Lambda Evolution - Past, Present and Future

Y. Bernat

Core C++ Meetup, Dec. 2017

Outline

Overview

- A new way (C++11) to write a function

Overview

- A new way (C++11) to write a function
- Can be defined locally (while “normal” functions can’t)

Overview

- A new way (C++11) to write a function
- Can be defined locally (while “normal” functions can’t)
- That’s all

Overview

- A new way (C++11) to write a function
- Can be defined locally (while “normal” functions can’t)
- That’s all :)

Outline

Predicate for STL - The Problem

- STL includes many algorithms for common actions on a collection

Predicate for STL - The Problem

- STL includes many algorithms for common actions on a collection
  - `std::find_if` - find the first item that matches a criteria

Predicate for STL - The Problem

- STL includes many algorithms for common actions on a collection
  - `std::find_if` - find the first item that matches a criteria
  - `std::copy_if` - copy all items that match a criteria

Predicate for STL - The Problem

- STL includes many algorithms for common actions on a collection
  - `std::find_if` - find the first item that matches a criteria
  - `std::copy_if` - copy all items that match a criteria
  - ...

Predicate for STL - The Problem

- STL includes many algorithms for common actions on a collection
  - `std::find_if` - find the first item that matches a criteria
  - `std::copy_if` - copy all items that match a criteria
  - ...
- How do we pass the criteria to the algorithm?

Use a Simple Function

```
1  bool between3and7 (int x) {
2    return 3 <= x && x <= 7;
3  }
4  void f () {
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           &between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 &between3and7);
17 }
```

Use a Simple Function

```
1  bool between3and7 (int x) {
2    return 3 <= x && x <= 7;
3  }
4  void f () {
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           &between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 &between3and7);
17 }
```

Use a Simple Function

```
1  bool between3and7 (int x) {
2    return 3 <= x && x <= 7;
3  }
4  void f () {
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           &between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 &between3and7);
17 }
```

Use a Simple Function

```
 1  bool between3and7 (int x) {
 2    return 3 <= x && x <= 7;
 3  }
 4  void f () {
 5    std::vector<int> v;
 6    // ... fill v with data ...
 7    auto i = std::find_if (v.begin(), v.end(),
 8                           &between3and7);
 9    if (i != v.end()) {
10      std::cout << "First item between 3 and 7 is "
11                << *i << '\n';
12    }
13    std::vector<int> v2;
14    std::copy_if (v.begin(), v.end(),
15                  std::back_inserter(v2),
16                  &between3and7);
17  }
```

Use a Simple Function

```
1  bool between3and7 (int x) {
2    return 3 <= x && x <= 7;
3  }
4  void f () {
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           &between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 &between3and7);
17 }
```

Use a Simple Function

```
1  bool between3and7 (int x) {
2    return 3 <= x && x <= 7;
3  }
4  void f () {
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           &between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 &between3and7);
17 }
```

Simple Function - Pros and Cons

- Pros:

Simple Function - Pros and Cons

- Pros:
  - It's simple

Simple Function - Pros and Cons

- Pros:
  - It's simple
  - It's well known

Simple Function - Pros and Cons

- Pros:
  - It's simple
  - It's well known
- Cons:

Simple Function - Pros and Cons

- Pros:
  - It's simple
  - It's well known
- Cons:
  - We need a new one for each case (e.g. `between0and42`)

Simple Function - Pros and Cons

- Pros:
  - It's simple
  - It's well known
- Cons:
  - We need a new one for each case (e.g. `between0and42`)
  - We have to find the correct scope to define it

