Getting the most out of the modern C++ language and standard libraries

Oxford Research Software Engineering

Fergus Cooper ~ Graham Lee ~ Thibault Lestang ~ Martin Robinson ~ Abhishek Dasgupta

C++ has changed a lot

C++ was first standardised as *ISO/IEC 14882:1998* (C++98), and since then:

- C++11 (huge update)
- C++14 (bug fixes, plus a bit)
- C++17 (fairly hefty)
- C++20 (huge update)

A (very quick) overview of C++11

- type inference (`auto`)
- move semantics
- uniform initialisation
- compile time programming (`constexpr`)
- atomic operations
- variadic templates
- lambda expressions
- range-based `for` loops
- `<random>` number generation
- `<chrono>` time library
- *much, much more*

A (very quick) overview of C++14

- more `constexpr`
- improved lambda support
- function return-type deduction
- digit separators
- standard user-defined literals

A (very quick) overview of C++17

- more `constexpr`
- cross-platform filesystem library
- parallel STL algorithms
- structured bindings
- class template argument deduction
- mathematical special functions (`std::riemann_zeta`, ...)
- `std::string_view`
- `std::optional`, `std::any`, and `std::variant`

A (very quick) overview of C++20

- more `constexpr`
- concepts
- modules
- ranges
- coroutines
- 'spaceship' `operator<=>`
- calendar and timezone support
- designated initializers
- `<version>` header
- `std::source_location`

Is this all a bit overwhelming?

How are you supposed to know which features to use and how to use them well?

The trend in C++ has been to add features and then recommend a **reduced subset** of features and some **best practices** that will allow developers to write code that is:

- easier to write
- easier to read
- safer and less prone to errors
- with better performance by default

Is this all a bit overwhelming?

To help navigate the labyrinth of new features and best practices, we have the C++ core guidelines together with a raft of static analysis tools such as clang tidy.

But the best way to learn is to play around with new features, and that's what we're going to do today.

Workshop overview

Today we are going to modernise some C++ code!

The code is broken into a number of checkpoints, and each practical session will get us from one checkpoint to the next.

First, let's:

- Log in to the VM with the details provided
- Grab the latest version:

```
cd ~/RSEConUK2019CppWorkshop
git pull
```

Workshop overview

Next, let's configure, build and run the first checkpoint to ensure everything is working for everyone in the room:

```
cd ~/RSEConUK2019CppWorkshop/
mkdir -p build && cd build
cmake ..
make checkpoint_0
./checkpoint_0
```

Workshop overview

Now, let's have a quick look through the code together.

Use your favourite text editor (`CLion`, `VSCode` and `Emacs` are all installed on the VM), open:

```
~/RSEConUK2019CppWorkshop/checkpoint_0/main.cpp
```

Part 1 — The filesystem library

C++17 added a filesystem library!

- It's very similar to the boost filesystem library on which it's based
- It has intuitive syntax
- It has useful utilities for dealing with files and directories
- It works well across platforms
- Where possible, we should **always** use it when dealing with files

```
namespace fs = std::filesystem;  // names can be a bit verbose

fs::path p = fs::path("base") / "subdir" / "file.ext";
std::cout << p << '\n';

const bool exists = fs::exists(p);

fs::path new_dir = fs::path(".") / "some" / "new" / "dir";
fs::create_directories(new_dir);
```

Part 1a - the `for` loop

Let's say we have a vector. The first way we were probably all taught to loop over the contents of a vector was with an index-based **`for`** loop:

```
std::vector<double> v = {1.0, 2.0, 3.0, 4.0};

for (int i = 0; i < v.size(); ++i) {
   std::cout << v[i] << std::endl;
}
```

(Can you spot a subtle issue here?)

Iterator-based `for` loop

`std::vector` is a container in the Standard Template Library. Every container defines its own **iterators**, so we can also loop over a vector in the following way:

```
std::vector<double> v = {1.0, 2.0, 3.0, 4.0};

for (std::vector<double>::iterator i = v.begin();
     i != v.end(); ++i) {
   std::cout << *i << std::endl;
}
```

This can end up looking quite verbose...

