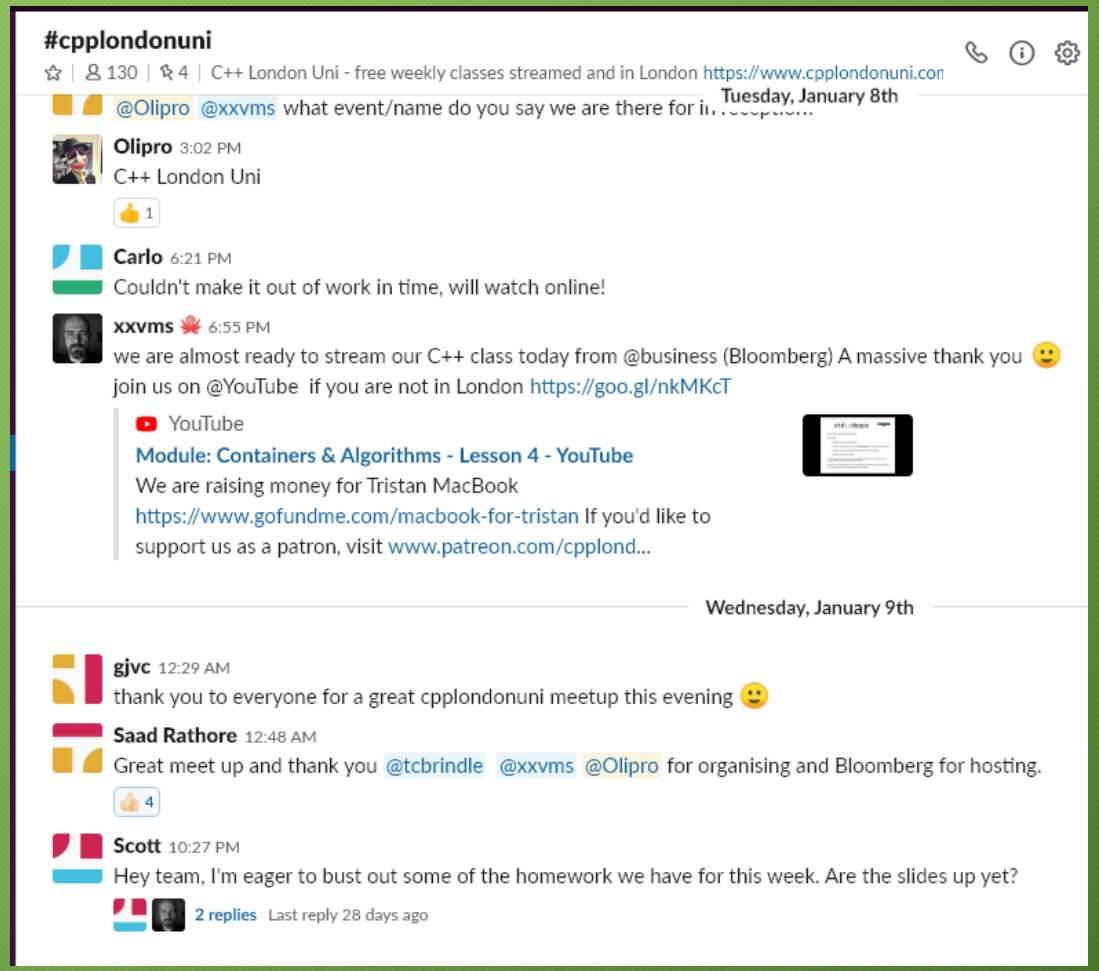
Getting to Know the Standard Library Session 7

Alex Voronov

Feedback

- We'd love to hear from you!
- The easiest way is via the *CPPLang* Slack organisation. Our chatroom is #cpplondonuni
- If you already use Slack, don't worry, it supports multiple workgroups!
- Go to https://slack.cpp.al to register.