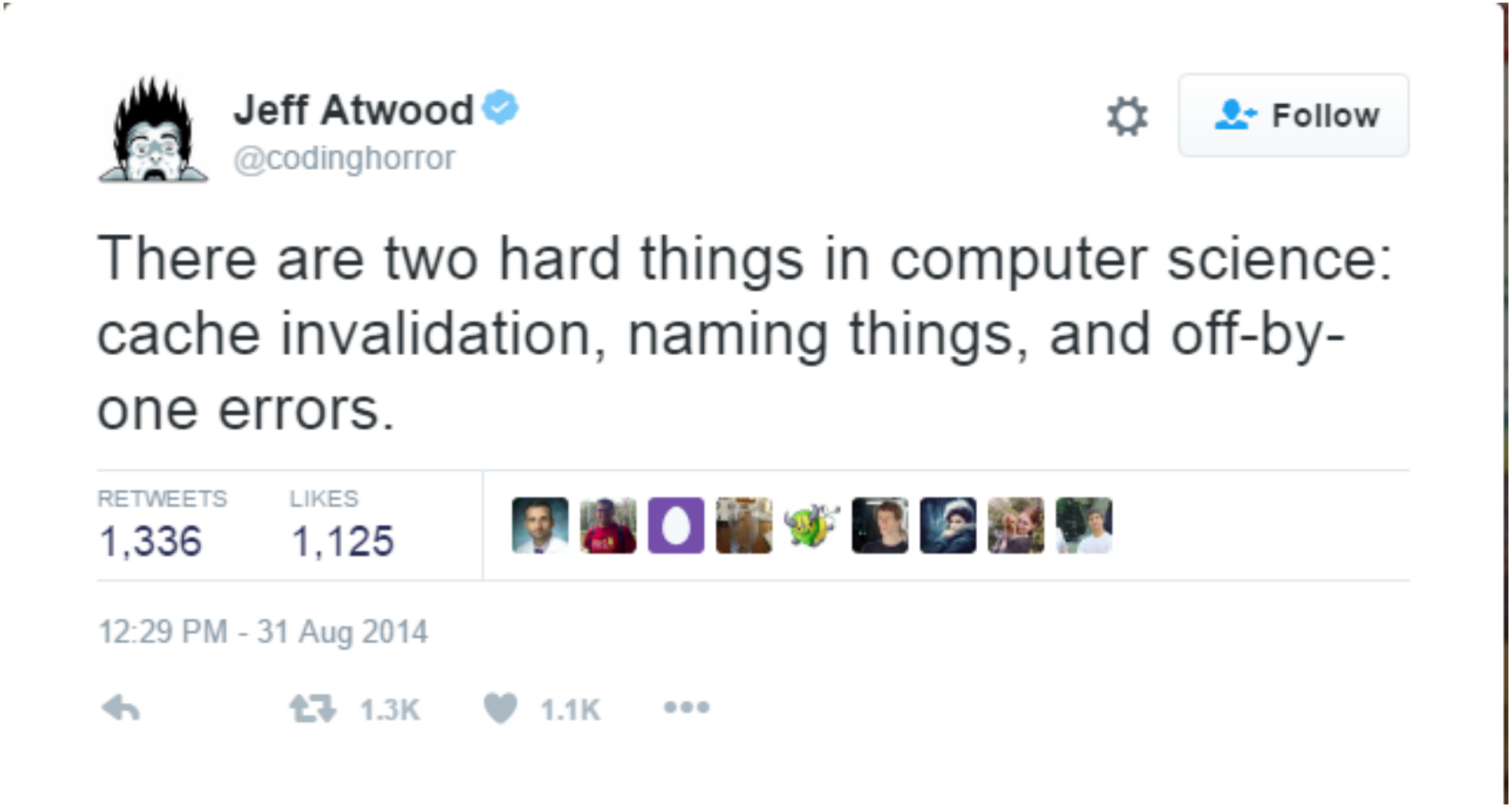
Simple Function - Pros and Cons

- Pros:
  - It's simple
  - It's well known
- Cons:
  - We need a new one for each case (e.g. `between0and42`)
  - We have to find the correct scope to define it
  - We need to find a good name each time

Simple Function - Pros and Cons

- Pros:
  - It's simple
  - It's well known
- Cons:
  - We need a new one for each case (e.g. `between0and42`)
  - We have to find the correct scope to define it
  - We need to find a good name each time
  - The logic is spread over the code instead of being local

Find Good Names



Use a Functor

- Templates are heavily-used with STL algorithms

Use a Functor

- Templates are heavily-used with STL algorithms
- The predicate doesn't have to be a function pointer

Use a Functor

- Templates are heavily-used with STL algorithms
- The predicate doesn't have to be a function pointer
- It can be anything the "behaves like a function" - anything that we can use () on it

Use a Functor

- Templates are heavily-used with STL algorithms
- The predicate doesn't have to be a function pointer
- It can be anything the "behaves like a function" - anything that we can use () on it
- A pointer to a function is a good fit but not the only one

Use a Functor

- Templates are heavily-used with STL algorithms
- The predicate doesn't have to be a function pointer
- It can be anything the "behaves like a function" - anything that we can use () on it
- A pointer to a function is a good fit but not the only one
- We can overload the call operator () of a class so any object of this type is also "behaves like a function"

Use a Functor

- Templates are heavily-used with STL algorithms
- The predicate doesn't have to be a function pointer
- It can be anything the "behaves like a function" - anything that we can use () on it
- A pointer to a function is a good fit but not the only one
- We can overload the call operator () of a class so any object of this type is also "behaves like a function"
- Such an object is commonly known as "function object" or "functor" (even if this name isn't accurate)

Functor Example

```
1  struct Between {
2    int m_l, m_h;
3    Between(int l, int h) : m_l(l), m_h(h) {}
4    bool operator()(int x) {
5      return m_l <= x && x <= m_h;
6    }
7  };
8  // ...
9  auto i = std::find_if(v.begin(), v.end(),
10                       Between(3, 7));
11 auto j = std::find_if(v.begin(), v.end(),
12                       Between(0, 42));
```

Functor Example

```
1  struct Between {
2    int m_l, m_h;
3    Between(int l, int h) : m_l(l), m_h(h) {}
4    bool operator()(int x) {
5      return m_l <= x && x <= m_h;
6    }
7  };
8  // ...
9  auto i = std::find_if(v.begin(), v.end(),
10                       Between(3, 7));
11 auto j = std::find_if(v.begin(), v.end(),
12                       Between(0, 42));
```

Functor Example

```
1  struct Between {
2    int m_l, m_h;
3    Between(int l, int h) : m_l(l), m_h(h) {}
4    bool operator()(int x) {
5      return m_l <= x && x <= m_h;
6    }
7  };
8  // ...
9  auto i = std::find_if(v.begin(), v.end(),
10                       Between(3, 7));
11 auto j = std::find_if(v.begin(), v.end(),
12                       Between(0, 42));
```

Functor Example

```
1  struct Between {
2    int m_l, m_h;
3    Between(int l, int h) : m_l(l), m_h(h) {}
4    bool operator()(int x) {
5      return m_l <= x && x <= m_h;
6    }
7  };
8  // ...
9  auto i = std::find_if(v.begin(), v.end(),
10                       Between(3, 7));
11 auto j = std::find_if(v.begin(), v.end(),
12                       Between(0, 42));
```

Functor Example

```
1  struct Between {
2    int m_l, m_h;
3    Between(int l, int h) : m_l(l), m_h(h) {}
4    bool operator()(int x) {
5      return m_l <= x && x <= m_h;
6    }
7  };
8  // ...
9  auto i = std::find_if(v.begin(), v.end(),
10                       Between(3, 7));
11 auto j = std::find_if(v.begin(), v.end(),
12                       Between(0, 42));
```

Functor Example

```
1  struct Between {
2    int m_l, m_h;
3    Between(int l, int h) : m_l(l), m_h(h) {}
4    bool operator()(int x) {
5      return m_l <= x && x <= m_h;
6    }
7  };
8  // ...
9  auto i = std::find_if(v.begin(), v.end(),
10                        Between(3, 7));
11 auto j = std::find_if(v.begin(), v.end(),
12                        Between(0, 42));
```

Functor Example

```
 1  struct Between {
 2    int m_l, m_h;
 3    Between(int l, int h) : m_l(l), m_h(h) {}
 4    bool operator()(int x) {
 5      return m_l <= x && x <= m_h;
 6    }
 7  };
 8  // ...
 9  auto i = std::find_if(v.begin(), v.end(),
10                        Between(3, 7));
11  auto j = std::find_if(v.begin(), v.end(),
12                        Between(0, 42));
```

Functor Example

```
 1  struct Between {
 2    int m_l, m_h;
 3    Between(int l, int h) : m_l(l), m_h(h) {}
 4    bool operator()(int x) {
 5      return m_l <= x && x <= m_h;
 6    }
 7  };
 8  // ...
 9  auto i = std::find_if(v.begin(), v.end(),
10                        Between(3, 7));
11  auto j = std::find_if(v.begin(), v.end(),
12                        Between(0, 42));
```

