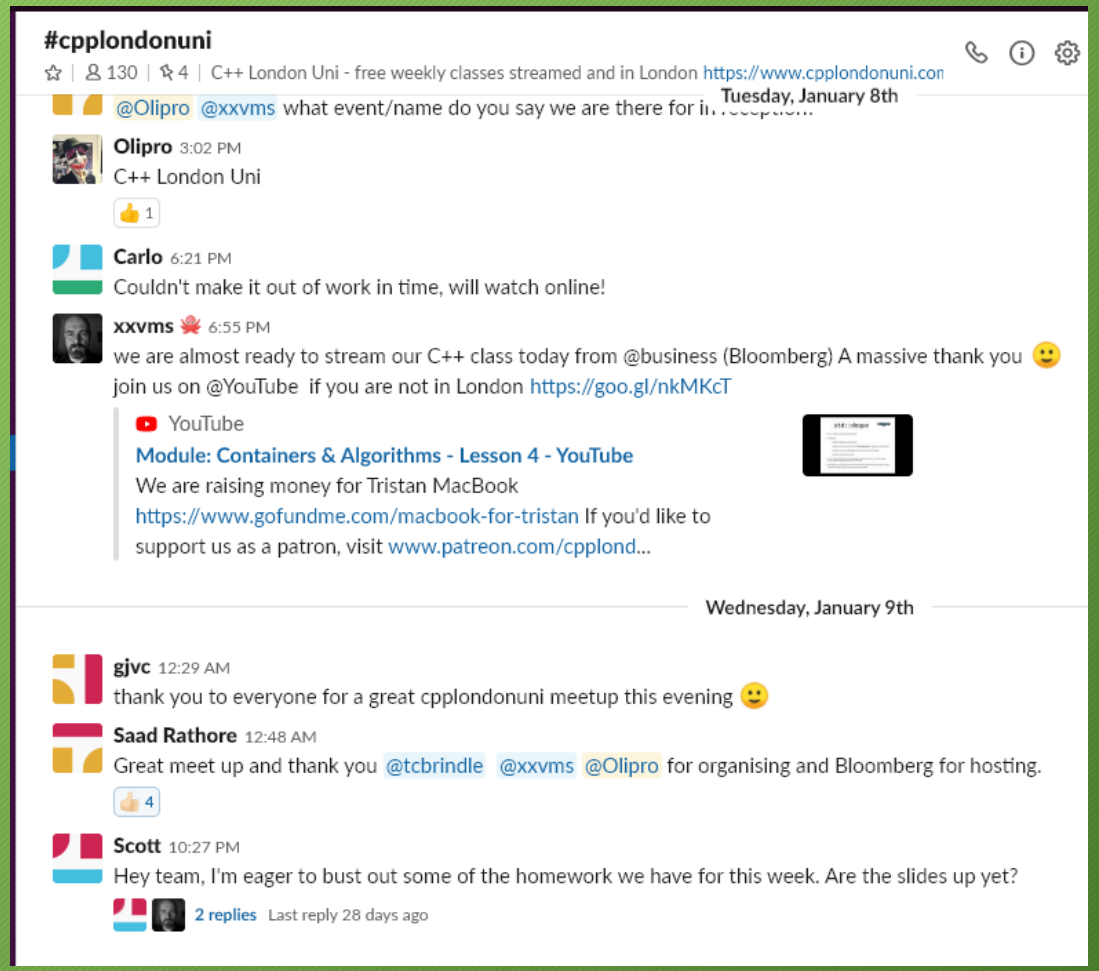
Getting to Know the Standard Library Session 6

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

C++ London Uni

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. More standard-library algorithms

Custom types in maps and sets

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - Composite-key type
  - Practice
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - Practice
- Few words about multimaps and multisets
- Summary
- Home exercise

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

Github spring cleaning

C++ London Uni — Getting to Know the Standard Library

This course is an introduction into C++ standard library containers and algorithms. We talk about the common principles and patterns for the standard library components and go into details for most widely used containers and algorithms.

The course uses small algorithmic problems (for example, checking if a word is a palindrome) to illustrate the uses of containers and algorithms and to discuss the best practices and common mistakes.

Prerequisites: basic knowledge of C++ (types, classes, loops, constructors) or familiarity with any other programming language with interest in C++.