Getting to Know the Standard Library

1. Introduction to unit testing with Catch2
2. Basic containers
   - std::vector
   - std::string
3. Basics of the standard-library algorithms
4. Associative containers
   - std::map and std::unordered_map
   - std::set and std::unordered_set
   - Associative containers with custom types
5. **std::function, structured bindings and a few more algorithms**

What I planned in the beginning (a slide from the first session =)

C++ London Uni

1. Introduction to unit testing with Catch2
2. Basic containers
   - std::vector
   - std::string
3. Lambda functions and std::function
4. Associative containers
   - std::map and std::unordered_map
   - std::set and std::unordered_set
   - Associative containers with custom types
   - Set algorithms
5. Overview of algorithms in the standard library

Getting to Know the Standard Library

1. Introduction to unit testing with Catch2
2. Basic containers
   - std::vector
   - std::string
3. Basics of the standard-library algorithms
4. Associative containers
   - std::map and std::unordered_map
   - std::set and std::unordered_set
   - Associative containers with custom types
5. **std::function, structured bindings and a few more algorithms**

We write unit tests instead of using std::cout

Catch2 is a single-header unit-testing framework

https://github.com/catchorg/Catch2

Download *catch.hpp* from the project page and include it in your .cpp file

Last time: Custom types in maps and sets

C++ London Uni

```
struct Date {
    int year, month, day;
};

TEST_CASE("date unordered_map") {
    std::unordered_map<Date, std::string> classes{
        {Date{2020, 03, 24}, "First look into the standard library algorithms"},
        {Date{2020, 04, 21}, "Custom types in associative containers"}
    };

    REQUIRE(classes.size() == 2);
    CHECK(classes.find({2020, 03, 24}) != classes.end());
    CHECK(classes.find({2020, 04, 21}) != classes.end());
}
```

Home exercise: Anagrams in a dictionary

C++ London Uni

```
TEST_CASE("find anagrams -- test template") {
    const auto exciting_anagrams = find_anagrams(read_all_words());
    REQUIRE_FALSE(exciting_anagrams.most_anagrams.empty());
    REQUIRE_FALSE(exciting_anagrams.longest_anagrams.empty());

    // Put proper values here
    REQUIRE(exciting_anagrams.most_anagrams.anagram_count() == 0);
    REQUIRE(exciting_anagrams.longest_anagrams.word_length() == 0);

    // Put proper values here
    CHECK(exciting_anagrams.most_anagrams.word_sets == std::set<WordSet>{});
    CHECK(exciting_anagrams.longest_anagrams.word_sets == std::set<WordSet>{});
}
```

```cpp
using WordSet = std::set<std::string>;

struct MostAnagrams {
    bool empty() const {
        return word_sets.empty() || word_sets.begin()->empty();
    }
    size_t anagram_count() const { return word_sets.begin()->size(); };
    std::set<WordSet> word_sets; // populate this
};
struct LongestAnagrams {
    bool empty() const {
        return word_sets.empty() || word_sets.begin()->empty();
    }
    size_t word_length() const { return word_sets.begin()->begin()->size(); }
    std::set<WordSet> word_sets; // populate this
};

struct ExcitingAnagrams {
    MostAnagrams most_anagrams{};
    LongestAnagrams longest_anagrams{};
};

ExcitingAnagrams find_anagrams(const std::vector<std::string> &words);
```

Home exercise: Getting the anagrams

```
TEST_CASE("find anagrams -- test template") {
    const auto exciting_anagrams = find_anagrams(read_all_words());
    REQUIRE_FALSE(exciting_anagrams.most_anagrams.empty());
    REQUIRE_FALSE(exciting_anagrams.longest_anagrams.empty());

    // Put proper values here
    REQUIRE(exciting_anagrams.most_anagrams.anagram_count() == 0);
    REQUIRE(exciting_anagrams.longest_anagrams.word_length() == 0);

    // Put proper values here
    CHECK(exciting_anagrams.most_anagrams.word_sets == std::set<WordSet>{});
    CHECK(exciting_anagrams.longest_anagrams.word_sets == std::set<WordSet>{});
}
```