Functor Example

```
1  struct Between {
2    int m_l, m_h;
3    Between(int l, int h) : m_l(l), m_h(h) {}
4    bool operator()(int x) {
5      return m_l <= x && x <= m_h;
6    }
7  };
8  // ...
9  auto i = std::find_if(v.begin(), v.end(),
10                       Between(3, 7));
11 auto j = std::find_if(v.begin(), v.end(),
12                       Between(0, 42));
```

Functor - Pros and Cons

- Pros:

Functor - Pros and Cons

- Pros:
  - More generic one for multiple cases

Functor - Pros and Cons

- Pros:
  - More generic one for multiple cases
- Cons:

Functor - Pros and Cons

- Pros:
  - More generic one for multiple cases
- Cons:
  - We (still) have to find the correct scope to define it

Functor - Pros and Cons

- Pros:
  - More generic one for multiple cases
- Cons:
  - We (still) have to find the correct scope to define it
  - We (still) need to find a good name each time

Functor - Pros and Cons

- Pros:
  - More generic one for multiple cases
- Cons:
  - We (still) have to find the correct scope to define it
  - We (still) need to find a good name each time
  - The logic is (still) spread over the code instead of being local

Functor - Pros and Cons

- Pros:
  - More generic one for multiple cases
- Cons:
  - We (still) have to find the correct scope to define it
  - We (still) need to find a good name each time
  - The logic is (still) spread over the code instead of being local
  - A lot of boilerplate

Outline

Lambda Expression (Function)

- Added to C++11

Lambda Expression (Function)

- Added to C++11
- Basic syntax - `[ ]( ){ }`

Lambda Expression (Function)

- Added to C++11
- Basic syntax - `[ ]( ){ }`
- `[ ]` - “Lambda function introducer” (not a real formal term)

Lambda Expression (Function)

- Added to C++11
- Basic syntax - `[ ]( ){ }`
- `[ ]` - "Lambda function introducer" (not a real formal term)
  - Also for capturing, which we'll discuss later

Lambda Expression (Function)

- Added to C++11
- Basic syntax - `[ ]( ){ }`
- `[ ]` - "Lambda function introducer" (not a real formal term)
  - Also for capturing, which we'll discuss later
- `( )` - for parameters, as in every function (but usually can be omitted if empty)

Lambda Expression (Function)

- Added to C++11
- Basic syntax - `[ ]( ){ }`
- `[ ]` - "Lambda function introducer" (not a real formal term)
  - Also for capturing, which we'll discuss later
- `( )` - for parameters, as in every function (but usually can be omitted if empty)
- `{ }` - the (lambda) function body

Use a Lambda Function

```
1  auto i = std::find_if(v.begin(), v.end(),
2                        [](int i){
3                          return 3 <= i && i <= 7;
4                        });
5  auto j = std::find_if(v.begin(), v.end(),
6                        [](int i){
7                          return 0 <= i && i <= 42;
8                        });
```

Use a Lambda Function

```
1  auto i = std::find_if(v.begin(), v.end(),
2                        [](int i){
3                          return 3 <= i && i <= 7;
4                        });
5  auto j = std::find_if(v.begin(), v.end(),
6                        [](int i){
7                          return 0 <= i && i <= 42;
8                        });
```

Use a Lambda Function

```
1  auto i = std::find_if(v.begin(), v.end(),
2                        [](int i){
3                          return 3 <= i && i <= 7;
4                        });
5  auto j = std::find_if(v.begin(), v.end(),
6                        [](int i){
7                          return 0 <= i && i <= 42;
8                        });
```

Use a Lambda Function

```
1  auto i = std::find_if(v.begin(), v.end(),
2                        [](int i){
3                          return 3 <= i && i <= 7;
4                        });
5  auto j = std::find_if(v.begin(), v.end(),
6                        [](int i){
7                          return 0 <= i && i <= 42;
8                        });
```

Use a Lambda Function

```
1  auto i = std::find_if(v.begin(), v.end(),
2                        [](int i){
3                          return 3 <= i && i <= 7;
4                        });
5  auto j = std::find_if(v.begin(), v.end(),
6                        [](int i){
7                          return 0 <= i && i <= 42;
8                        });
```

Use a Lambda Function

```
1  auto i = std::find_if(v.begin(), v.end(),
2                        [](int i){
3                          return 3 <= i && i <= 7;
4                        });
5  auto j = std::find_if(v.begin(), v.end(),
6                        [](int i){
7                          return 0 <= i && i <= 42;
8                        });
```

Lambda Function - Pros and Cons

- Pros:

Lambda Function - Pros and Cons

- Pros:
  - Written inline, so:

Lambda Function - Pros and Cons

- Pros:
  - Written inline, so:
  - No need to find the correct scope to define it

Lambda Function - Pros and Cons

- Pros:
  - Written inline, so:
  - No need to find the correct scope to define it
  - No need to find a good name each time

Lambda Function - Pros and Cons

- Pros:
  - Written inline, so:
  - No need to find the correct scope to define it
  - No need to find a good name each time
  - The logic is now local, easy to read and to reason about

Lambda Function - Pros and Cons

- Pros:
  - Written inline, so:
  - No need to find the correct scope to define it
  - No need to find a good name each time
  - The logic is now local, easy to read and to reason about
  - No boilerplate, or at least a minimal one

Lambda Function - Pros and Cons

- Pros:
  - Written inline, so:
  - No need to find the correct scope to define it
  - No need to find a good name each time
  - The logic is now local, easy to read and to reason about
  - No boilerplate, or at least a minimal one
- Cons:

Lambda Function - Pros and Cons

- Pros:
  - Written inline, so:
  - No need to find the correct scope to define it
  - No need to find a good name each time
  - The logic is now local, easy to read and to reason about
  - No boilerplate, or at least a minimal one
- Cons:
  - We lost the generality achieved with functor

Lambda Function - Pros and Cons

- Pros:
  - Written inline, so:
  - No need to find the correct scope to define it
  - No need to find a good name each time
  - The logic is now local, easy to read and to reason about
  - No boilerplate, or at least a minimal one
- Cons:
  - We lost the generality achieved with functor
- Did we lost even the simplest reuse we had with functions?