Lessons

1. Quick introduction into unit testing in C++ (PDF slides, YouTube stream)
2. std::vector (PDF slides, YouTube stream)
3. std::string (PDF slides, YouTube stream failed)
4. First look into the standard library algorithms (PDF slides, YouTube stream)
5. Associative containers — sets and maps (PDF slides, YouTube stream)
6. *The next lesson comes 21st April 2020*

https://github.com/CPPLondonUni/course_materials_standard_library_spring_2020

Last time: Associative containers

```cpp
size_t most_frequent_word_count(const std::vector<std::string> &words) {
    if (words.empty()) {
        return 0;
    }
    std::map<std::string, size_t> word_count;
    for (const auto &word : words) {
        ++word_count[word];
    }
    return std::max_element(word_count.cbegin(), word_count.cend(),
                            [](const auto &left, const auto &right) {
                                return left.second < right.second;
                            })->second;
}
```

Home exercise: Route finding

Given a vector of couples of directly connected airports,
find if there is a route with transfers between two particular airports

```
using Route = std::array<std::string, 2>;
bool route_exists(const std::vector<Route> &direct_routes, const Route &query);

TEST_CASE("route exists") {
    std::vector<Route> direct_flights{
        {"LHR", "FRA"}, {"FRA", "RIX"}, {"LHR", "JFK"}, {"GRU", "EZE"}};

    CHECK(route_exists(direct_flights, {"FRA", "RIX"}));
    CHECK(route_exists(direct_flights, {"RIX", "FRA"}));
    CHECK(route_exists(direct_flights, {"RIX", "JFK"}));
    CHECK_FALSE(route_exists(direct_flights, {"EZE", "JFK"}));
    CHECK_FALSE(route_exists(direct_flights, {"TXL", "CDG"}));
}
```

```cpp
bool route_exists(const std::vector<Route> &direct_routes, const Route &query) {
    std::unordered_map<std::string, std::unordered_set<std::string>> connected;
    for (const auto &connection : direct_routes) {
        connected[connection[0]].insert(connection[1]);
        connected[connection[1]].insert(connection[0]);
    }
    std::unordered_set<std::string> visited;
    std::unordered_set<std::string> to_visit = connected[query[0]];
    while (!to_visit.empty()) {
        auto under_review = *to_visit.begin();
        if (under_review == query[1]) {
            return true;
        }
        to_visit.erase(to_visit.begin());
        visited.insert(under_review);
        for (const auto &destination : connected.at(under_review)) {
            if (visited.find(destination) == visited.end()) {
                to_visit.insert(destination);
            }
        }
    }
    return false;
}
```

Custom types in maps and sets

C++ London Uni

- Recap and updates
- **Custom key types in `std::map`**
  - Introduction and a simple example
  - Composite-key type
  - Practice
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - Practice
- Few words about multimaps and multisets
- Summary
- Home exercise

