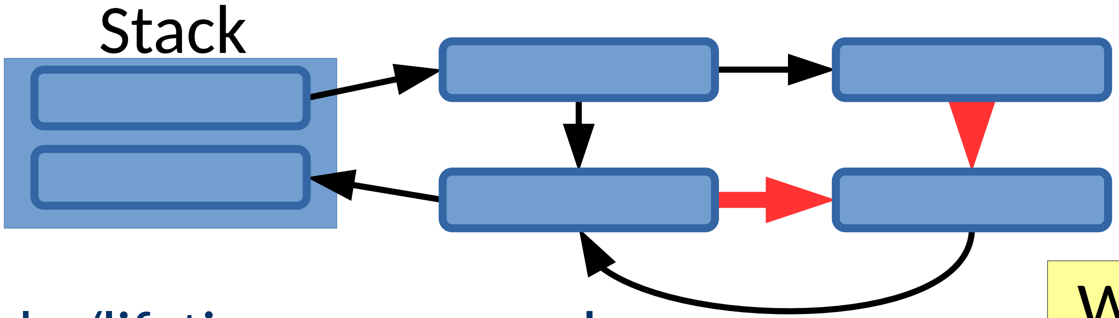
Managing Object Lifetimes



Object graphs/lifetimes are complex

Which pointers can I **`delete`** & when?

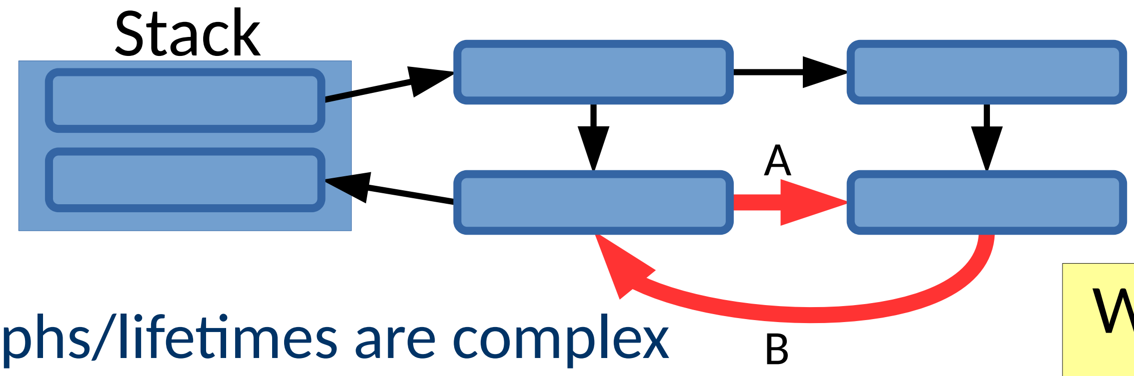
Managing Object Lifetimes



Object graphs/lifetimes are complex

Which pointers can I `delete` & when?

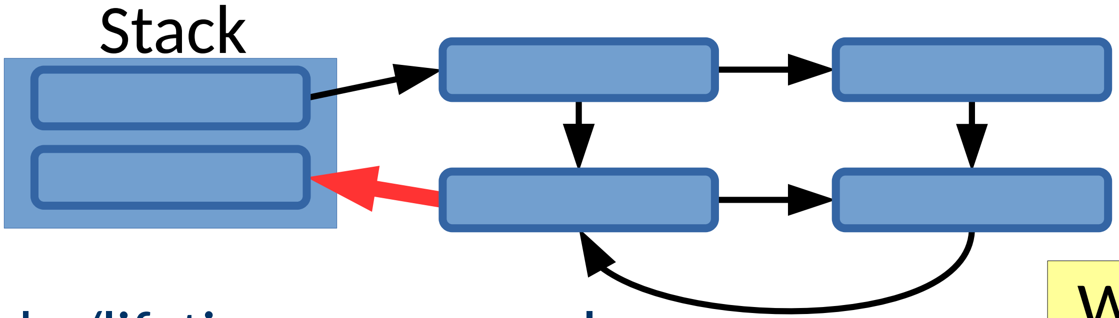
Managing Object Lifetimes



Object graphs/lifetimes are complex

Which pointers can I **`delete`** & when?

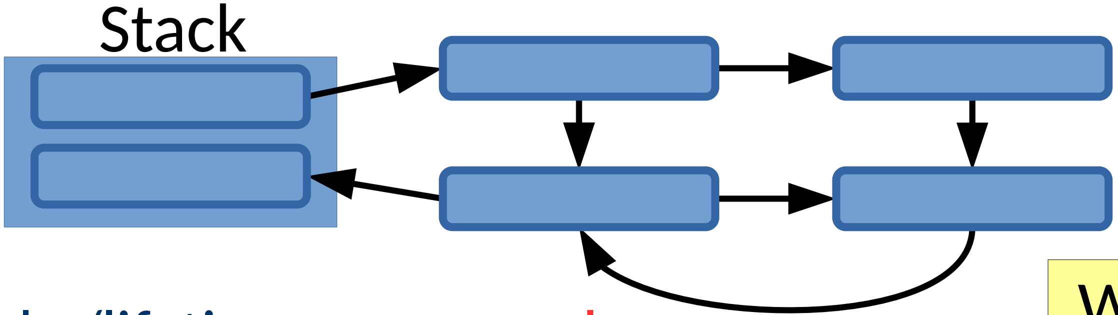
Managing Object Lifetimes



Object graphs/lifetimes are complex

Which pointers can I `delete` & when?

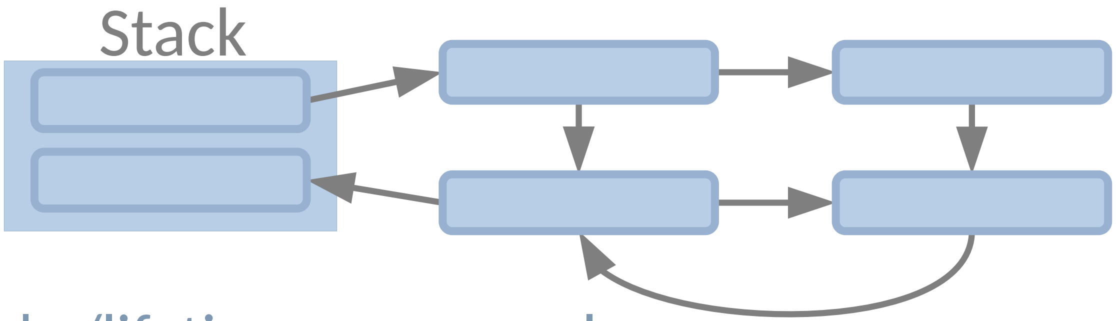
Managing Object Lifetimes



Object graphs/lifetimes are complex

Which pointers can I `delete` & when?