Reuse a Lambda Function

```
1  void f () {
2    auto between3and7 = [] (int x) {
3      return 3 <= x && x <= 7;
4    };
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 between3and7);
17 }
```

Reuse a Lambda Function

```
1  void f () {
2    auto between3and7 = [] (int x) {
3      return 3 <= x && x <= 7;
4    };
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 between3and7);
17 }
```

Reuse a Lambda Function

```
1  void f () {
2    auto between3and7 = [] (int x) {
3      return 3 <= x && x <= 7;
4    };
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 between3and7);
17 }
```

Reuse a Lambda Function

```
1  void f () {
2    auto between3and7 = [] (int x) {
3      return 3 <= x && x <= 7;
4    };
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 between3and7);
17 }
```

Reuse a Lambda Function

```
1  void f () {
2    auto between3and7 = [] (int x) {
3      return 3 <= x && x <= 7;
4    };
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 between3and7);
17 }
```

Reuse a Lambda Function

```
1  void f () {
2    auto between3and7 = [] (int x) {
3      return 3 <= x && x <= 7;
4    };
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 between3and7);
17 }
```

Reuse a Lambda Function

```
1  void f () {
2    auto between3and7 = [] (int x) {
3      return 3 <= x && x <= 7;
4    };
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 between3and7);
17 }
```

Reuse a Lambda Function

```
1  void f () {
2    auto between3and7 = [] (int x) {
3      return 3 <= x && x <= 7;
4    };
5    std::vector<int> v;
6    // ... fill v with data ...
7    auto i = std::find_if (v.begin(), v.end(),
8                           between3and7);
9    if (i != v.end()) {
10     std::cout << "First item between 3 and 7 is "
11               << *i << '\n';
12   }
13   std::vector<int> v2;
14   std::copy_if (v.begin(), v.end(),
15                 std::back_inserter(v2),
16                 between3and7);
17 }
```

Outline

Return Type of a Lambda Function

- What about declaring the return type?

Return Type of a Lambda Function

- What about declaring the return type?
- Auto deduced from the single (or no) return statement

Return Type of a Lambda Function

- What about declaring the return type?
- Auto deduced from the single (or no) return statement
- Can be mentioned explicitly

Return Type of a Lambda Function

- What about declaring the return type?
- Auto deduced from the single (or no) return statement
- Can be mentioned explicitly
  - For cases where the auto deducing result isn't desired

Return Type of a Lambda Function

- What about declaring the return type?
- Auto deduced from the single (or no) return statement
- Can be mentioned explicitly
  - For cases where the auto deducing result isn't desired
  - For cases with multiple return statements

Return Type of a Lambda Function

- What about declaring the return type?
- Auto deduced from the single (or no) return statement
- Can be mentioned explicitly
  - For cases where the auto deducing result isn't desired
  - For cases with multiple return statements
- Uses "trailing return type" style (available also for regular functions)

Compound Initialization

```
1  class Logger {
2  private:
3    std::ostream& m_stream;
4  public:
5    Logger(std::ostream& stream)
6      : m_stream(stream) {}
7    // Some actually useful methods here
8  };
```

Compound Initialization

```
1  class Logger {
2  private:
3    std::ostream& m_stream;
4  public:
5    Logger(std::ostream& stream)
6      : m_stream(stream) {}
7    // Some actually useful methods here
8  };
```

Compound Initialization

```
1  class Logger {
2  private:
3    std::ostream& m_stream;
4  public:
5    Logger(std::ostream& stream)
6      : m_stream(stream) {}
7    // Some actually useful methods here
8  };
```

Compound Initialization

```
1  class Logger {
2  private:
3    std::ostream& m_stream;
4  public:
5    Logger(std::ostream& stream)
6      : m_stream(stream) {}
7    // Some actually useful methods here
8  };
```

Compound Initialization - 2

```
1  class MyClass {
2  private:
3      std::ofstream m_file;
4      Logger m_logger;
5  public:
6      enum LogTarget { Stdout, File };
7      MyClass(LogTarget target);
8  };
```

Compound Initialization - 2

```
1  class MyClass {
2  private:
3      std::ofstream m_file;
4      Logger m_logger;
5  public:
6      enum LogTarget { Stdout, File };
7      MyClass(LogTarget target);
8  };
```

Compound Initialization - 2

```
1  class MyClass {
2  private:
3      std::ofstream m_file;
4      Logger m_logger;
5  public:
6      enum LogTarget { Stdout, File };
7      MyClass(LogTarget target);
8  };
```

Compound Initialization - 2

```
1  class MyClass {
2  private:
3      std::ofstream m_file;
4      Logger m_logger;
5  public:
6      enum LogTarget { Stdout, File };
7      MyClass(LogTarget target);
8  };
```

Compound Initialization - 2

```
1  class MyClass {
2  private:
3      std::ofstream m_file;
4      Logger m_logger;
5  public:
6      enum LogTarget { Stdout, File };
7      MyClass(LogTarget target);
8  };
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2      : m_logger(
3          [&]() -> std::ostream& {
4                  if (target == Stdout)
5                      return std::cout;
6                  while (!m_file.is_open())
7                      m_file.open(getFilename());
8                  return m_file;
9              }()
10     )
11 {
12 }
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2    : m_logger(
3        [&]() -> std::ostream& {
4                  if (target == Stdout)
5                    return std::cout;
6                  while (!m_file.is_open())
7                      m_file.open(getFilename());
8                  return m_file;
9                }()
10     )
11 {
12 }
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2    : m_logger(
3        [&]() -> std::ostream& {
4                  if (target == Stdout)
5                    return std::cout;
6                  while (!m_file.is_open())
7                      m_file.open(getFilename());
8                  return m_file;
9                }()
10     )
11 {
12 }
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2    : m_logger(
3      [&]() -> std::ostream& {
4                  if (target == Stdout)
5                    return std::cout;
6                  while (!m_file.is_open())
7                    m_file.open(getFilename());
8                  return m_file;
9                }()
10     )
11 {
12 }
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2    : m_logger(
3      [&]() -> std::ostream& {
4                  if (target == Stdout)
5                    return std::cout;
6                  while (!m_file.is_open())
7                      m_file.open(getFilename());
8                  return m_file;
9                }()
10     )
11 {
12 }
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2    : m_logger(
3      [&]() -> std::ostream& {
4                  if (target == Stdout)
5                    return std::cout;
6                  while (!m_file.is_open())
7                      m_file.open(getFilename());
8                  return m_file;
9                }()
10     )
11 {
12 }
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2    : m_logger(
3      [&]() -> std::ostream& {
4                 if (target == Stdout)
5                   return std::cout;
6                 while (!m_file.is_open())
7                     m_file.open(getFilename());
8                 return m_file;
9               }()
10     )
11 {
12 }
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2    : m_logger(
3      [&]() -> std::ostream& {
4                 if (target == Stdout)
5                   return std::cout;
6                 while (!m_file.is_open())
7                     m_file.open(getFilename());
8                 return m_file;
9               }()
10     )
11 {
12 }
```

