Optiver

C++20 and real world C++

COMP6771 Guest Lecture

Videesha Saparamadu 02/08/2022

Who am I?

Videesha Saparamadu

- Senior Software Developer and Technology Educator at Optiver
- BSc CompSci (UNSW)
- Worked in airline software and security engineering before joining Optiver 7 years ago

Why am I here today?

- Optiver is proud to sponsor Advanced C++ Programming at the University of New South Wales.

- At Optiver developers design, build and maintain a world-class automated trading platform, mostly in C++.
- This means:
  - Designing, developing, testing and deploying their own systems.
  - Choosing appropriate algorithms and data-structures.
  - Optimising their systems for low-latency.
  - Employing up-to-date, industry-best practice.

- This course supports these skills and provides a great foundation for working with Optiver
- Optiver donates $500 in prizes for the best performance in COMP 6771.

Agenda

- C++20 Features
  - Modules
  - Coroutines
  - Concepts
  - Ranges
- How we use C++ day to day at Optiver

Modules

- Standardised mechanism for code reuse
- Organise C++ code into logical components

- A module explicitly exports the classes and functions that code outside of the module are allowed to access.

- Other code remains private
- This behaviour will be extended to all c++ library header files (pending compiler support)

```
// helloworld.cppm
export module helloworld;  // module declaration
import <iostream>;         // import declaration

export void hello() {      // export declaration
    std::cout << "Hello world!\n";
}

void goodbye() {
    std::cout << "Goodbye world!\n";
}

//main.cpp
import helloworld;  // import declaration

int main() {
    hello();
    goodbye(); //error not exported
}
```

Modules

Benefits

- Reduces dependency on the pre-processor and header files
- #include file order no longer matters, less error prone, no cyclic dependencies
- Avoids issues with macro leaking
- Faster build times, modules are pre-compiled only once

```
// helloworld.cppm
export module helloworld;  // module declaration
import <iostream>;         // import declaration

export void hello() {      // export declaration
    std::cout << "Hello world!\n";
}

void goodbye() {
    std::cout << "Goodbye world!\n";
}

//main.cpp
import helloworld;  // import declaration

int main() {
    hello();
    goodbye(); //error not exported
}
```

Coroutines

- A coroutine is a function that can suspend execution to be resumed later
- Control is returned to the caller
- The current state of the coroutine is saved to be resumed where it left off

- Keywords co_yield, co_await, co_return

```
generator<int> getNextNumber()
{
    int n = 0;
    while (true)
    {
        co_yield n++;
    }
}

void printNumbers()
{
    std::cout << getNextNumber() << std::endl;
    std::cout << getNextNumber() << std::endl;
}
```

Coroutines

Benefits

- Stackless - Coroutine invocations do not have independent stacks, they allocate data for the coroutine on the heap – efficient memory usage an context switching
- Allow for sequential code that executes asynchronously
- No callbacks, can yield control and resume when necessary

However

- C++ 20 only provides a very low-level api, the generator class used here doesn't yet exist
- Rules of interaction between the caller and the callee are complex
- Can use some third party libraries for example cppcoro

```
generator<int> getNextNumber()
{
    int n = 0;
    while (true)
    {
        co_yield n++;
    }
}

void printNumbers()
{
    std::cout << getNextNumber() << std::endl;
    std::cout << getNextNumber() << std::endl;
}
```

Concepts

- Concepts allow us to specify what is needed from a template argument so this can be checked by the compiler
- Constraints model semantic requirements

- In this example the parameter T is unconstrained, but it won't compile for any type that doesn't have a + operator
- These error messages can be very complex

```
template <typename T>
auto add(T const a, T const b)
{
    return a + b;
}

int main()
{
    std::cout << add(1, 3) << std::endl;
}
```

Concepts

- Add a requires clause

```
template <typename T>
requires std::integral<T>
auto add(T const a, T const b)
{
    return a + b;
}

int main()
{
    std::cout << add(1, 3) << std::endl;

}
```

Concepts

- Creating our own concept

```
template <typename T>
concept Number = std::integral<T> || std::floating_point<T>;

template <typename T, typename U>
requires Number<T> && Number<U>
auto add(T const a, U const b)
{
    return a + b;
}

int main()
{
    std::cout << add(1, 3.1) << std::endl;
}
```