Managing Object Lifetimes (*Tangent*)



Object graphs/lifetimes are complex

When you ***use*** a data structure,
do you usually worry about these?

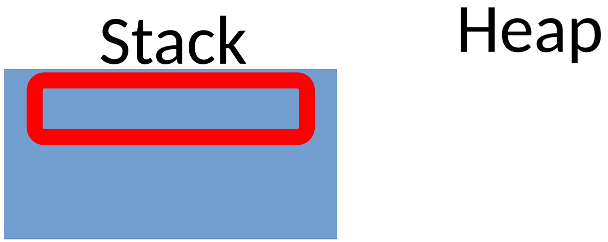
Managing Object Lifetimes (*Tangent*)

```
{
  std::vector<Widget> widgets
  widgets.emplace_back(3, “Fritter”);
  widgets.emplace_back(2, “Double chocolate”);
  widgets.emplace_back(3, “Maple Cream”);
}
```

Stack

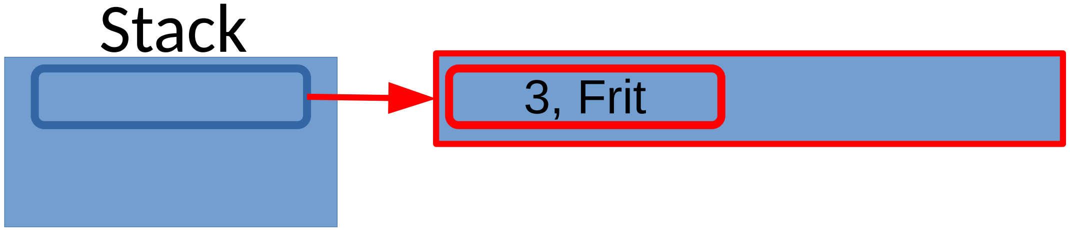
Managing Object Lifetimes (*Tangent*)

```
{
  std::vector<Widget> widgets
  widgets.emplace_back(3, “Fritter”);
  widgets.emplace_back(2, “Double chocolate”);
  widgets.emplace_back(3, “Maple Cream”);
}
```



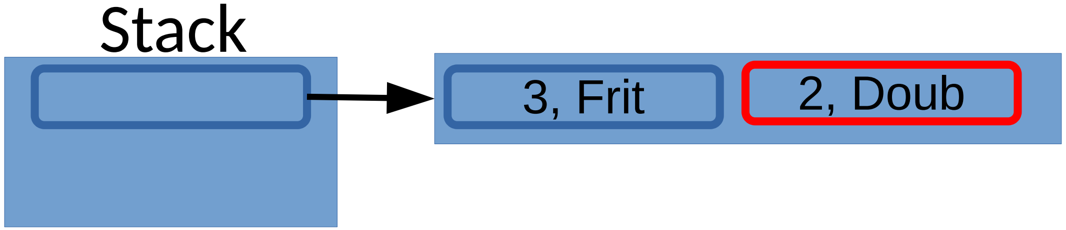
Managing Object Lifetimes (*Tangent*)

```
{
  std::vector<Widget> widgets
  widgets.emplace_back(3, "Fritter");
  widgets.emplace_back(2, "Double chocolate");
  widgets.emplace_back(3, "Maple Cream");
}
```



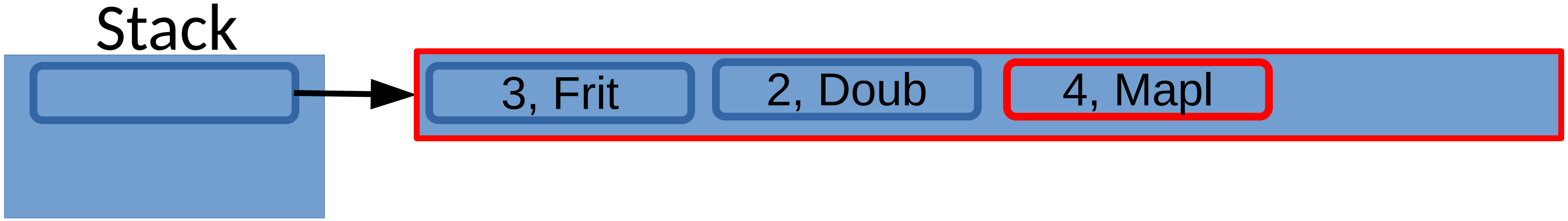
Managing Object Lifetimes (*Tangent*)

```
{
  std::vector<Widget> widgets
  widgets.emplace_back(3, "Fritter");
  widgets.emplace_back(2, "Double chocolate");
  widgets.emplace_back(3, "Maple Cream");
}
```



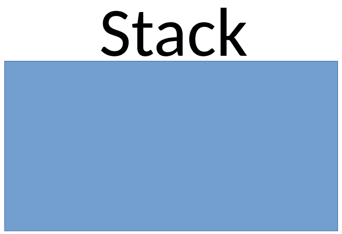
Managing Object Lifetimes (*Tangent*)

```
{
  std::vector<Widget> widgets
  widgets.emplace_back(3, "Fritter");
  widgets.emplace_back(2, "Double chocolate");
  widgets.emplace_back(4, "Maple Cream");
}
```