Compound Initialization - 3

```
1  MyClass::MyClass(LogTarget target)
2    : m_logger(
3      [&]() -> std::ostream& {
4                 if (target == Stdout)
5                   return std::cout;
6                 while (!m_file.is_open())
7                     m_file.open(getFilename());
8                 return m_file;
9               }()
10     )
11 {
12 }
```

The New Things We Just Saw

- Specifying the return type of a lambda function:

The New Things We Just Saw

- Specifying the return type of a lambda function:
  - `[ ] ( ) -> type { }`

The New Things We Just Saw

- Specifying the return type of a lambda function:
  - `[ ] ( ) -> type { }`
  - The parentheses are mandatory in this case

The New Things We Just Saw

- Specifying the return type of a lambda function:
  - `[ ] ( ) -> type { }`
  - The parentheses are mandatory in this case
- IIFE - Immediately-invoked Function Expressions

The New Things We Just Saw

- Specifying the return type of a lambda function:
  - `[ ] ( ) -> type { }`
  - The parentheses are mandatory in this case
- IIFE - Immediately-invoked Function Expressions
  - A term borrowed from JS

The New Things We Just Saw

- Specifying the return type of a lambda function:
  - `[ ] ( ) -> type { }`
  - The parentheses are mandatory in this case
- IIFE - Immediately-invoked Function Expressions
  - A term borrowed from JS
  - Sometimes it's useful to declare a lambda function and immediately invoke it

The New Things We Just Saw

- Specifying the return type of a lambda function:
  - `[ ] ( ) -> type { }`
  - The parentheses are mandatory in this case
- IIFE - Immediately-invoked Function Expressions
  - A term borrowed from JS
  - Sometimes it's useful to declare a lambda function and immediately invoke it
  - Especially useful for initializing things and for simplifying complex decision making logic

The New Things We Just Saw

- Specifying the return type of a lambda function:
  - `[ ] ( ) -> type { }`
  - The parentheses are mandatory in this case
- IIFE - Immediately-invoked Function Expressions
  - A term borrowed from JS
  - Sometimes it's useful to declare a lambda function and immediately invoke it
  - Especially useful for initializing things and for simplifying complex decision making logic
- First example of capturing

The New Things We Just Saw

- Specifying the return type of a lambda function:
  - `[ ] ( ) -> type { }`
  - The parentheses are mandatory in this case
- IIFE - Immediately-invoked Function Expressions
  - A term borrowed from JS
  - Sometimes it's useful to declare a lambda function and immediately invoke it
  - Especially useful for initializing things and for simplifying complex decision making logic
- First example of capturing
  - Let's dive in

Outline

Capturing

- Can we reuse the lambda function more like we did with the manually-written functor?

Capturing

- Can we reuse the lambda function more like we did with the manually-written functor?
- Yes, to some extent

Capturing

- Can we reuse the lambda function more like we did with the manually-written functor?
- Yes, to some extent
- Capturing is the way to give access for the lambda expression to externally declared variables

Capturing - Syntax

- `[x]` - copy `x` in

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` “by-reference”

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` “by-reference”
- `[x, &y]` - copy `x` in and get access to `y` by-ref

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` “by-reference”
- `[x, &y]` - copy `x` in and get access to `y` by-ref
- `[this]` - get access to all the members of the current object

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` "by-reference"
- `[x, &y]` - copy `x` in and get access to `y` by-ref
- `[this]` - get access to all the members of the current object
  - (if used from inside a member function, of course)

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` "by-reference"
- `[x, &y]` - copy `x` in and get access to `y` by-ref
- `[this]` - get access to all the members of the current object
  - (if used from inside a member function, of course)
  - Including private members!

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` "by-reference"
- `[x, &y]` - copy `x` in and get access to `y` by-ref
- `[this]` - get access to all the members of the current object
  - (if used from inside a member function, of course)
  - Including private members!
- `[=]` - copy in everything used inside

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` "by-reference"
- `[x, &y]` - copy `x` in and get access to `y` by-ref
- `[this]` - get access to all the members of the current object
  - (if used from inside a member function, of course)
  - Including private members!
- `[=]` - copy in everything used inside
- `[&]` - access by-ref everything that used inside

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` "by-reference"
- `[x, &y]` - copy `x` in and get access to `y` by-ref
- `[this]` - get access to all the members of the current object
  - (if used from inside a member function, of course)
  - Including private members!
- `[=]` - copy in everything used inside
- `[&]` - access by-ref everything that used inside
- `[=, &x]` - copy in everything, except `x`, which will be used by-ref

Capturing - Syntax

- `[x]` - copy `x` in
- `[&x]` - get access to `x` "by-reference"
- `[x, &y]` - copy `x` in and get access to `y` by-ref
- `[this]` - get access to all the members of the current object
  - (if used from inside a member function, of course)
  - Including private members!
- `[=]` - copy in everything used inside
- `[&]` - access by-ref everything that used inside
- `[=, &x]` - copy in everything, except `x`, which will be used by-ref
- Both "default capturing" options capture also `this`, if available and relevant

Use capturing

```
1  int lo = 3, hi = 7;
2  auto between = [&lo, &hi](int i){
3                              return lo <= i && i <= hi;
4                            };
5
6  auto i = std::find_if(v.begin(), v.end(), between);
7
8  lo = 0; hi = 42;
9  auto j = std::find_if(v.begin(), v.end(), between);
```

Use capturing

```
1  int lo = 3, hi = 7;
2  auto between = [&lo, &hi](int i){
3                              return lo <= i && i <= hi;
4                            };
5
6  auto i = std::find_if(v.begin(), v.end(), between);
7
8  lo = 0; hi = 42;
9  auto j = std::find_if(v.begin(), v.end(), between);
```