Home exercise: The answer

```
TEST_CASE("find anagrams -- the answer") {
    auto response = find_anagrams(read_all_words());
    REQUIRE_FALSE(response.most_anagrams.empty());
    REQUIRE_FALSE(response.longest_anagrams.empty());

    REQUIRE(response.most_anagrams.anagram_count() == 15);
    REQUIRE(response.longest_anagrams.word_length() == 22);

    CHECK(response.most_anagrams.word_sets == std::set<WordSet>{
        {"alerts", "alters", "artels", "estral", "laster", "lastre", "rastle",
            "ratels", "relast", "resalt", "salter", "slater", "staler", "stelar",
            "talers"}});
    CHECK(response.longest_anagrams.word_sets == std::set<WordSet>{
        {"chlorotrifluoromethane", "trifluorochloromethane"},
        {"cholecystoduodenostomy", "duodenocholecystostomy"},
        {"hydropneumopericardium", "pneumohydropericardium"}});
}
```

```cpp
std::vector<std::string> read_all_words() {
    const std::string filename{"../words_alpha.txt"};
    std::ifstream input(filename);

    std::vector<std::string> result;
    std::string line;
    while (std::getline(input, line)) {
        result.push_back(line);
    }

    return result;
}

TEST_CASE("read all words") {
    auto words = read_all_words();
    REQUIRE_FALSE(words.empty());
    CHECK(words.size() == 370104);
    CHECK(words.front() == "a");
    CHECK(words.back() == "zwitterionic");
}
```

Validating elements in a range

```
bool is_lowercase(const std::string &word) {
    return std::all_of(word.begin(), word.end(), [](unsigned char c) {
        return std::islower(c) && std::isalpha(c);
    });
}

TEST_CASE("read all words -- more checks") {
    auto words = read_all_words();
    REQUIRE_FALSE(words.empty());

    CHECK(std::all_of(words.begin(), words.end(), is_lowercase));
    REQUIRE(std::none_of(words.begin(), words.end(),
                         [](const std::string &word) { return word.empty(); }));
    CHECK(std::any_of(
        words.begin(), words.end(),
        [](const std::string &word) { return word.front() == 'k'; }));
}
```

Empty-range edge case

```
TEST_CASE("range checks on an empty range") {
    std::vector<std::string> words;

    auto never_called = [](const std::string &) { return true; };
    CHECK(std::all_of(words.begin(), words.end(), never_called));
    CHECK(std::none_of(words.begin(), words.end(), never_called));
    CHECK_FALSE(std::any_of(words.begin(), words.end(), never_called));
}
```

Anagram sets: Solution structure

1. Collect all the anagram sets in the dictionary
2. Find anagram sets with the most words
    1. Determine the largest size of the set among the anagram sets
    2. Collect all the sets with such size
3. Find anagram sets with the longest words
    1. Determine the length of the longest word in the anagram sets
    2. Collect all sets with such length of a word
4. Pack everything into the struct

Anagram sets: Solution structure in code

C++ London Uni

```
std::vector<WordSet> get_anagram_sets(const std::vector<std::string> &words);

std::set<WordSet>
find_longest_anagrams(const std::vector<WordSet> &anagram_sets);

std::set<WordSet>
find_anagrams_with_most_words(const std::vector<WordSet> &anagram_sets);

ExcitingAnagrams find_anagrams(const std::vector<std::string> &words) {
    const auto anagram_sets = get_anagram_sets(words);

    return ExcitingAnagrams{
        MostAnagrams{find_anagrams_with_most_words(anagram_sets)},
        LongestAnagrams{find_longest_anagrams(anagram_sets)}};
}
```

Anagram sets: Solution structure

1. **Collect all the anagram sets in the dictionary**
2. Find anagram sets with the most words
    1. Determine the largest size of the set among the anagram sets
    2. Collect all the sets with such size
3. Find anagram sets with the longest words
    1. Determine the length of the longest word in the anagram sets
    2. Collect all sets with such length of a word
4. Pack everything into the struct

Collecting all anagrams in the dictionary

```cpp
std::string sort_letters(std::string word) {
    std::sort(word.begin(), word.end());
    return word;
}

std::vector<WordSet> get_anagram_sets(const std::vector<std::string> &words) {
    std::unordered_map<std::string, std::vector<std::string>> groups;
    for (const std::string &word : words) {
        groups[sort_letters(word)].push_back(word);
    }

    std::vector<WordSet> anagram_sets;
    for (const auto&[key, word_set] : groups) {
        if (word_set.size() > 1) {
            anagram_sets.push_back(WordSet{word_set.begin(), word_set.end()});
        }
    }
    return anagram_sets;
}
```