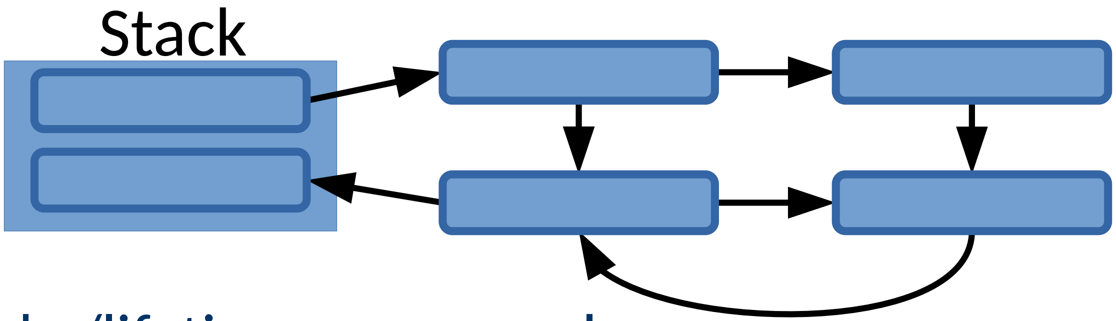


Managing Object Lifetimes (*Tangent*)

```
{
  std::vector<Widget> widgets
  widgets.emplace_back(3, “Fritter”);
  widgets.emplace_back(2, “Double chocolate”);
  widgets.emplace_back(4, “Maple Cream”);
}
```



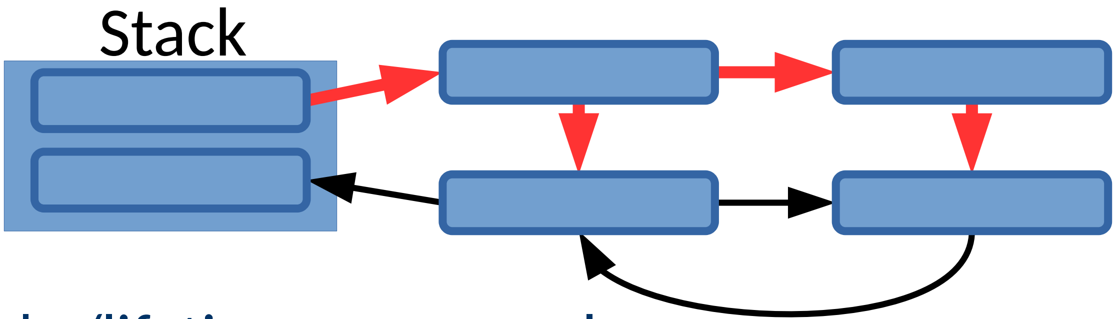
Managing Object Lifetimes (Revisiting)



Object graphs/lifetimes are complex

- Could this problem be solved using only **`std::vector`**?

Managing Object Lifetimes

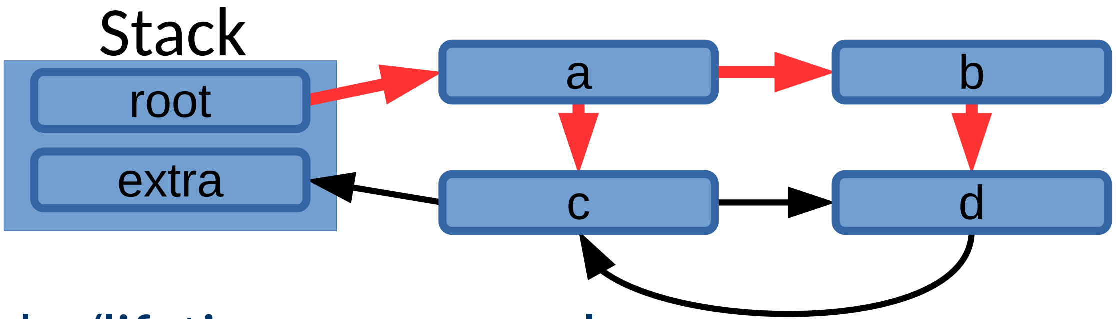


Object graphs/lifetimes are complex

- Could this problem be solved using only **`std::vector`**?

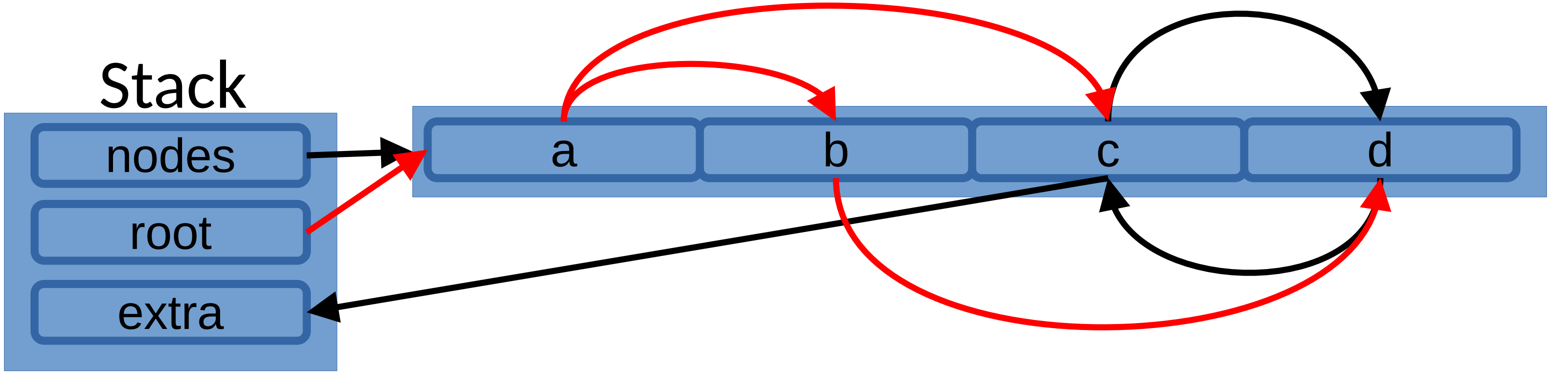
In a few different ways...