Use capturing

```
1  int lo = 3, hi = 7;
2  auto between = [&lo, &hi](int i){
3                              return lo <= i && i <= hi;
4                            };
5
6  auto i = std::find_if(v.begin(), v.end(), between);
7
8  lo = 0; hi = 42;
9  auto j = std::find_if(v.begin(), v.end(), between);
```

Use capturing

```
1  int lo = 3, hi = 7;
2  auto between = [&lo, &hi](int i){
3                              return lo <= i && i <= hi;
4                            };
5
6  auto i = std::find_if(v.begin(), v.end(), between);
7
8  lo = 0; hi = 42;
9  auto j = std::find_if(v.begin(), v.end(), between);
```

Use capturing

```
1  int lo = 3, hi = 7;
2  auto between = [&lo, &hi](int i){
3                              return lo <= i && i <= hi;
4                            };
5
6  auto i = std::find_if(v.begin(), v.end(), between);
7
8  lo = 0; hi = 42;
9  auto j = std::find_if(v.begin(), v.end(), between);
```

Use capturing

```
1  int lo = 3, hi = 7;
2  auto between = [&lo, &hi](int i){
3                              return lo <= i && i <= hi;
4                            };
5
6  auto i = std::find_if(v.begin(), v.end(), between);
7
8  lo = 0; hi = 42;
9  auto j = std::find_if(v.begin(), v.end(), between);
```

Capturing - The Risks

- When capturing something by-ref, lifetime issues must be considered

Capturing - The Risks

- When capturing something by-ref, lifetime issues must be considered
- E.g. returning a lambda from a function or passing it as a callback to something that will stay alive after the current scope

Capturing - Interesting Use-Case - ScopeGuard

- Finally having `finally` in C++

Capturing - Interesting Use-Case - ScopeGuard

- Finally having `finally` in C++
- We all know and love `std::unique_ptr`, `std::shared_ptr`, `std::lock_guard` and more RAII tools from the standard library

Capturing - Interesting Use-Case - ScopeGuard

- Finally having `finally` in C++
- We all know and love `std::unique_ptr`, `std::shared_ptr`, `std::lock_guard` and more RAII tools from the standard library
- Occasionally, we write a class of our own to handle a specific resource / usage in our application

Capturing - Interesting Use-Case - ScopeGuard

- Finally having `finally` in C++
- We all know and love `std::unique_ptr`, `std::shared_ptr`, `std::lock_guard` and more RAII tools from the standard library
- Occasionally, we write a class of our own to handle a specific resource / usage in our application
  - The c-tor creates the resource / takes ownership

Capturing - Interesting Use-Case - ScopeGuard

- Finally having `finally` in C++
- We all know and love `std::unique_ptr`, `std::shared_ptr`, `std::lock_guard` and more RAII tools from the standard library
- Occasionally, we write a class of our own to handle a specific resource / usage in our application
  - The c-tor creates the resource / takes ownership
  - The d-tor destroys it / does any cleanup task needed

Capturing - Interesting Use-Case - ScopeGuard

- Finally having `finally` in C++
- We all know and love `std::unique_ptr`, `std::shared_ptr`, `std::lock_guard` and more RAII tools from the standard library
- Occasionally, we write a class of our own to handle a specific resource / usage in our application
  - The c-tor creates the resource / takes ownership
  - The d-tor destroys it / does any cleanup task needed
- Sometimes we want an ad-hoc tool for use only once in the code

ScopeGuard - Usage

```
1  // ...
2
3  _variant_t vtProp;
4  auto hres =
5    message.Get(L"Message", 0, &vtProp, 0, 0);
6
7  CComSafeArray<uint8_t> arr;
8  arr.Attach(vtProp.parray);
9  // DON'T FORGET TO DETACH AT THE END
10 const auto detach = makeScopeGuard([&arr] () {
11                                     arr.Detach();
12                                   });
13
14 // Continue working; no fear of throwing exceptions
15
```

ScopeGuard - Usage

```
1  // ...
2
3  _variant_t vtProp;
4  auto hres =
5    message.Get(L"Message", 0, &vtProp, 0, 0);
6
7  CComSafeArray<uint8_t> arr;
8  arr.Attach(vtProp.parray);
9  // DON'T FORGET TO DETACH AT THE END
10 const auto detach = makeScopeGuard([&arr] () {
11                                     arr.Detach();
12                                 });
13
14 // Continue working; no fear of throwing exceptions
15
```

ScopeGuard - Usage

```
1  // ...
2
3  _variant_t vtProp;
4  auto hres =
5    message.Get(L"Message", 0, &vtProp, 0, 0);
6
7  CComSafeArray<uint8_t> arr;
8  arr.Attach(vtProp.parray);
9  // DON'T FORGET TO DETACH AT THE END
10 const auto detach = makeScopeGuard([&arr] () {
11                                     arr.Detach();
12                                   });
13
14 // Continue working; no fear of throwing exceptions
15
```

ScopeGuard - Usage

```
1  // ...
2
3  _variant_t vtProp;
4  auto hres =
5    message.Get(L"Message", 0, &vtProp, 0, 0);
6
7  CComSafeArray<uint8_t> arr;
8  arr.Attach(vtProp.parray);
9  // DON'T FORGET TO DETACH AT THE END
10 const auto detach = makeScopeGuard([&arr] () {
11                                     arr.Detach();
12                                 });
13
14 // Continue working; no fear of throwing exceptions
15
```

ScopeGuard - Usage

```
1  // ...
2
3  _variant_t vtProp;
4  auto hres =
5    message.Get(L"Message", 0, &vtProp, 0, 0);
6
7  CComSafeArray<uint8_t> arr;
8  arr.Attach(vtProp.parray);
9  // DON'T FORGET TO DETACH AT THE END
10 const auto detach = makeScopeGuard([&arr] () {
11                                     arr.Detach();
12                                 });
13
14 // Continue working; no fear of throwing exceptions
15
```

ScopeGuard - Another Example

```
1  m_inFWUpdate = true;
2  // MUST RESET THE FLAG AT THE END
3
4  const auto flagGuard = gsl::finally([this] {
5                                  m_inFWUpdate = false;
6                              });
7  // Continue working; status will auto-reset itself
8
```

ScopeGuard - Another Example

```
1  m_inFWUpdate = true;
2  // MUST RESET THE FLAG AT THE END
3
4  const auto flagGuard = gsl::finally([this] {
5                                  m_inFWUpdate = false;
6                              });
7  // Continue working; status will auto-reset itself
8
```