Problem: Using custom type as a key

C++ London Uni

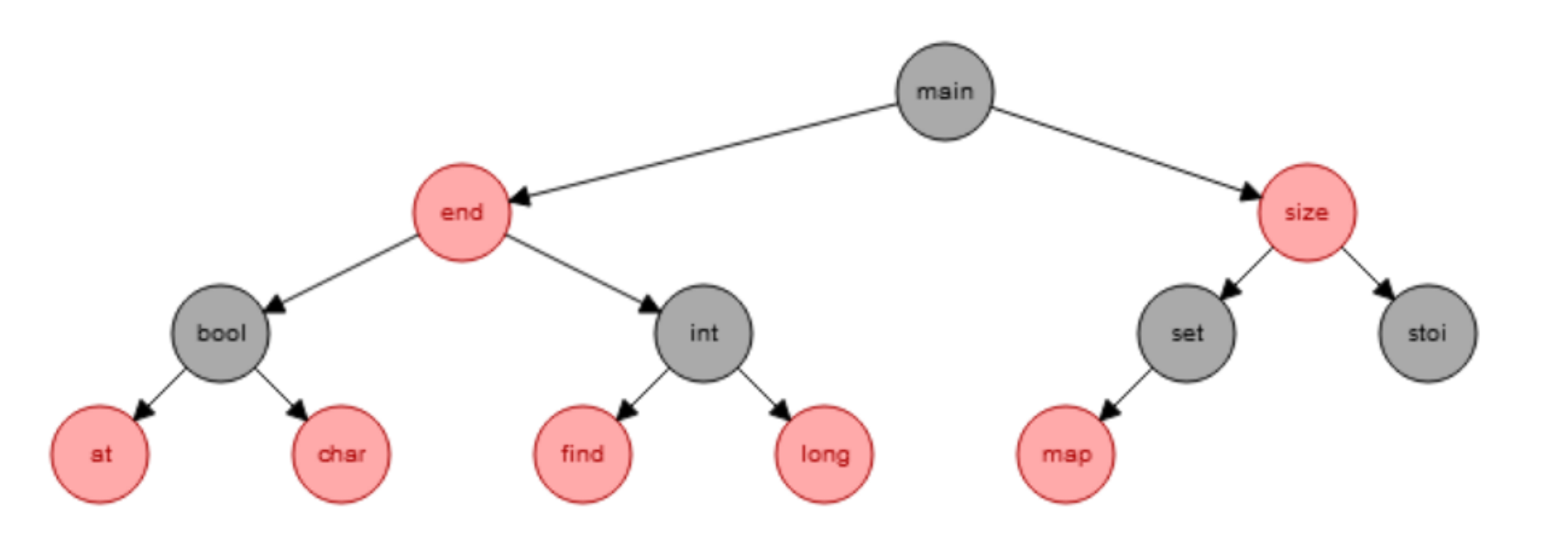
```
struct UserID {
    unsigned long long value;
};


TEST_CASE("UserID map") {
    std::map<UserID, std::string> user_names;
    user_names.insert({UserID{1}, "Alex Voronov"});
    user_names.insert({UserID{2}, "Tom Breza"});
    CHECK(user_names.size() == 2);
    REQUIRE(user_names.find(UserID{1}) != user_names.end());
    CHECK(user_names.at(UserID{1}) == "Alex Voronov");
    REQUIRE(user_names.find(UserID{2}) != user_names.end());
    CHECK(user_names.at(UserID{2}) == "Tom Breza");
}
```

Underlying data structure

Balanced binary search tree requires:

- comparison by less-than
- … and that's it!



https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Requirement: Comparison by less-than

```
struct UserID {
    unsigned long long value;
    bool operator<(const UserID &other) const { return value < other.value; }
};

TEST_CASE("UserID map") {
    CHECK(UserID{1} < UserID{2});
    CHECK_FALSE(UserID{2} < UserID{1});
    CHECK_FALSE(UserID{1} < UserID{1});

    std::map<UserID, std::string> user_names;
    user_names.insert({UserID{1}, "Alex Voronov"});
    user_names.insert({UserID{2}, "Tom Breza"});
    CHECK(user_names.size() == 2);
    REQUIRE(user_names.find(UserID{1}) != user_names.end());
    CHECK(user_names.at(UserID{1}) == "Alex Voronov");
    REQUIRE(user_names.find(UserID{2}) != user_names.end());
    CHECK(user_names.at(UserID{2}) == "Tom Breza");
}
```

```cpp
struct UserIDCompareLess {
    bool operator()(const UserID &left, const UserID &right) const {
        return left.value < right.value;
    }
};

TEST_CASE("Another way: UserID map with comparator type") {
    UserIDCompareLess compare_less;
    CHECK(compare_less(UserID{1}, UserID{2}));
    CHECK_FALSE(compare_less(UserID{2}, UserID{1}));
    CHECK_FALSE(compare_less(UserID{1}, UserID{1}));

    std::map<UserID, std::string, UserIDCompareLess> user_names;
    user_names.insert({UserID{1}, "Alex Voronov"});
    user_names.insert({UserID{2}, "Tom Breza"});
    CHECK(user_names.size() == 2);
    REQUIRE(user_names.find(UserID{1}) != user_names.end());
    CHECK(user_names.at(UserID{1}) == "Alex Voronov");
    REQUIRE(user_names.find(UserID{2}) != user_names.end());
    CHECK(user_names.at(UserID{2}) == "Tom Breza");
}
```

```cpp
struct UserIDCompareLess {
    bool operator()(const UserID &left, const UserID &right) const {
        return left.value < right.value;
    }
};

struct UserIDCompareGreater {
    bool operator()(const UserID &left, const UserID &right) const {
        return left.value > right.value;
    }
};

TEST_CASE("Comparators and ordered containers") {
    std::set<UserID, UserIDCompareLess> set_ordered_by_less{
        UserID{1}, UserID{2}};
    std::set<UserID, UserIDCompareGreater> set_ordered_by_greater{
        UserID{1}, UserID{2}};

    REQUIRE(set_ordered_by_less.size() == 2);
    REQUIRE(set_ordered_by_greater.size() == 2);
    CHECK(set_ordered_by_less.begin()->value == 1);
    CHECK(set_ordered_by_greater.begin()->value == 2);
}
```