Managing Object Lifetimes

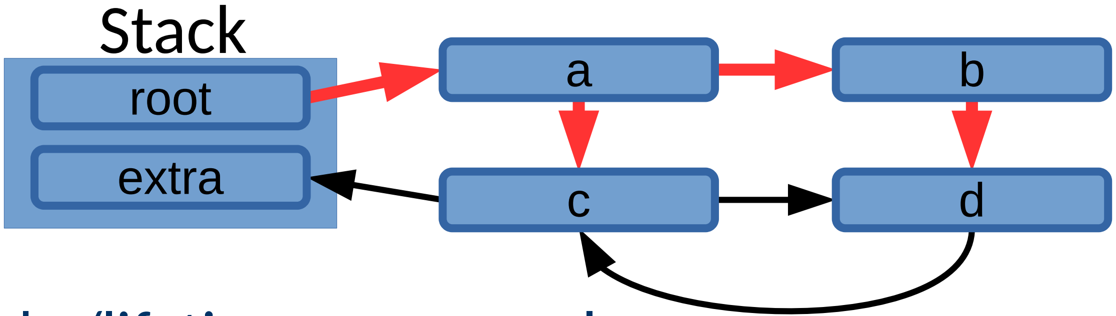


Object graphs/lifetimes are complex

- Could this problem be solved using only **`std::vector`**?



Managing Object Lifetimes

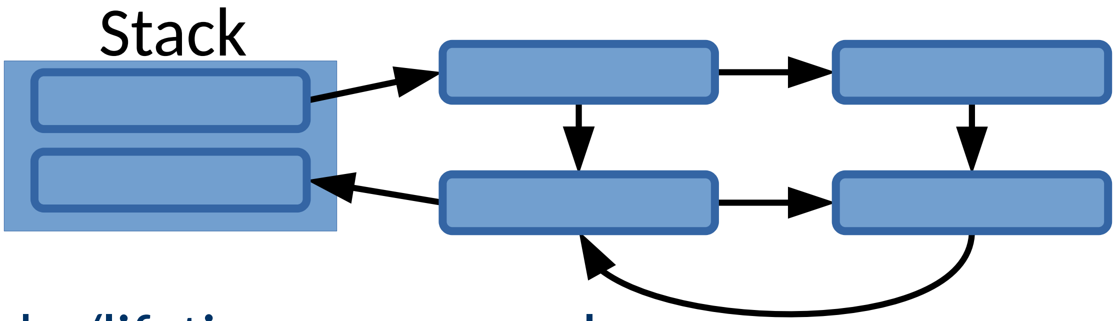


Object graphs/lifetimes are complex

- Could this problem be solved using only **`std::vector`**?

Could instead have a, b, c, d be vectors of 1 element.

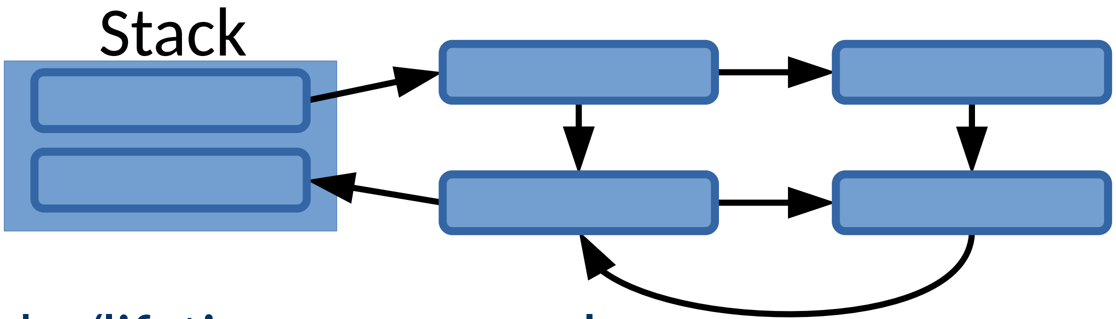
Managing Object Lifetimes



Object graphs/lifetimes are complex

- Could this problem be solved using only `std::vector`?
- Are there any downsides to doing so?

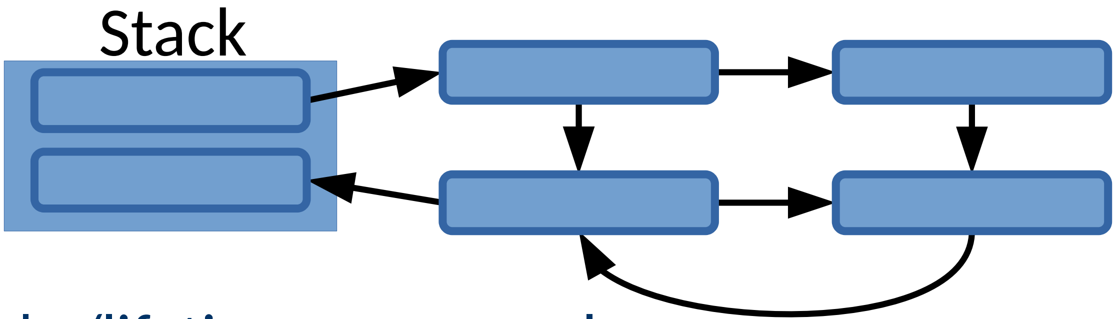
Managing Object Lifetimes



Object graphs/lifetimes are complex

- Could this problem be solved using only `std::vector`?
- Are there any downsides to doing so?
  - Unclear?
  - Unnecessary overheads?
  - Mismatched lifetimes?

Managing Object Lifetimes



Object graphs/lifetimes are complex

- Could this problem be solved using only `std::vector`?
- Are there any downsides to doing so?
  - Unclear?
  - Unnecessary overheads?
  - Mismatched lifetimes?

What we want is a clear, intentional way to express *ownership*.

Managing Object Lifetimes

- 2 types of ownership in modern C++

Managing Object Lifetimes

- 2 types of ownership in modern C++
  - Unique ownership (**`std::unique_ptr<T>`**)

```
auto w = std::make_unique<Widget>(0, "cruller");
```

Managing Object Lifetimes

- 2 types of ownership in modern C++
  - Unique ownership (**`std::unique_ptr<T>`**)

```
auto w = std::make_unique<Widget>(0, "cruller");
```

    - **`delete`**s the object when **`w`** goes out of scope
    - Automated (even with exceptions)

Managing Object Lifetimes

- 2 types of ownership in modern C++
  - Unique ownership (**`std::unique_ptr<T>`**)

```
auto w = std::make_unique<Widget>(0, "cruller");
```

    - **`delete`**s the object when **`w`** goes out of scope
    - Automated (even with exceptions)
    - Generally preferred