ScopeGuard - Another Example

```
1  m_inFWUpdate = true;
2  // MUST RESET THE FLAG AT THE END
3
4  const auto flagGuard = gsl::finally([this] {
5                                 m_inFWUpdate = false;
6                              });
7  // Continue working; status will auto-reset itself
8
```

ScopeGuard - Another Example

```
1  m_inFWUpdate = true;
2  // MUST RESET THE FLAG AT THE END
3
4  const auto flagGuard = gsl::finally([this] {
5                                 m_inFWUpdate = false;
6                              });
7  // Continue working; status will auto-reset itself
8
```

Outline

The Type of Lambda

- We saved the lambda function till now by using `auto`

The Type of Lambda

- We saved the lambda function till now by using `auto`
- Lambda function is a compiler-generated type that is different for each lambda function

The Type of Lambda

- We saved the lambda function till now by using `auto`
- Lambda function is a compiler-generated type that is different for each lambda function
- Lambda that captures nothing can be "converted" to a function pointer with the matching signature

The Type of Lambda

- We saved the lambda function till now by using `auto`
- Lambda function is a compiler-generated type that is different for each lambda function
- Lambda that captures nothing can be "converted" to a function pointer with the matching signature
- Every lambda can be saved inside `std::function` object with the matching signature

```
1 std::function<void(int)> f = [](int i){
2             std::cout << i << '\n';
3 };
```

The Type of Lambda

- We saved the lambda function till now by using `auto`
- Lambda function is a compiler-generated type that is different for each lambda function
- Lambda that captures nothing can be "converted" to a function pointer with the matching signature
- Every lambda can be saved inside `std::function` object with the matching signature

```
1 std::function<void(int)> f = [](int i){
2             std::cout << i << '\n';
3 };
```

The Type of Lambda

- We saved the lambda function till now by using `auto`
- Lambda function is a compiler-generated type that is different for each lambda function
- Lambda that captures nothing can be "converted" to a function pointer with the matching signature
- Every lambda can be saved inside `std::function` object with the matching signature

```
1 std::function<void(int)> f = [](int i){
2             std::cout << i << '\n';
3 };
```

  - Less recommended due to a non-trivial overhead, but sometimes we have no better choice

Specifiers

- `mutable` allows mutating variables captured by-value

Specifiers

- `mutable` allows mutating variables captured by-value
- `noexcept` can be used if desired

Specifiers

- `mutable` allows mutating variables captured by-value
- `noexcept` can be used if desired
- Both are coming after the `( )`

Outline

Capturing Improvements

- With C++11, capturing move-only types wasn't great

Capturing Improvements

- With C++11, capturing move-only types wasn't great
- Could be captured by-ref, but this isn't always desired

Capturing Improvements

- With C++11, capturing move-only types wasn't great
- Could be captured by-ref, but this isn't always desired
- With C++14, init-capture allowed, enabling usage of move-only types but many other use-cases

Capturing Improvements - Syntax

- `[p = std::move(ptr)]` - moves `ptr` in and call it `p`

Capturing Improvements - Syntax

- `[p = std::move(ptr)]` - moves `ptr` in and call it `p`
- This allows "renaming" in general:

Capturing Improvements - Syntax

- `[p = std::move(ptr)]` - moves `ptr` in and call it `p`
- This allows "renaming" in general:
  - `[x = y]` - copy `y` in but call it `x` inside

Capturing Improvements - Syntax

- `[p = std::move(ptr)]` - moves `ptr` in and call it `p`
- This allows "renaming" in general:
  - `[x = y]` - copy `y` in but call it `x` inside
  - `[&r = p]` - user the name `r` inside as a reference to `p`

Capturing Improvements - Syntax

- `[p = std::move(ptr)]` - moves `ptr` in and call it `p`
- This allows "renaming" in general:
  - `[x = y]` - copy `y` in but call it `x` inside
  - `[&r = p]` - user the name `r` inside as a reference to `p`
- Or even introducing new variables:

Capturing Improvements - Syntax

- `[p = std::move(ptr)]` - moves `ptr` in and call it `p`
- This allows "renaming" in general:
  - `[x = y]` - copy `y` in but call it `x` inside
  - `[&r = p]` - user the name `r` inside as a reference to `p`
- Or even introducing new variables:
  - `[answer = 42]` - create a variable `answer` inside with type `int` and initialize it to `42`

Capturing Improvements - Syntax

- `[p = std::move(ptr)]` - moves `ptr` in and call it `p`
- This allows "renaming" in general:
  - `[x = y]` - copy `y` in but call it `x` inside
  - `[&r = p]` - user the name `r` inside as a reference to `p`
- Or even introducing new variables:
  - `[answer = 42]` - create a variable `answer` inside with type `int` and initialize it to `42`
- ScopeGuard became more usable

Return Type Deduction Improvements

- Return type deduction works with multiple return statements too

Return Type Deduction Improvements

- Return type deduction works with multiple return statements too
- As long as all of them agree on the same type!

Return Type Deduction Improvements

- Return type deduction works with multiple return statements too
- As long as all of them agree on the same type!
- Explicitly stating the return type is still usefult if:

Return Type Deduction Improvements

- Return type deduction works with multiple return statements too
- As long as all of them agree on the same type!
- Explicitly stating the return type is still usefult if:
  - The deduced type isn't what we want (e.g. adding ref)

Return Type Deduction Improvements

- Return type deduction works with multiple return statements too
- As long as all of them agree on the same type!
- Explicitly stating the return type is still usefult if:
  - The deduced type isn't what we want (e.g. adding ref)
  - Deduction fails for differences between return statments (e.g. `int` vs. `double`, `MyObj*` vs. `nullptr`)

Return Type Deduction Improvements

- Return type deduction works with multiple return statements too
- As long as all of them agree on the same type!
- Explicitly stating the return type is still usefult if:
  - The deduced type isn't what we want (e.g. adding ref)
  - Deduction fails for differences between return statments (e.g. `int` vs. `double`, `MyObj*` vs. `nullptr`)
- BTW, regular functions got this feature too for C++14; you can use just `auto` for the return type

Return Type Deduction Improvements

- Return type deduction works with multiple return statements too
- As long as all of them agree on the same type!
- Explicitly stating the return type is still usefult if:
  - The deduced type isn't what we want (e.g. adding ref)
  - Deduction fails for differences between return statments (e.g. `int` vs. `double`, `MyObj*` vs. `nullptr`)
- BTW, regular functions got this feature too for C++14; you can use just `auto` for the return type
  - (but have the function body available for the compiler wherever used, just like template)