Concepts

Benefits

- Generates meaningful error messages that are much easier to understand
- Clearly documents expectations
- Much easier to use than previous **enable_if** syntax

Ranges

- Ranges are an abstraction of a "collection of items" or "something iterable"
- Containers are ranges, they own their elements
- Views are ranges that that are usually defined on another range
- Views do not own any data beyond their algorithm

- Less error prone than using iterators

```
std::vector v;
std::sort(v.begin(), v.end());
std::ranges::sort(v);
```

Ranges

- Allow us to lazily filter and transform data through a pipeline
- Views are applied when an element is requested, not when the view is created

```
#include <iostream>
#include <ranges>
#include <vector>

int main() {
  std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };

  auto is_even = [](int n) { return n % 2 == 0; };

  auto results = numbers | std::views::filter(is_even)
    | std::views::transform([](int n) { return n++; })
    | std::views::reverse;

  for (auto v: results) {
    std::cout << v << " ";   // Output: 7 3 5
  }
}
```

C++20

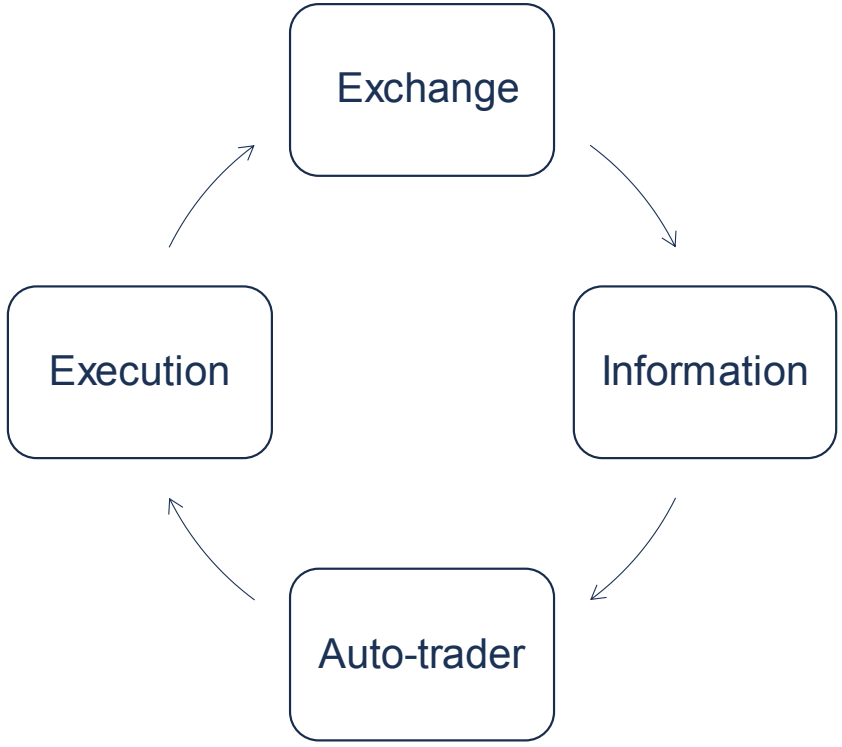
Summary

- Modules
- Coroutines
- Concepts
- Ranges

- Further info see
- Timur Doumler – *How C++20 changes the way we write code*

How do we use C++ at Optiver?

- *Information* flows to us from an exchange.
- Our auto-traders estimate prices and determine if we wish to *execute* any order operations – that is: place, amend and/or delete orders.
- If so, those order operations are sent to the exchange.
- Rinse and repeat!



How do we use C++ at Optiver

Object Oriented Design

- Critically important and included in our interview processs

How do we use C++ at Optiver

C++ Features heavily used

- STL containers and algorithms
  - std::vector, std::find
  - critical to understand the performance implications of your data structure choices
- Smart pointers
  - safety features of std::unique_ptr
- Auto
- Lambda functions
- std::string_view

How do we use C++ at Optiver

C++ Features not used

- Multithreading

How do we use C++ at Optiver

Internal libraries

- Event processing
- More performant data structures

Opportunities at Optiver

Opportunities at Optiver

GRADUATES

- For final year students or recent graduates (< 4 years experience)
- $250K first year package + perks (2023 start)
- Roles: Trader | Researcher | Software Developer | FPGA Developer