Collecting all anagrams in the dictionary

```cpp
std::string sort_letters(std::string word) {
    std::sort(word.begin(), word.end());
    return word;
}

std::vector<WordSet> get_anagram_sets(const std::vector<std::string> &words) {
    std::unordered_map<std::string, std::vector<std::string>> groups;
    for (const std::string &word : words) {
        groups[sort_letters(word)].push_back(word);
    }

    std::vector<WordSet> anagram_sets;
    for (const auto&[key, word_set] : groups) {
        if (word_set.size() > 1) {
            anagram_sets.push_back(WordSet{word_set.begin(), word_set.end()});
        }
    }
    return anagram_sets;
}
```

```
TEST_CASE("std::sort") {
    std::vector<int> numbers{15, 76, 26, 39, 87, 9, 111};

    SECTION("comparison with operator<") {
        std::sort(numbers.begin(), numbers.end());
        CHECK(numbers == std::vector{9, 15, 26, 39, 76, 87, 111});
    }

    SECTION("reverse order sorting") {
        std::sort(numbers.begin(), numbers.end(),
                  [](int left, int right) { return left > right; });
        CHECK(numbers == std::vector{111, 87, 76, 39, 26, 15, 9});
    }

    SECTION("custom predicate example") {
        std::sort(numbers.begin(), numbers.end(),
                  [](int left, int right) {
                      return sum_of_digits(left) < sum_of_digits(right);
                  });
        CHECK(numbers == std::vector{111, 15, 26, 9, 39, 76, 87});
    }
}
```

Few words about sorting: std::stable_sort

```
TEST_CASE("std::stable_sort by last digit") {
    std::vector<int> numbers{15, 76, 26, 39, 87, 9, 111};

    auto compare_by_last_digit =
        [](int left, int right) { return left % 10 < right % 10; };

    // Preserves order of equivalent elements
    std::stable_sort(numbers.begin(), numbers.end(), compare_by_last_digit);
    CHECK(numbers == std::vector{111, 15, 76, 26, 87, 39, 9});
}
```

Collecting all anagrams in the dictionary

```cpp
std::string sort_letters(std::string word) {
    std::sort(word.begin(), word.end());
    return word;
}

std::vector<WordSet> get_anagram_sets(const std::vector<std::string> &words) {
    std::unordered_map<std::string, std::vector<std::string>> groups;
    for (const std::string &word : words) {
        groups[sort_letters(word)].push_back(word);
    }

    std::vector<WordSet> anagram_sets;
    for (const auto&[key, word_set] : groups) {
        if (word_set.size() > 1) {
            anagram_sets.push_back(WordSet{word_set.begin(), word_set.end()});
        }
    }
    return anagram_sets;
}
```

```cpp
struct Date {
    int year, month, day;
    bool operator<(const Date &other) const {
        return std::tie(year, month, day) <
               std::tie(other.year, other.month, other.day);
    }
};

TEST_CASE("structured bindings example") {
    std::multimap<Date, std::string> classes{
        {Date{2020, 02, 25}, "Unit testing"},
        {Date{2020, 02, 25}, "std::vector"},
        {Date{2020, 03, 10}, "std::string"},
    };

    // Splitting std::pair by with names better than .first and .second
    const auto&[first_class_date, first_class_topic] = *classes.begin();

    // Splitting the Date struct by value for shorter access names
    const auto[year, month, day] = first_class_date;
    CHECK(year == 2020);
    CHECK(month == 02);
}
```

Structured bindings with arrays

```
TEST_CASE("structured bindings with arrays") {
    const int east[2] = {1, 0};
    auto[x1, y1] = east;
    CHECK(x1 == 1);
    CHECK(y1 == 0);

    std::array<int, 2> another_direction{0, 1};
    auto&[x2, y2] = another_direction;
    CHECK(x2 == 0);
    CHECK(y2 == 1);
    y2 = -1;
    CHECK(another_direction[1] == -1);
}
```