Generic Lambda

- With C++14, it's possible to use `auto` for a parameter

Generic Lambda

- With C++14, it's possible to use `auto` for a parameter
- It means the resulted lambda object gets a templated `operator()`

Generic Lambda

- With C++14, it's possible to use `auto` for a parameter
- It means the resulted lambda object gets a templated `operator()`
- Shorter to write :)

Generic Lambda

- With C++14, it's possible to use `auto` for a parameter
- It means the resulted lambda object gets a templated `operator()`
- Shorter to write :)
- `auto size = [](const auto& m){ return m.size(); };`

Generic Lambda

- With C++14, it's possible to use `auto` for a parameter
- It means the resulted lambda object gets a templated `operator()`
- Shorter to write :)
- `auto size = [](const auto& m){ return m.size(); };`
- Very useful as a visitor for `std::variant` (C++17)

Generic Lambda

- With C++14, it's possible to use `auto` for a parameter
- It means the resulted lambda object gets a templated `operator()`
- Shorter to write :)
- `auto size = [](const auto& m){ return m.size(); };`
- Very useful as a visitor for `std::variant` (C++17)
- The conversion to function pointer and `std::function` still works!

Outline

Another Specifier

- `constexpr` can be specified

Another Specifier

- `constexpr` can be specified
- Automatically `constexpr` even if not specified as long as it staisfies all the requirements!

Another Specifier

- `constexpr` can be specified
- Automatically `constexpr` even if not specified as long as it staisfies all the requirements!
- For `constexpr` in general, watching "constexpr ALL the things" is recommended

Another Specifier

- `constexpr` can be specified
- Automatically `constexpr` even if not specified as long as it staisfies all the requirements!
- For `constexpr` in general, watching "constexpr ALL the things" is recommended
  - (by Jason Turner and Ben Deane; CppCon2017 and more)

Another Capturing Option

- Now it's possible to capture the current object by-value

Another Capturing Option

- Now it's possible to capture the current object by-value
- `[*this]`

Outline

Additional Capturing Syntax

- `[=, this]` used to be an error, as capturing `this` is already done by `=`

Additional Capturing Syntax

- `[=, this]` used to be an error, as capturing `this` is already done by `=`
- C++17 added `[=, *this]` option

Additional Capturing Syntax

- `[=, this]` used to be an error, as capturing `this` is already done by `=`
- C++17 added `[=, *this]` option
- So people may want to mention explicitly that they do want `this` by-ref

Additional Capturing Syntax

- `[=, this]` used to be an error, as capturing `this` is already done by `=`
- C++17 added `[=, *this]` option
- So people may want to mention explicitly that they do want `this` by-ref
- C++20 allows `[=, this]` syntax

Easier Access to Generic Lambda Template

- Or: improving diversity and equality :)

Easier Access to Generic Lambda Template

- Or: improving diversity and equality :)
- `[ ] <tparams> ( ) { }`

Easier Access to Generic Lambda Template

- Or: improving diversity and equality :)
- `[ ] <tparams> ( ) { }`
- `auto glambda = []<class T>(T a, auto&& b) { return a < b; };`

Default-Constructible, Usabe in Template Arguments

- Today, for using a lambda function as a comparator for a container, we have to do the following:

```
1 auto comp = [](const auto& l, const auto& r) {
2                         return l.key < r.key;
3 };
4
5 std::set<MyObj, decltype(comp)> set(comp);
```

Default-Constructible, Usabe in Template Arguments

- Today, for using a lambda function as a comparator for a container, we have to do the following:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set(comp);
```

Default-Constructible, Usabe in Template Arguments

- Today, for using a lambda function as a comparator for a container, we have to do the following:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set(comp);
```

Default-Constructible, Usabe in Template Arguments

- Today, for using a lambda function as a comparator for a container, we have to do the following:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set(comp);
```

- C++17 removed the need of repeating (auto-deduction of template arguments for templated classes)

Default-Constructible, Usabe in Template Arguments

- Today, for using a lambda function as a comparator for a container, we have to do the following:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set(comp);
```

- C++17 removed the need of repeating (auto-deduction of template arguments for templated classes)
- It's still inconvenient (C++ Weekly - Ep 94 - Lambdas as Comparators - Jason Turner):

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set set({{MyObj{"example"}}, comp);
```

Default-Constructible, Usabe in Template Arguments

- Today, for using a lambda function as a comparator for a container, we have to do the following:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set(comp);
```

- C++17 removed the need of repeating (auto-deduction of template arguments for templated classes)
- It's still inconvenient (C++ Weekly - Ep 94 - Lambdas as Comparators - Jason Turner):

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set set({{MyObj{"example"}}, comp);
```

Default-Constructible, Usabe in Template Arguments

- Today, for using a lambda function as a comparator for a container, we have to do the following:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set(comp);
```

- C++17 removed the need of repeating (auto-deduction of template arguments for templated classes)
- It's still inconvenient (C++ Weekly - Ep 94 - Lambdas as Comparators - Jason Turner):

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set set({{MyObj{"example"}}, comp);
```

C++20-Style

- With C++20 we can do just:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set;
```

C++20-Style

- With C++20 we can do just:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set;
```

C++20-Style

- With C++20 we can do just:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set;
```

C++20-Style

- With C++20 we can do just:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set;
```

- It's default desctructible!

C++20-Style

- With C++20 we can do just:

```
1  auto comp = [](const auto& l, const auto& r) {
2                          return l.key < r.key;
3  };
4
5  std::set<MyObj, decltype(comp)> set;
```

- It's default desctructible!
- We could even write the lambda directly inside the `decltype` but this is just ugly...

Outline

Summary

- We have seen the various options of lambda function

Summary

- We have seen the various options of lambda function
- Learned the syntax

Summary

- We have seen the various options of lambda function
- Learned the syntax
- Learned the evolution of it over the standard versions

Summary

- We have seen the various options of lambda function
- Learned the syntax
- Learned the evolution of it over the standard versions
- Saw a few interesting usages

Summary

- We have seen the various options of lambda function
- Learned the syntax
- Learned the evolution of it over the standard versions
- Saw a few interesting usages
- Please don't overuse!