Managing Object Lifetimes

- 2 types of ownership in modern C++
  - Unique ownership (**`std::unique_ptr<T>`**)

```
auto w = std::make_unique<Widget>(0, "cruller");
```

    - **`delete`**s the object when **`w`** goes out of scope
    - Automated (even with exceptions)
    - Generally preferred

You can think of this as a vector of 1 item

Managing Object Lifetimes

- 2 types of ownership in modern C++
  - Unique ownership (`std::unique_ptr<T>`)

```
auto w = std::make_unique<Widget>(0, "cruller");
```

    - `delete`s the object when `w` goes out of scope
    - Automated (even with exceptions)
    - Generally preferred

  - Shared ownership (`std::shared_ptr<T>`)

```
auto w = std::make_shared<Widget>(0, "ponchik");
```

Managing Object Lifetimes

- 2 types of ownership in modern C++
  - Unique ownership (**std::unique_ptr<T>**)

```
auto w = std::make_unique<Widget>(0, "cruller");
```

    - **delete**s the object when **w** goes out of scope
    - Automated (even with exceptions)
    - Generally preferred
  - Shared ownership (**std::shared_ptr<T>**)

```
auto w = std::make_shared<Widget>(0, "ponchik");
```

    - Counts the number of owners
    - **delete**s the object when # owners --> 0

Managing Object Lifetimes

- 2 types of ownership in modern C++
  - Unique ownership (`std::unique_ptr<T>`)

```
auto w = std::make_unique<Widget>(0, "cruller");
```

    - `delete`s the object when `w` goes out of scope
    - Automated (even with exceptions)
    - Generally preferred
  - Shared ownership (`std::shared_ptr<T>`)

```
auto w = std::make_shared<Widget>(0, "ponchik");
```

    - Counts the number of owners
    - `delet`

What happens if you have a cycle?

Managing Object Lifetimes

- 2 types of ownership in modern C++
  - Unique ownership (**std::unique_ptr<T>**)

```
auto w = std::make_unique<Widget>(0, "cruller");
```

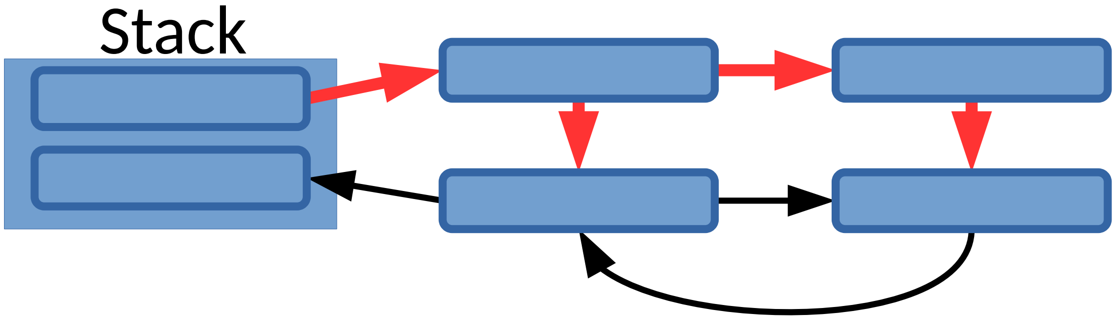
    - **delete**s the object when **w** goes out of scope
    - Automated (even with exceptions)
    - Generally preferred
  - Shared ownership (**std::shared_ptr<T>**)

```
auto w = std::make_shared<Widget>(0, "ponchik");
```

    - Counts the number of owners
    - **delete**s the object when # owners --> 0
- Ownership *can* also be transferred