Custom types in maps and sets

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - **Composite-key type**
  - Practice
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - Practice
- Few words about multimaps and multisets
- Summary
- Home exercise

What do we do with more complex types?

C++ London Uni

```
struct PlayerID {
    unsigned int year;
    std::string team;
    unsigned int player_number;
};

TEST_CASE("PlayerID map") {
    std::map<PlayerID, std::string> players;
    players.insert({{2019, "Liverpool", 14}, "Jordan Henderson"});
    players.insert({{2019, "Tottenham Hotspur", 1}, "Hugo Lloris"});

    REQUIRE(players.size() == 2);
    CHECK(players.at({2019, "Liverpool", 14}) == "Jordan Henderson");
    CHECK(players.at({2019, "Tottenham Hotspur", 1}) == "Hugo Lloris");
}
```

What do we do with more complex types? (2)

C++ London Uni

```
struct PlayerID {
    unsigned int year;
    std::string team;
    unsigned int player_number;

    bool operator<(const PlayerID &other) const {
        if (year != other.year) {
            return year < other.year;
        } else if (player_number != other.player_number) {
            return player_number < other.player_number;
        } else {
            return team < other.team;
        }
    }
};
```

What do we do with more complex types? (3)

C++ London Uni

```
struct PlayerID {
    unsigned int year;
    std::string team;
    unsigned int player_number;

    bool operator<(const PlayerID &other) const {
        // std::tie returns an std::tuple of references.
        // And the tuple type already has operator< defined
        // by lexicographical comparison of tuple elements
        return std::tie(year, player_number, team) <
               std::tie(other.year, other.player_number, other.team);
    }
};
```

Ordered maps and sets: Conclusion

`std::map` and `std::set` require less-than comparison to be defined for the key type

- Two ways to define less-than comparison:
  - Implement `operator<(…)` for the type
  - Create a custom functor class and specify it as map/set-type parameter

- Prefer `std::tie` when comparing multiple member variables
- Implementation of the comparison affects the order of elements in the container

Custom types in maps and sets

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - Composite-key type
  - **Practice**
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - Practice
- Few words about multimaps and multisets
- Summary
- Home exercise

Practice: Date map

Add missing properties of the `Date` type to make the example test compile and work

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

TEST_CASE("date map") {
    std::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());
}
```

Solution: Date map

```
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("date map") {
    std::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());
}
```

Custom types in maps and sets

C++ London Uni

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - Composite-key type
  - Practice
- **Custom key types in `std::unordered_map`**
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - Practice
- Few words about multimaps and multisets
- Summary
- Home exercise

Unordered map: Back to basics

C++ London Uni

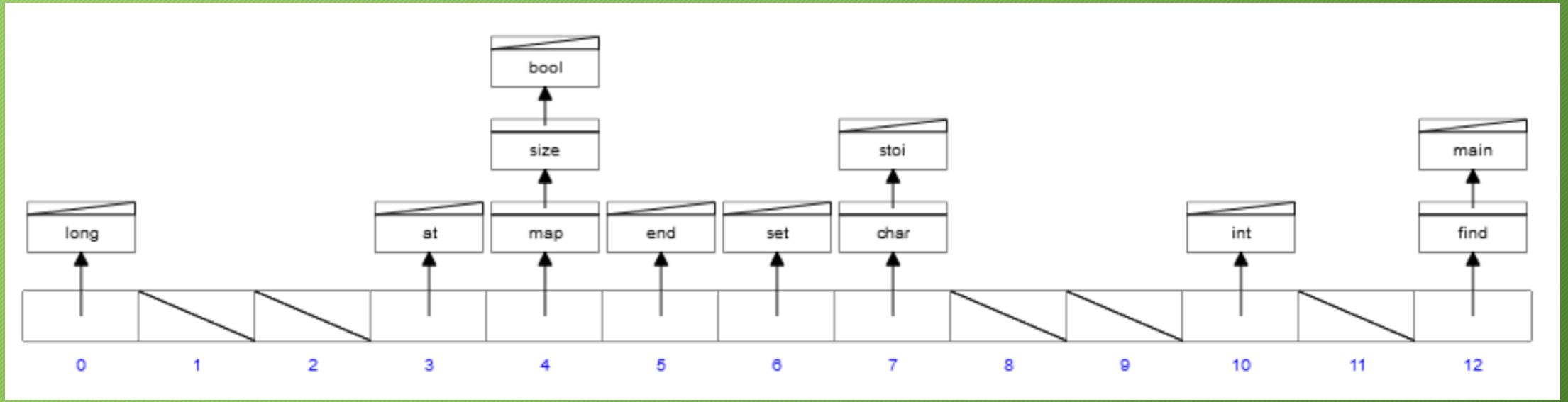
```cpp
struct UserID {
    unsigned long long value;
};


