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

Oxford RSE @ RSEConUK 2019

Fergus Cooper ~ Graham Lee ~ Thibault Lestang ~ Martin Robinson

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 changes:

```
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/build
cmake ..
make checkpoint_0
./checkpoint_0
```

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

Part 2 - 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
}
```

Moving beyond the `for` loop: STL algorithms

Having just told you about...