Structured bindings example

C++ London Uni

```
TEST_CASE("multimap range values (from the previous lesson)") {
    std::multimap<Date, std::string> classes{
        {Date{2020, 02, 25}, "Unit testing"},
        {Date{2020, 02, 25}, "std::vector"},
        {Date{2020, 03, 10}, "std::string"},
    };

    auto range = classes.equal_range({2020, 02, 25});
    REQUIRE(std::distance(range.first, range.second) == 2);

    std::set<std::string> values;
    for (auto it = range.first; it != range.second; ++it) {
        values.insert(it->second);
    }

    CHECK(values == std::set<std::string>{"Unit testing", "std::vector"});
}
```

Structured bindings example

```
TEST_CASE("multimap range values with structured bindings") {
    std::multimap<Date, std::string> classes{
        {Date{2020, 02, 25}, "Unit testing"},
        {Date{2020, 02, 25}, "std::vector"},
        {Date{2020, 03, 10}, "std::string"},
    };

    auto[range_begin, range_end] = classes.equal_range(Date{2020, 02, 25});
    REQUIRE(std::distance(range_begin, range_end) == 2);

    std::set<std::string> values;
    for (auto it = range_begin; it != range_end; ++it) {
        values.insert(it->second);
    }

    CHECK(values == std::set<std::string>{"Unit testing", "std::vector"});
}
```

Collecting all anagrams in the dictionary

```cpp
std::string sort_letters(std::string word) {
    std::sort(word.begin(), word.end());
    return word;
}

std::vector<WordSet> get_anagram_sets(const std::vector<std::string> &words) {
    std::unordered_map<std::string, std::vector<std::string>> groups;
    for (const std::string &word : words) {
        groups[sort_letters(word)].push_back(word);
    }

    std::vector<WordSet> anagram_sets;
    for (const auto&[key, word_set] : groups) {
        if (word_set.size() > 1) {
            anagram_sets.push_back(WordSet{word_set.begin(), word_set.end()});
        }
    }
    return anagram_sets;
}
```

Anagram sets: Solution structure

1. Collect all the anagram sets in the dictionary
2. **Find anagram sets with the most words**
    1. Determine the largest size of the set among the anagram sets
    2. Collect all the sets with such size
3. Find anagram sets with the longest words
    1. Determine the length of the longest word in the anagram sets
    2. Collect all sets with such length of a word
4. Pack everything into the struct

Anagram set with the most words

C++ London Uni

```
using ComputeProperty = std::function<size_t(const WordSet &)>;
size_t get_max_property_value(const std::vector<WordSet> &anagram_sets,
                              const ComputeProperty &compute_property);

using AnagramSetPredicate = std::function<bool(const WordSet &)>;
std::set<WordSet> filter_anagram_sets(const std::vector<WordSet> &input_sets,
                                      const AnagramSetPredicate &predicate);

std::set<WordSet>
find_anagrams_with_most_words(const std::vector<WordSet> &anagram_sets) {
    const size_t max_anagram_count = get_max_property_value(
        anagram_sets, [](const WordSet &word_set) { return word_set.size(); });

    auto has_most_anagrams = [max_anagram_count](const WordSet &word_set) {
        return word_set.size() == max_anagram_count;
    };
    return filter_anagram_sets(anagram_sets, has_most_anagrams);
}
```

std::function examples

```
size_t count_words(const std::vector<std::string> &words,
                   const std::function<bool(const std::string &)> &predicate) {
    return std::count_if(words.begin(), words.end(), predicate);
}

TEST_CASE("std::function with a lambda expression") {
    const std::vector<std::string> words = read_all_words();

    auto starts_with_y = [](const std::string &word) {
        return !word.empty() && word.front() == 'y';
    };

    CHECK(count_words(words, starts_with_y) == 1143);
}
```