INTERNS

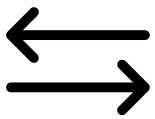
- Pre-penultimate or penultimate year students
- $175K p.a. + super (pro-rated) + perks
- Roles: Trader | Researcher| Software Developer | FPGA Developer

ELIGIBILITY

- AU/NZ citizen, AU permanent resident or able to secure full working rights under the temporary graduate (subclass 485) or skilled-independent visa (subclass 189)

- Applications will open in mid-Feburary

Trading / Quant roles at Optiver



TRADING

- Undertake trading through our auto traders
- Identify profitable opportunities in the market
- Identify trends in market data

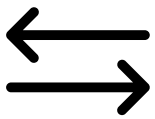
RESEARCH

- Identify trends in market data
- Identify solutions to increase our trading execution success

RISK MANAGER

- Manage market, credit and technology related risks
- Providing risk opinions and views to Trading and Management

Trading / Quant – What we look for



TRADING

- Quantitative skillset
- Lateral thinker
- Have a drive for success
- Interest in trading / financial markets
- Coding experience a +

RESEARCH

- Working with a diverse team
- Self-motivated
- Can communicate ideas & problems
- Coding experience a +

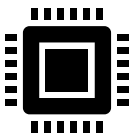
RISK MANAGER

- Interest in financial markets
- An adept communicator (in person & writing)
- VBA, Python, Matlab or other

Technology roles at Optiver

SOFTWARE DEVELOPER

- Design & develop our trading systems
- Maximise speed, reliability & scalability
- C++, C# & some Python



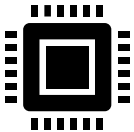
FPGA DEVELOPER

- Accelerate our networks & trading systems
- Explore mechanisms for faster communications
- Work with the fastest devices & platforms

Technology – What we look for

SOFTWARE DEVELOPER

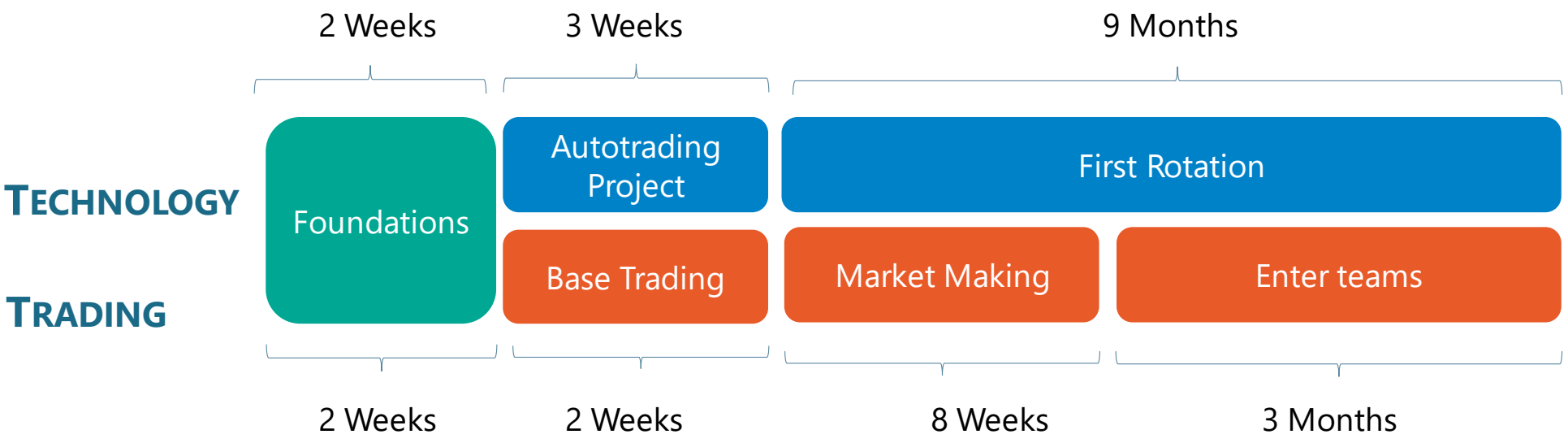
- Low latency, high performance systems
- Personal projects
- Collaboration
- C++, C# or Java



FPGA DEVELOPER

- Hardware passion
- Network protocols / digital design concepts
- Personal projects
- Collaboration
- VHDL and/or Verilog

GRADUATE PROGRAM



Optiver

Questions?