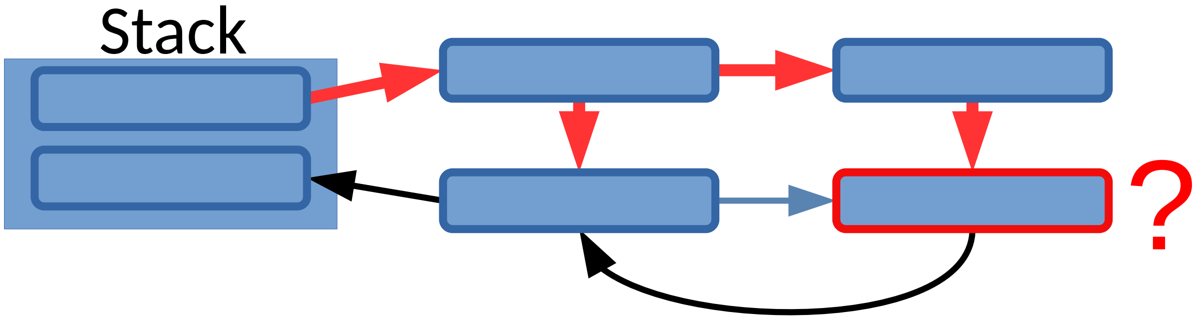
Managing Object Lifetimes

- A few rules:
  - Every object has (preferably) one owner



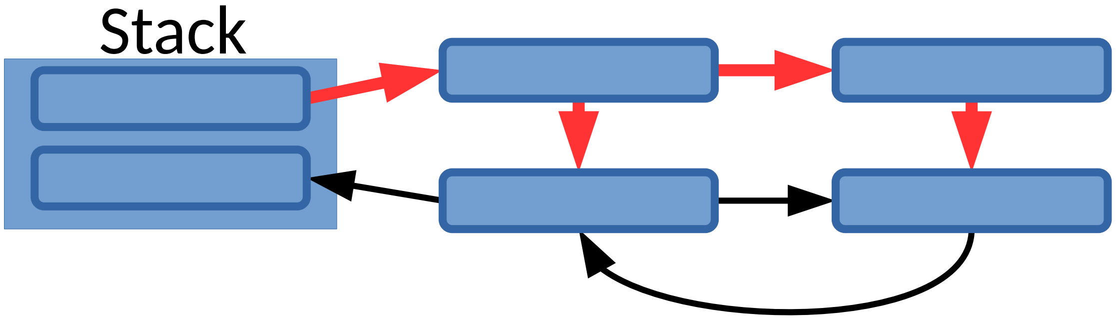
Managing Object Lifetimes

- A few rules:
  - Every object has (preferably) one owner



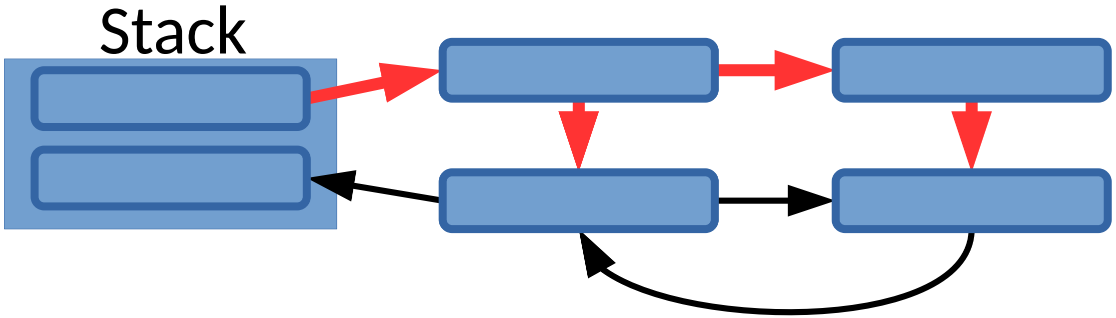
Managing Object Lifetimes

- A few rules:
  - Every object has (preferably) one owner
  - No object outlives the scope of its owning pointer



Managing Object Lifetimes

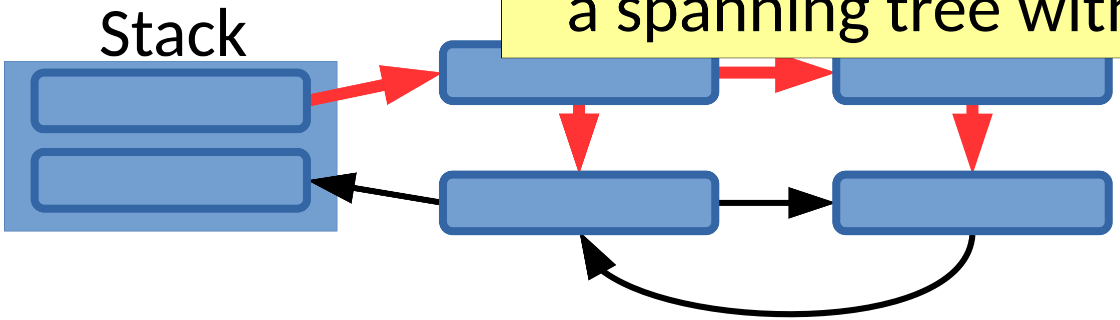
- A few rules:
  - Every object has (preferably) one owner
  - No object outlives the scope of its owning pointer
  - Non-owning pointers/references can be unlimited
    - But should not outlive the owning scope by design



Managing Object Lifetimes

- A few rules:
  - Every object has (preferably) one owner
  - No object outlives the scope of its owning pointer
  - Non-owning pointers/references can be unlimited
    - But should not outlive the owning scope by design

Note: Unique owning pointers form a spanning tree within the heap.



Functions (a slight digression)

What is the signature to…

- pass an argument of class type X to a function?

Functions (a slight digression)

What is the signature to…

- pass an argument of class type X to a function?

```
foo(const X&)
```

Functions (a slight digression)

What is the signature to…

- pass an argument of class type X to a function?

```
foo(const X&)
```

- pass a *mutable* argument of class type X to a function?

Functions (a slight digression)

What is the signature to…

- pass an argument of class type X to a function?

```
foo(const X&)
```

- pass a *mutable* argument of class type X to a function?

```
foo(X&)
```

Functions (a slight digression)

What is the signature to…

- pass an argument of class type X to a function?

  `foo(const X&)`

- pass a *mutable* argument of class type X to a function?

  `foo(X&)`

- pass an instance of X to a function making a copy?

Functions (a slight digression)

What is the signature to…

- pass an argument of class type X to a function?

```
foo(const X&)
```

- pass a *mutable* argument of class type X to a function?

```
foo(X&)
```

- pass an instance of X to a function making a copy?

```
foo(X)
```

Using What You Know

```
void foo( [1] );

void bar() {
  auto [2] ique<Widget>(42, "churro");
  foo( [2] );
}
```

- What should go in 1 and 2 to pass **`w`** to **`foo`**?
  - (It may depend on what you want to do...)
  - Do you just want to give foo *access* to the Widget?
  - Do you want foo to *modify* the ownership?
  - Do you want to *transfer* ownership to foo?

Using What You Know

```
void foo(      1      );

void bar() {
  auto w = std::make_unique<Widget>(42, "churro");
  foo(      2      );
}
```

- What should go in 1 and 2 to pass **`w`** to **`foo`**?
  - (It may depend on what you want to do…)
  - Do you just want to give foo *access* to the Widget?
  - Do you want foo to *modify* the ownership?
  - Do you want to *transfer* ownership to foo?

Note: These are behaviors that would already happen.
Smart pointers make them *explicit* and *automatic*.

General Resource Management

- Memory management is just one example of *resource management*.

General Resource Management

- Memory management is just one example of *resource management*.
  - Properly acquiring & releasing resources

General Resource Management

- Memory management is just one example of *resource management*.
  - Properly acquiring & releasing resources
    - No double acquisition.
    - No double free.
    - No use after free.
    - No leaks

General Resource Management

- Memory management is just one example of *resource management*.
  - Properly acquiring & releasing resources
    - No double acquisition.
    - No double free.
    - No use after free.
    - No leaks
  - What *other* resources do you manage?

General Resource Management

- Memory management is just one example of *resource management*.
  - Properly acquiring & releasing resources
    - No double acquisition.
    - No double free.
    - No use after free.
    - No leaks
  - What *other* resources do you manage?
    - Files
    - Locks
    - Database connections
    - Printers
    - ...