std::function examples (2)

```
size_t count_words(const std::vector<std::string> &words,
                   const std::function<bool(const std::string &)> &predicate);

bool is_palindrome(const std::string &word) {
    const size_t mid_point = word.size() / 2;
    return std::equal(word.begin(), word.begin() + mid_point,
                      word.rbegin(), word.rbegin() + mid_point);
}

TEST_CASE("std::function with a free function") {
    const std::vector<std::string> words = read_all_words();

    auto starts_with_y = [](const std::string &word) {
        return !word.empty() && word.front() == 'y';
    };

    CHECK(count_words(words, starts_with_y) == 1143);
    CHECK(count_words(words, is_palindrome) == 232);
}
```

std::function examples (3)

```
size_t count_words(const std::vector<std::string> &words,
                   const std::function<bool(const std::string &)> &predicate);

struct HasLengthEqualTo {
    explicit HasLengthEqualTo(size_t length) : length_{length} {};
    HasLengthEqualTo() = delete;
    bool operator()(const std::string &word) const {
        return word.size() == length_;
    }
private:
    size_t length_;
};

TEST_CASE("std::function with a functor") {
    const std::vector<std::string> words = read_all_words();

    CHECK(count_words(words, HasLengthEqualTo{2}) == 427);
    CHECK(count_words(words, HasLengthEqualTo{15}) == 8846);
}
```

Anagram set with the most words

C++ London Uni

```
using ComputeProperty = std::function<size_t(const WordSet &)>;
size_t get_max_property_value(const std::vector<WordSet> &anagram_sets,
                              const ComputeProperty &compute_property);

using AnagramSetPredicate = std::function<bool(const WordSet &)>;
std::set<WordSet> filter_anagram_sets(const std::vector<WordSet> &input_sets,
                                      const AnagramSetPredicate &predicate);

std::set<WordSet>
find_anagrams_with_most_words(const std::vector<WordSet> &anagram_sets) {
    const size_t max_anagram_count = get_max_property_value(
        anagram_sets, [](const WordSet &word_set) { return word_set.size(); });

    auto has_most_anagrams = [max_anagram_count](const WordSet &word_set) {
        return word_set.size() == max_anagram_count;
    };
    return filter_anagram_sets(anagram_sets, has_most_anagrams);
}
```

Functions of other functions

```
size_t get_max_property_value(const std::vector<WordSet> &anagram_sets,
                              const ComputeProperty &compute_property) {
    assert(!anagram_sets.empty());
    auto compare = [&compute_property](const auto &left, const auto &right) {
        return compute_property(left) < compute_property(right);
    };
    return compute_property(
        *std::max_element(anagram_sets.begin(), anagram_sets.end(), compare));
}

std::set<WordSet> filter_anagram_sets(const std::vector<WordSet> &input_sets,
                                      const AnagramSetPredicate &predicate) {
    std::set<WordSet> filtered;
    std::copy_if(input_sets.begin(), input_sets.end(),
                 std::inserter(filtered, filtered.end()), predicate);

    return filtered;
}
```

Anagram sets: Solution structure

1. Collect all the anagram sets in the dictionary
2. Find anagram sets with the most words
    1. Determine the largest size of the set among the anagram sets
    2. Collect all the sets with such size
3. **Find anagram sets with the longest words**
    1. Determine the length of the longest word in the anagram sets
    2. Collect all sets with such length of a word
4. Pack everything into the struct

Anagrams with the longest words

C++ London Uni

```
size_t anagram_length(const WordSet &word_set) {
    assert(!word_set.empty() && !word_set.begin()->empty());
    return word_set.begin()->size();
}

std::set<WordSet>
find_longest_anagrams(const std::vector<WordSet> &anagram_sets) {
    const size_t max_anagram_length =
        get_max_property_value(anagram_sets, anagram_length);

    auto has_max_length = [max_anagram_length](const WordSet &word_set) {
        return anagram_length(word_set) == max_anagram_length;
    };
    return filter_anagram_sets(anagram_sets, has_max_length);
}
```

```cpp
std::set<WordSet>
find_anagrams_with_most_words(const std::vector<WordSet> &anagram_sets) {
    const size_t max_anagram_count = get_max_property_value(
        anagram_sets, [](const WordSet &word_set) { return word_set.size(); });

    auto has_most_anagrams = [max_anagram_count](const WordSet &word_set) {
        return word_set.size() == max_anagram_count;
    };
    return filter_anagram_sets(anagram_sets, has_most_anagrams);
}

std::set<WordSet>
find_longest_anagrams(const std::vector<WordSet> &anagram_sets) {
    const size_t max_anagram_length =
        get_max_property_value(anagram_sets, anagram_length);

    auto has_max_length = [max_anagram_length](const WordSet &word_set) {
        return anagram_length(word_set) == max_anagram_length;
    };
    return filter_anagram_sets(anagram_sets, has_max_length);
}
```