TEST_CASE("UserID unordered_map") {
    std::unordered_map<UserID, std::string> user_names;
    user_names.insert({UserID{1}, "Alex Voronov"});
    user_names.insert({UserID{2}, "Tom Breza"});
    CHECK(user_names.size() == 2);
    REQUIRE(user_names.find(UserID{1}) != user_names.end());
    CHECK(user_names.at(UserID{1}) == "Alex Voronov");
    REQUIRE(user_names.find(UserID{2}) != user_names.end());
    CHECK(user_names.at(UserID{2}) == "Tom Breza");
}
```

Underlying data structure

Hash table requires:

- Hashing function to distribute elements among buckets
- Equality comparison for elements inside the same bucket



https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

```cpp
struct UserID {
    unsigned long long value;
    bool operator==(const UserID &other) const { return value == other.value; }
};
namespace std {
template<>
struct hash<UserID> {
    size_t operator()(const UserID &id) const {
        return std::hash<unsigned long long>{}(id.value);
    }
};
} // namespace std

TEST_CASE("UserID unordered map") {
    std::unordered_map<UserID, std::string> user_names;
    user_names.insert({UserID{1}, "Alex Voronov"});
    user_names.insert({UserID{2}, "Tom Breza"});
    CHECK(user_names.size() == 2);
    REQUIRE(user_names.find(UserID{1}) != user_names.end());
    CHECK(user_names.at(UserID{1}) == "Alex Voronov");
    REQUIRE(user_names.find(UserID{2}) != user_names.end());
    CHECK(user_names.at(UserID{2}) == "Tom Breza");
}
```

```cpp
struct UserIDHasher {
    size_t operator()(const UserID &id) const {
        return std::hash<unsigned long long>{}(id.value);
    }
};

struct UserIDCompareEqual {
    bool operator()(const UserID &left, const UserID &right) const {
        return left.value == right.value;
    }
};

TEST_CASE("UserID unordered map with custom parameters") {
    std::unordered_map<UserID, std::string, UserIDHasher, UserIDCompareEqual>
        user_names;
    user_names.insert({UserID{1}, "Alex Voronov"});
    user_names.insert({UserID{2}, "Tom Breza"});
    CHECK(user_names.size() == 2);
    REQUIRE(user_names.find(UserID{1}) != user_names.end());
    CHECK(user_names.at(UserID{1}) == "Alex Voronov");
    REQUIRE(user_names.find(UserID{2}) != user_names.end());
    CHECK(user_names.at(UserID{2}) == "Tom Breza");
}
```

```cpp
struct UserIDPoorHasher {
    size_t operator()(const UserID &) const { return 0; }
};