Keyword `auto`

The keyword **`auto`** (C++11) tells the compiler to infer the type of a variable.

```
auto j = 3;     // j is ???
auto x = 1.2;   // x is ???

std::vector v = {1, 2, 3};      // C++17

auto s = v.size();              // s is ???
auto i = v.begin();             // i is ???
auto d = v.end() - v.begin();   // d is ???
```

Keyword `auto`

The keyword **`auto`** (C++11) tells the compiler to infer the type of a variable.

```
auto j = 3;     // j is int
auto x = 1.2;   // x is double

std::vector v = {1, 2, 3};      // C++17

auto s = v.size();              // s is ???
auto i = v.begin();             // i is ???
auto d = v.end() - v.begin();   // d is ???
```

Keyword `auto`

The keyword **`auto`** (C++11) tells the compiler to infer the type of a variable.

```
auto j = 3;     // j is int
auto x = 1.2;   // x is double

std::vector v = {1, 2, 3};      // C++17

auto s = v.size();              // s is std::size_t
auto i = v.begin();             // i is std::vector<int>::iterator
auto d = v.end() - v.begin();   // d is std::ptrdiff_t
```

Keyword `auto`

Use `auto` when you **don't know** or when you **don't care** what the type of the variable is.

Often, we don't care what the type of a variable is - we're happy for the compiler to 'do the right thing'. Replacing it with `auto` *can* make our code easier to read and less prone to errors.

Sometimes *only* the compiler knows the type of a variable - we'll see this later with lambdas.

Iterator-based `for` loop using `auto`

In the context of an iterator-based `for` loop, we can simplify the syntax by using **`auto`** to infer the type returned by `v.begin()`:

```
std::vector<double> v = {1.0, 2.0, 3.0, 4.0};

for (auto i = v.begin(); i != v.end(); ++i) {
   std::cout << *i << std::endl;
}
```

Not only does it look nicer, but it's **easier to read** and **less prone to errors**.

Range-based loops

Range-based loops have the most compact syntax and are often the most intuitive to use. They work with any container that defines `begin` and `end` methods.

```
std::vector<double> v = {1.0, 2.0, 3.0, 4.0};

for (double x: v) {
   std::cout << x << std::endl;
}
```

Range-based loops using `auto`

You can use **auto** here if you don't care about the type...

```
std::vector<double> v = {1.0, 2.0, 3.0, 4.0};

for (auto x: v) {
   std::cout << x << std::endl;  // x is a value
}
```

Range-based loops using `auto` with qualifiers

You can use **auto&** if you want a reference...

```
std::vector<double> v = {1.0, 2.0, 3.0, 4.0};

for (auto& x: v) {
    x += 1.0;  // x is a reference
}
```

Range-based loops using `auto` with qualifiers

You can use `const auto&` if you want a const reference...

```
std::vector<double> v = {1.0, 2.0, 3.0, 4.0};

for (const auto& x: v) {
   std::cout << x << std::endl;  // x is a const reference
}
```

Task 1

We currently have hardcoded paths to three data files. That's not great!

Write some code that will:

- Recursively search through your entire home directory
- Add any data files that end with `"_rse_workshop.dat"` to a `std::vector<fs::path>`
  - hint: strings have a `.ends_with()` method since C++20
- Print out all the data files you found

Part 2 - Moving beyond the `for` loop: STL algorithms

Having just told you about all the great new ways you can write a **`for`** loop, we're going to spend the rest of the workshop trying to convince you to use them as little as possible!

```cpp
std::vector v = {1, 2, 3, 4, 5};

// Option 1
int sum1 = 0;
for (const auto x : v) {
    sum1 += x;
}

// Option 2 (<numeric> header)
const int sum2 = std::accumulate(v.begin(), v.end(), 0);
```

Using the algorithms library

Things about option 2:

- `sum2` is `const`
- the operation has an explicit name (accumulate)
- you are conveying your intent to the compiler
- it's more concise
- it requires another header...

Using the algorithms library

There are algorithms for:

- Adding things up (`std::accumulate`, `std::reduce`)
- Doing "something" to a range (`std::transform`)
- Doing "something" to a range and then adding up (`std::inner_product`, `std::transform_reduce`)
- Sorting (`std::sort`)
- Rotating (`std::rotate`)
- Permuting (`std::next_permutation`)
- Many, many other things

Task 2

We're currently using a `for` loop to calculate the mean and the variance. Yuck!