Anagram sets: Solution structure

1. Collect all the anagram sets in the dictionary
2. Find anagram sets with the most words
    1. Determine the largest size of the set among the anagram sets
    2. Collect all the sets with such size
3. Find anagram sets with the longest words
    1. Determine the length of the longest word in the anagram sets
    2. Collect all sets with such length of a word
4. **Pack everything into the struct**

Anagram sets: Packing the result

```
std::vector<WordSet> get_anagram_sets(const std::vector<std::string> &words);

std::set<WordSet>
find_longest_anagrams(const std::vector<WordSet> &anagram_sets);

std::set<WordSet>
find_anagrams_with_most_words(const std::vector<WordSet> &anagram_sets);

ExcitingAnagrams find_anagrams(const std::vector<std::string> &words) {
    const auto anagram_sets = get_anagram_sets(words);

    return ExcitingAnagrams{
        MostAnagrams{find_anagrams_with_most_words(anagram_sets)},
        LongestAnagrams{find_longest_anagrams(anagram_sets)}};
}
```

Lesson Summary

- We know 14 anagrams to the word “alters”
- Use std::function when you want to create a generic function (but don’t overuse it =)
- Use structured bindings to give things more accurate names in more concise way

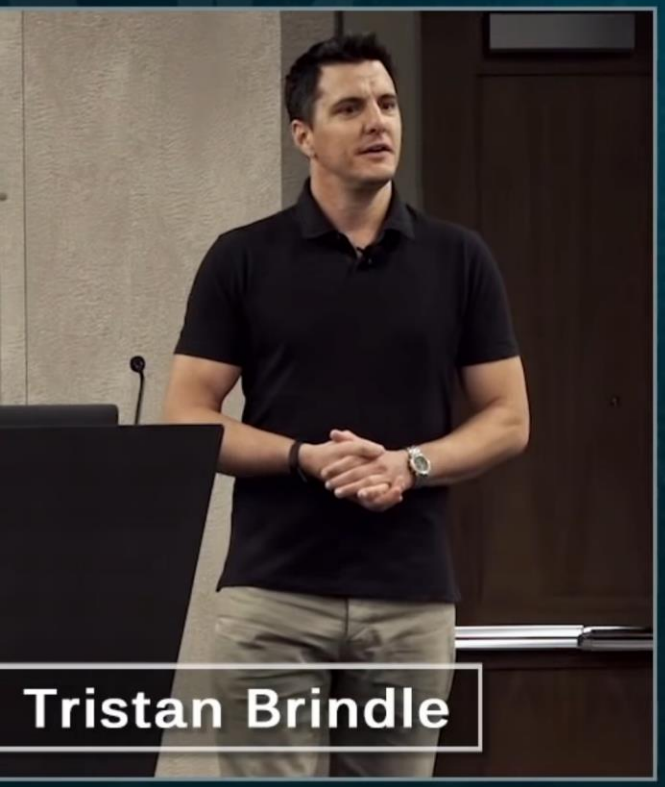
Course highlights

C++ London Uni

What did we accomplish?