TEST_CASE("Hasher comparison") {
    std::vector<UserID> ids{UserID{1}, UserID{2}, UserID{3}, UserID{4}};
    std::unordered_set<UserID, UserIDHasher> well_hashed(
        ids.begin(), ids.end());
    std::unordered_set<UserID, UserIDPoorHasher> poorly_hashed(
        ids.begin(), ids.end());
    REQUIRE(well_hashed.size() == 4);
    REQUIRE(poorly_hashed.size() == 4);

    auto check_poorly_hashed = [&poorly_hashed](const UserID &id) {
        return poorly_hashed.bucket(UserID{1}) == poorly_hashed.bucket(id);
    };
    auto check_well_hashed = [&well_hashed](const UserID &id) {
        return well_hashed.bucket(UserID{1}) == well_hashed.bucket(id);
    };
    CHECK(std::all_of(ids.begin(), ids.end(), check_poorly_hashed));
    CHECK_FALSE(std::all_of(ids.begin(), ids.end(), check_well_hashed));
}
```

Custom types in maps and sets

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - Composite-key type
  - Practice
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - **Composite-key type and combining hashes**
  - Practice
- Few words about multimaps and multisets
- Summary
- Home exercise

What do we do with more complex types?

C++ London Uni

```
TEST_CASE("PlayerID unordered map") {
    std::unordered_map<PlayerID, std::string> players;
    players.insert({{2019, "Liverpool", 14}, "Jordan Henderson"});
    players.insert({{2019, "Tottenham Hotspur", 1}, "Hugo Lloris"});

    REQUIRE(players.size() == 2);
    CHECK(players.at({2019, "Liverpool", 14}) == "Jordan Henderson");
    CHECK(players.at({2019, "Tottenham Hotspur", 1}) == "Hugo Lloris");

    std::hash<PlayerID> hasher{};
    CHECK(hasher({2019, "Liverpool", 14}) !=
          hasher({2019, "Tottenham Hotspur", 1}));
}
```

```cpp
struct PlayerID {
    unsigned int year;
    std::string team;
    unsigned int player_number;

    bool operator==(const PlayerID &other) const {
        return std::tie(year, player_number, team) ==
               std::tie(other.year, other.player_number, other.team);
    }
};

size_t simple_hash_combine(size_t seed, size_t hash) { return (seed << 1) ^ hash; }

namespace std {
template<>
struct hash<PlayerID> {
    size_t operator()(const PlayerID &id) const {
        size_t hash = 0;
        hash = simple_hash_combine(hash, std::hash<unsigned int>{}(id.year));
        hash = simple_hash_combine(hash, std::hash<std::string>{}(id.team));
        hash = simple_hash_combine(hash, std::hash<unsigned int>{}(id.player_number));
        return hash;
    }
};
} // namespace std
```

Revision: Bitwise shift and xor

C++ London Uni

```
TEST_CASE("bitwise shift and xor") {
    unsigned int all_ones = 0b11111111'11111111'11111111'11111111;
    CHECK((all_ones << 2) == 0b11111111'11111111'11111111'11111100);
    CHECK((all_ones >> 5) == 0b00000111'11111111'11111111'11111111);
    CHECK((all_ones >> 31) == 1);
    CHECK((all_ones << 31) == 0b10000000'00000000'00000000'00000000);

    unsigned int millennium_epoch = 0b00111000'01101101'01000011'10000000;
    CHECK((millennium_epoch >> 8) == 0b00000000'00111000'01101101'01000011);
    CHECK((millennium_epoch << 4) == 0b10000110'11010100'00111000'00000000);

    CHECK((0u ^ 0u) == 0u);
    CHECK((1u ^ 0u) == 1u);
    CHECK((0u ^ 1u) == 1u);
    CHECK((1u ^ 1u) == 0u);
    CHECK((0b0101u ^ 0b1100u) == 0b1001u);
    CHECK((0b0101u ^ 0b0101u) == 0u);
}
```

```cpp
struct PlayerID {
    unsigned int year;
    std::string team;
    unsigned int player_number;

    bool operator==(const PlayerID &other) const {
        return std::tie(year, player_number, team) ==
               std::tie(other.year, other.player_number, other.team);
    }
};

size_t simple_hash_combine(size_t seed, size_t hash) { return (seed << 1) ^ hash; }