Replace those for loops with:

- Algorithms!
- First, try `std::accumulate` and `std::inner_product`
- Then, try `std::reduce` and `std::transform_reduce`

Many algorithms allow customisation.

```
template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

Here, `std::sort` is templated over `class Compare` (as well as the iterator type). How can we make use of this customisation point?

There are several ways, but usually the most convenient in modern C++ is the **lambda function**.

The lambda function

You can define a **lambda function** within the current scope:

```
auto empty_lambda = [](){};

auto hello = [](std::string name) {
      std::cout << "hello " << name << std::endl;
    };

hello("world");  // prints "hello world"
```

With a lambda function, you **cannot** specify the type, so we rely on the keyword `auto`.

The lambda function

The square brackets **capture** variables from the outside scope, for example

```
int i = 1;
auto add_i_to_arg = [i](int arg) { return arg + i; };
std::cout << add_i_to_arg(3) << std::endl;  // prints 4
```

This captures `i` by value. To capture by reference use `&`:

```
int i = 1;
auto add_arg_to_i = [&i](int arg) { i += arg; };
add_arg_to_i(3);
std::cout << i << std::endl;  // prints 4
```

The lambda function

You can capture all variables used in the lambda function using either `[=]`, which captures everything by value, or `[&]`, which captures everything by reference:

```
int i = 1;
auto add_i_to_arg = [=](int arg) { return arg + i; };
std::cout << add_i_to_arg(3) << std::endl; // prints 4

auto add_arg_to_i = [&](int arg) { i += arg; };
add_arg_to_i(3);
std::cout << i << std::endl;  // prints 4
```

Using lambdas with algorithms

Let's sort a range largest to smallest rather than smallest to largest...

```
std::vector v = {4.53, 2.38, 3.45, 9.68};

auto sort_greater = [](double x, double y) { return x > y; };
std::sort(v.begin(), v.end(), sort_greater);

std::cout << v.at(0) << '\n';  // prints 9.68
```

Task 3

We're currently using a `for` loop to calculate the skew. Yuck!

Replace that for loop with a call to `std::transform_reduce`

- You'll need to write your own Lambda, which might
  - capture `mean` and `std`
  - take a `double` as parameter
  - the default binary operation for reducing is `std::plus<>()`

Part 4 - other algorithms

There are many algorithms in the standard library!

It is well worth having a good working knowledge of what is available

- Some would be hard to implement efficiently yourself
- Many save you a lot of code
- All make your intent clearer to the compiler
- Most make your intent clearer to other humans

Part 4a - a <chrono> digression

<chrono> tracks time, and is one of the real gems of C++11.

It's also been updated for C++20 with calendar and timezone support!

```
auto t1 = std::chrono::high_resolution_clock::now();
auto t2 = std::chrono::high_resolution_clock::now();

std::chrono::duration<double, std::milli> ms = t2 - t1;
std::cout << ms.count() << " ms\n";
```

Task 4

First:

- Take 5-10 minutes to **read** the list of algorithms
- Click through onto any that you think sound interesting
- Look at some of the examples - get a feel for what is available!

Then:

- Select an appropriate algorithm to calculate the **median** in section 4
- Observe whether your new version is faster than the original. Why (or why not)?

Task 5 - even more algorithms!

There are another two operations on `v` being performed in Section 5. Try to replace those with appropriate algorithms.

Then, spend some time playing around with some other algorithms:

- Either on `v`, or a new container you create yourself
- What does `std::rotate` do?
- Can you think of a use-case for `std::next_permutation`?
- Have you ever written your own `std::max_element`?

Part 6 - parallel <execution>? Out of the box?!

That's right - C++ 17 supports* parallel execution out-of-the-box.

You can add:

```
std::execution::seq         // sequential
std::execution::par         // parallel
std::execution::par_unseq   // parallel and vectorised
std::execution::unseq       // C++20: vectorised
```

as the first parameter to all parallel algorithms.

* gcc 10.x requires linking against a newish version of Intel's TBB library

Task 6

- Go back through all the algorithms you have used, and add execution policies to them
- Think about where it would be appropriate to do so
- Time some examples with and without different policies
  - Are they faster? No? Why not?

Task 7

The final task. Section 7 puts the contents of `v` back out to a file in CSV format using some hard-coded directory and file manipulations, and with a naked `for` loop.

We don't like that much at all.

Drawing on the filesystem library and standard algorithms, have a go at rewriting this section of code.

- Hint: `std::string` has an `operator+` too

That's all, folks

This was only the tiniest tip of the modern C++ iceberg. There are so many great resources available, and here are just a few of them:

- The C++ Core Guidelines
- CppReference
- Compiler explorer
- Podcasts and YouTube series
  - CppCast
  - C++ weekly
- Conferences and meetups
  - C++ On Sea
  - C++ London

Thank you for coming!