- Wrote a lot of unit-tests for C++ code
- Figured out how to use iterators and learned to care about keeping them valid
- Learned about algorithms that can be applied the same way to many containers
- And about lambda expressions that are a convenient way to customise algorithms
- Wrote our own hash functions while fitting custom types into maps and sets
- Talked a lot about palindromes, anagrams and persimmon erasure

Cppcon | 2019
The C++ Conference | cppcon.org



An Overview of Standard Ranges

Video Sponsorship Provided By:
ansatz

AN OVERVIEW OF STANDARD RANGES

CppCon 2019

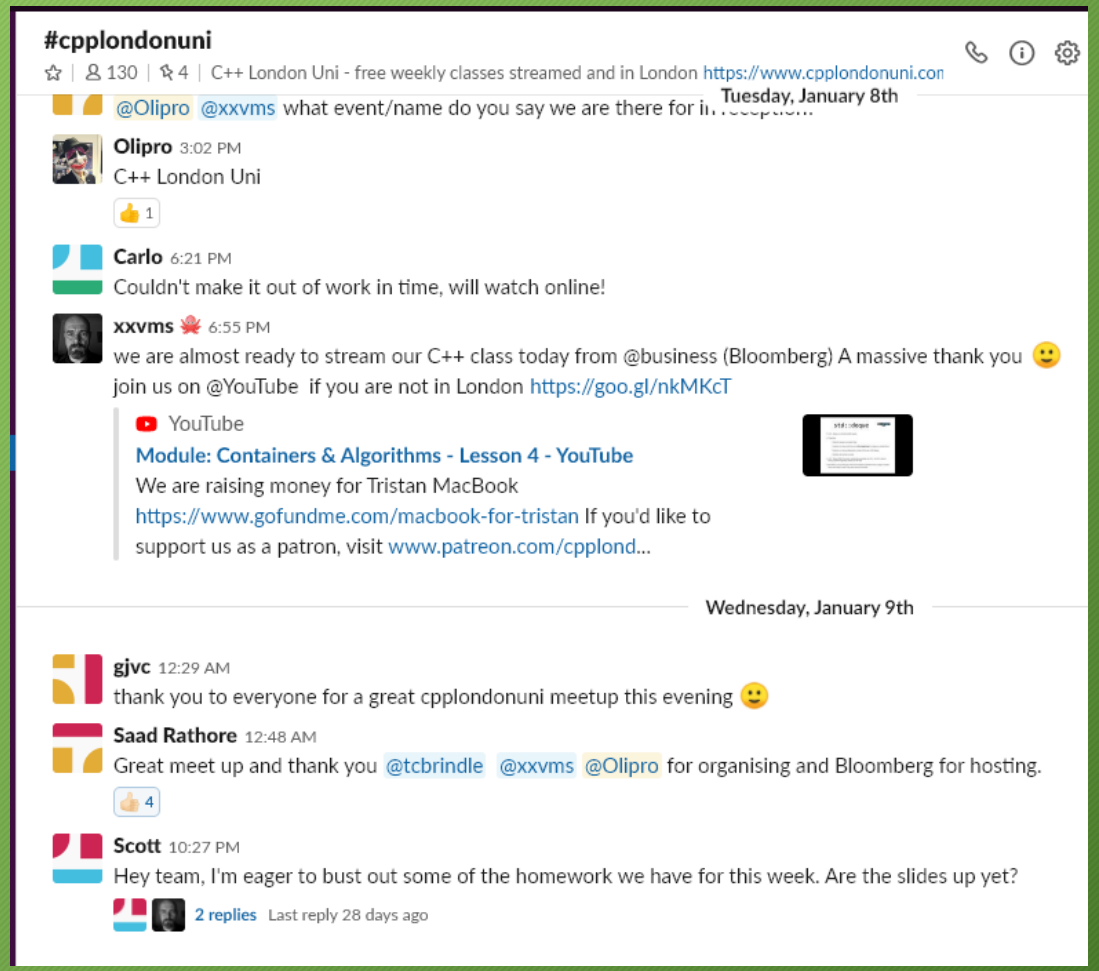
Tristan Brindle

1

https://youtu.be/SYLgG7Q5Zws

Feedback

- We'd love to hear from you!
- The easiest way is via the *CPPLang* Slack organisation. Our chatroom is #cpplondonuni
- If you already use Slack, don't worry, it supports multiple workgroups!
- Go to https://slack.cpp.al to register.



Thank You!

As usual, we will be going to the pub! Support us @ https://patreon.com/CPPLondonUni