General Resource Management

- The problem is pervasive enough to have general solutions

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: `?`

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: **`with`**

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: **`with`**
  - C#: **`using`**
  - Java: **`try`**-with-resources

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: **`with`**
  - C#: **`using`**
  - Java: **`try`**-with-resources
  - C++: **?**

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: **`with`**
  - C#: **`using`**
  - Java: **`try`**-with-resources
  - C++: RAII (Resource Acquisition is Initialization)

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: `with`
  - C#: `using`
  - Java: `try`-with-resources
  - C++: RAII (Resource Acquisition is Initialization)
- Goal: Simplify & control the lifetimes of resources

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: `with`
  - C#: `using`
  - Java: `try`-with-resources
  - C++: RAII (Resource Acquisition is Initialization)
- Goal: Simplify & control the lifetimes of resources
- RAII
  - Bind the lifetime of the resource to object lifetime

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: `with`
  - C#: `using`
  - Java: `try`-with-resources
  - C++: RAII (Resource Acquisition is Initialization)
- Goal: Simplify & control the lifetimes of resources
- RAII
  - Bind the lifetime of the resource to object lifetime
  - Acquire the resource in the constructor

General Resource Management

- The problem is pervasive enough to have general solutions
  - Python: `with`
  - C#: `using`
  - Java: `try`-with-resources
  - C++: RAII (Resource Acquisition is Initialization)
- Goal: Simplify & control the lifetimes of resources
- RAII
  - Bind the lifetime of the resource to object lifetime
  - Acquire the resource in the constructor
  - Release the resource in the destructor

General Resource Management

- Memory

```
void memoryResource() {
  auto w = std::make_unique<Widget>(3, "bofrot");
  foo(*w);
}
```

General Resource Management

- Memory

```
void memoryResource() {
   auto w = std::make_unique<Widget>(3, "bofrot");
   foo(*w);
}
```

w is *automatically* deallocated here.

General Resource Management

- Memory

```
void memoryResource() {
  auto w = std::make_unique<Widget>(3, "bofrot");
  foo(*w);
}
```

w is *automatically* deallocated here.

- Files

```
void fileResource() {
  auto out = std::ofstream{"output.txt"};
  out << "Boston cream\n";
}
```

General Resource Management

- Memory

```
void memoryResource() {
  auto w = std::make_unique<Widget>(3, "bofrot");
  foo(*w);
}
```

w is automatically deallocated here.

- Files

```
void fileResource() {
  auto out = std::ofstream{"output.txt"};
  out << "
}
```

out is automatically flushed & closed here.

General Resource Management

- Memory

```
void memoryResource() {
  auto w = std::make_unique<Widget>(3, "bofrot");
  foo(*w);
}
```

w is automatically deallocated here.

- Files

```
void fileResource() {
  auto out = std::ofstream{"output.txt"};
  out << "
}
```

out is automatically flushed & closed here.

- Because they are scoped, they handle exceptions & multiple return statements!

General Resource Management

- How does RAII relate to managing complexity?

General Resource Management

- How does RAII relate to managing complexity?
  - It makes resource designs explicit
  - It makes managing them automatic
  - It removes temporal coupling
  - It promotes composition & independence

General Resource Management

- How does RAII relate to managing complexity?
  - It makes resource designs explicit
  - It makes managing them automatic
  - It removes temporal coupling
  - It promotes composition & independence
- NOTE: What happens when you copy a resource object?

General Resource Management

- How does RAII relate to managing complexity?
  - It makes resource designs explicit
  - It makes managing them automatic
  - It removes temporal coupling
  - It promotes composition & independence
- NOTE: What happens when you copy a resource object?
  - In many cases, it is explicitly forbidden

Why?

General Resource Management

- How does RAII relate to managing complexity?
  - It makes resource designs explicit
  - It makes managing them automatic
  - It removes temporal coupling
  - It promotes composition & independence
- NOTE: What happens when you copy a resource object?
  - In many cases, it is explicitly forbidden
  - You can use **`std::move()`** to *transfer* resource ownership

Operating on Collections

- Iterating over collections can be painful

```
void oops() {
  std::vector numbers = {0, 1, 2, 3, 4};
  for (unsigned i = 0, e = 4; i <= 4; ++i) {
    std::cout << numbers[i] << "\n";
  }
}
```

Operating on Collections

- Iterating over collections can be painful

```
void oops() {
  std::vector numbers = {0, 1, 2, 3, 4};
  for (unsigned i = 0, e = 4; i <= 4; ++i) {
    std::cout << numbers[i] << "\n";
  }
}
```

- Range based for loops are preferable

```
void nice() {
  std::vector numbers = {0, 1, 2, 3, 4};
  for (auto number : numbers) {
    std::cout << number << "\n";
  }
}
```

Operating on Collections

- Iterating over collections can be painful

```
void oops() {
  std::vector numbers = {0, 1, 2, 3, 4};
  for (unsigned i = 0, e = 4; i <= 4; ++i) {
    std::cout << numbers[i] << "\n";
  }
}
```

- Range based for loops are pref

The “collection” can be anything with **`begin()`** and **`end()`** methods.

```
void nice() {
  std::vector numbers = {0, 1, 2, 3, 4};
  for (auto number : numbers) {
    std::cout << number << "\n";
  }
}
```

Operating on Collections

- Passing collections around can be error prone.

```
void oops(const std::vector<int> numbers) {
   ...
}
```

Operating on Collections

- Passing collections around can be error prone.

```
void oops(const std::vector<int> numbers) {
  ...
}
```

- Avoid unnecessary copies.

```
void better(const std::vector<int>& numbers) {
  ...
}
```

Operating on Collections

- Passing collections around can be error prone.

```
void oops(const std::vector<int> numbers) {
  ...
}
```

- Avoid unnecessary copies.

```
void better(const std::vector<int>& numbers) {
  ...
}
```

- Use std::span in C++20 for flexibility & correctness by design

```
void good(const std::span<int> numbers) {
  ...
}
```

Guideline Support Library

Some common classes for better code, specifically:

Guideline Support Library

Some common classes for better code, specifically:

- **`std::span<T>, gsl::span<T>`**
  - Makes interfaces generic & safer if you do not have C++20

[demo]

Guideline Support Library

Some common classes for better code, specifically:

- **`std::span<T>, gsl::span<T>`**
  - Makes interfaces generic & safer if you do not have c++20

[demo]

- **`std::string_view<T>`**
  - Avoid copying strings
  - Avoid conversions to and from C strings
    (a common mistake!)

Guideline Support Library

Some common classes for better code, specifically:

- `std::span<T>, gsl::span<T>`
  - Makes interfaces generic & safer

[demo]

- `std::string_view<T>`
  - Avoid copying strings
  - Avoid conversions to and from C strings
    (a common mistake!)
- Both of these abstractions are *non*-owning

λ (Lambdas)

- How should you check whether a list contains a number greater than 3?

λ (Lambdas)

- How should you check whether a list contains a number greater than 3?

```
bool hasGreaterThan3 = false;
for (auto number : numbers) {
  if (number > 3) {
    hasGreaterThan3 = true;
  }
}
```

λ (Lambdas)

- How should you check whether a list contains a number greater than 3?

```
bool hasGreaterThan3 = false;
for (auto number : numbers) {
  if (number > 3) {
    hasGreaterThan3 = true;
  }
}
```

Using a general purpose loop *hides* the high level intentions.

λ (Lambdas)

- How should you check whether a list contains a number greater than 3?

```
bool hasGreaterThan3 = false;
for (auto number : numbers) {
  if (number > 3) {
    hasGreaterThan3 = true;
  }
}
```

Using a general purpose loop *hides* the high level intentions.

```
bool hasGreaterThan3 =
  std::any_of(numbers.begin(), numbers.end(),
    [](auto number) { return number > 3; });
```

λ (Lambdas)

- How should you check whether a list contains a number greater than 3?

```cpp
bool hasGreaterThan3 = false;
for (auto number : numbers) {
  if (number > 3) {
    hasGreaterThan3 = true;
  }
}
```

Using a general purpose loop *hides* the high level intentions.

```cpp
bool hasGreaterThan3 =
  std::any_of(numbers.begin(), numbers.end(),
    [](auto number) { return number > 3; });
```

In C++20:

```cpp
bool hasGreaterThan3 =
  std::ranges::any_of(numbers,
    [](auto number) { return number > 3; });
```

λ (Lambdas)

- Lambdas allow you to create small, self contained functions local to other code

```
[local1, local2](auto arg1, auto arg2) {
  ...
}
```

λ (Lambdas)

- Lambdas allow you to create small, self contained functions local to other code

```
[local1, local2](auto arg1, auto arg2) {
   ...
}
```

You can capture arguments from the local scope.

λ (Lambdas)

- Lambdas allow you to create small, self contained functions local to other code

```
[local1, local2](auto arg1, auto arg2) {
   ...
}
```

Additional arguments are passed in when invoked.

λ (Lambdas)

- Lambdas allow you to create small, self contained functions local to other code

```
[local1, local2](auto arg1, auto arg2) {
  ...
}
```

- Lambdas allow you to use generic library functions in a clear, well localized fashion.

λ (Lambdas)

- Lambdas allow you to create small, self contained functions local to other code

```
[local1, local2](auto arg1, auto arg2) {
  ...
}
```

- Lambdas allow you to use generic library functions in a clear, well localized fashion.

```
auto found =
  std::ranges::find_if(numbers,
    [](auto number) { return number > 3; });
std::cout << *found << " is greater than 3.\n";
```

λ (Lambdas)

- Lambdas allow you to create small, self contained functions local to other code

```
[local1, local2](auto arg1, auto arg2) {
  ...
}
```

- Lambdas allow you to use generic library functions in a clear, well localized fashion.

```
auto found =
  std::ranges::find_if(numbers,
    [](auto number) { return number > 3; });
std::cout << *found                    .\n";
```

See <algorithm>

λ (Lambdas)

- Lambdas allow you to create small, self contained functions local to other code

```
[local1, local2](auto arg1, auto arg2) {
  …
}
```

I will expect you to make use of built in algorithms and lambdas instead of raw loops from now on.

- Lambdas allow you to use generic library functions in a clear, well localized fashion.

```
auto found =
  std::ranges::find_if(numbers,
    [](auto number) { return number > 3; });
std::cout << *found … .\n";
```

See <algorithm>

Exceptions

- Not new, but maybe new to you in C++

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types `<stdexcept>`

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types `<stdexcept>`

```
try {
  throw std::runtime_error("uh oh...");
} catch (const std::runtime_error& e) {
  std::cout << "Exception message: " << e.what();
}
```

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types <stdexcept>

```
try {
  throw std::runtime_error("uh oh...");
} catch (const std::runtime_error& e) {
  std::cout << "Exception message: " << e.what();
}
```

Throw by value.

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types `<stdexcept>`

```
try {
  throw std::runtime_error("uh oh...");
} catch (const std::runtime_error& e) {
  std::cout << "Exception message: " << e.what();
}
```

Catch by reference.

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types `<stdexcept>`

```
try {
  throw std::runtime_error("uh oh...");
} catch (const std::runtime_error& e) {
  std::cout << "Exception message: " << e.what();
}
```

Error messages.

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types `<stdexcept>`
- Or you can create custom exceptions

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types `<stdexcept>`
- Or you can create custom exceptions

```
class MyException : public std::runtime_error {
public:
  const char * what() const override {
    ...
  }
};
```

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types `<stdexcept>`
- Or you can create custom exceptions

```
class MyException : public std::runtime_error {
public:
  const char * what() const override {
    ...
  }
};
```

Exceptions

- Not new, but maybe new to you in C++
- Can use existing exception types **<stdexcept>**
- Or you can create custom exceptions

```
class MyException : public std::runtime_error {
public:
   const char * what() const override {
      ...
   }
};
```

More…

- `std::array<T,N>`

More...

- `std::array<T,N>`
- `nullptr`

More…

- **`std::array<T,N>`**
- **`nullptr`**
- **`auto`** (even for return & lambda arg types)

More…

- `std::array<T,N>`
- `nullptr`
- `auto` (even for return & lambda arg types)
- `constexpr`
- type safe enums
- delegating constructors
- `using` instead of `typedef`

More…

- **`std::array<T,N>`**
- **`nullptr`**
- **`auto`** (even for return & lambda arg types)
- **`constexpr`**
- type safe enums
- delegating constructors
- **`using`** instead of **`typedef`**
- Destructuring: **`auto [x, y] = std::make_pair(3,4);`**
- …

More…

- `std::array<T,N>`
- `nullptr`
- `auto` (even for return & lambda arg types)
- `constexpr`
- type safe enums
- delegating constructors
- `using` instead of `typedef`
- Destru
- …

And these are from almost a decade ago.