namespace std {
template<>
struct hash<PlayerID> {
    size_t operator()(const PlayerID &id) const {
        size_t hash = 0;
        hash = simple_hash_combine(hash, std::hash<unsigned int>{}(id.year));
        hash = simple_hash_combine(hash, std::hash<std::string>{}(id.team));
        hash = simple_hash_combine(hash, std::hash<unsigned int>{}(id.player_number));
        return hash;
    }
};
} // namespace std
```

boost::hash_combine

Support functions (Boost extension).

1. `template<typename T> void hash_combine(size_t & seed, T const& v);`

   Called repeatedly to incrementally create a hash value from several variables.

   Effects: `seed ^= hash value(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);`

https://www.boost.org/doc/libs/1_72_0/doc/html/hash/reference.html#boost.hash_combine

Unordered maps and sets: Conclusion

`std::unordered_map` and `std::unordered_set` require *equality comparison* and a *hash function* to be defined for the key type

- Both can be provided as
  - type properties,
  - functor-type parameters of the container
- To provide a hash function for a type, you need to implement `std::hash` struct template specialization for your type (Sounds scary but in practice is pretty straightforward)
- Good hash function is critical for the container performance

Custom types in maps and sets

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - Composite-key type
  - Practice
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - **Practice**
- Few words about multimaps and multisets
- Summary
- Home exercise

Practice: Unordered date map

Add missing properties of the Date type to make the example test compile and work

```
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());
}
```

```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);
    }
};

namespace std {
template<>
struct hash<Date> {
    bool operator()(const Date &date) const {
        size_t result = 0;
        result = simple_hash_combine(result, std::hash<int>{}(date.year));
        result = simple_hash_combine(result, std::hash<int>{}(date.month));
        result = simple_hash_combine(result, std::hash<int>{}(date.day));
        return result;
    }
};
} // namespace std
```

Custom types in maps and sets

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - Composite-key type
  - Practice
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - Practice
- **Few words about multimaps and multisets**
- Summary
- Home exercise

Few words about multimaps

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

    REQUIRE(classes.size() == 3);
    CHECK(classes.count({2020, 02, 25}) == 2);
    CHECK(classes.count({2020, 03, 10}) == 1);

    auto it = classes.find({2020, 02, 25});
    CHECK(it != classes.end());
    // Insertion order of values for std::map
    CHECK((it->second == "Unit testing" || it->second == "std::vector"));
}
```

Multimap: Querying all elements by the key

```
TEST_CASE("multimap equal_range") {
    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(range.first != range.second);
    REQUIRE(std::distance(range.first, range.second) == 2);

    auto empty_range = classes.equal_range({2020, 04, 21});
    REQUIRE(empty_range.first == empty_range.second);

    auto one_elem_range = classes.equal_range({2020, 03, 10});
    REQUIRE(std::distance(one_elem_range.first, one_elem_range.second) == 1);
    CHECK(one_elem_range.first->second == "std::string");
}
```

Multimap: Iterating over key elements

```
TEST_CASE("multimap range values") {
    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"});
}
```

Multimaps: Conclusion

`std::multimap` is there if you need it!

`std::multiset`, `std::unordered_multimap` and `std::unordered_multiset` also are!

Custom types in maps and sets

C++ London Uni

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - Composite-key type
  - Practice
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - Practice
- Few words about multimaps and multisets
- **Summary**
- Home exercise

Summary: Custom types in maps and sets

- In larger projects, using custom-types keys helps to easier understand what's in a map and saves from accidentally querying a wrong type
- In order to put a custom type in the map, you'll need to implement additional feature required by the map underlying data structure
- Good hashing function is critical to `unordered_map` and `unordered_set` performance
- If your project uses a generic library (like Boost, abseil, Qt), prefer hash-combine functions from the library instead of hand-written

Custom types in maps and sets

- Recap and updates
- Custom key types in `std::map`
  - Introduction and a simple example
  - Composite-key type
  - Practice
- Custom key types in `std::unordered_map`
  - Introduction and a simple example
  - Composite-key type and combining hashes
  - Practice
- Few words about multimaps and multisets
- Summary
- **Home exercise**

Request for questions

If something is interesting or not clear for you, it's likely someone else feels the same

Please send me questions or topics on the C++ standard library that you'd like me to cover in the last session

Submission deadline: Friday, April 24th, 11:59pm
(so I have some time to prepare)

Home exercise: Anagrams in a dictionary

C++ London Uni

1. Grab the dictionary here (it's a list of words separated by line breaks) https://github.com/dwyl/english-words/blob/master/words_alpha.txt
2. Write a function that reads a file into the vector of words
3. Write a function that for the vector of words tells the following thing
   - What's the longest anagram in the dictionary and all the sets of such words
   - What's the biggest set of words that are mutual anagrams and all the sets of that size

Home exercise: Reading the file

```
std::vector<std::string> read_all_words();

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");
}
```

```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

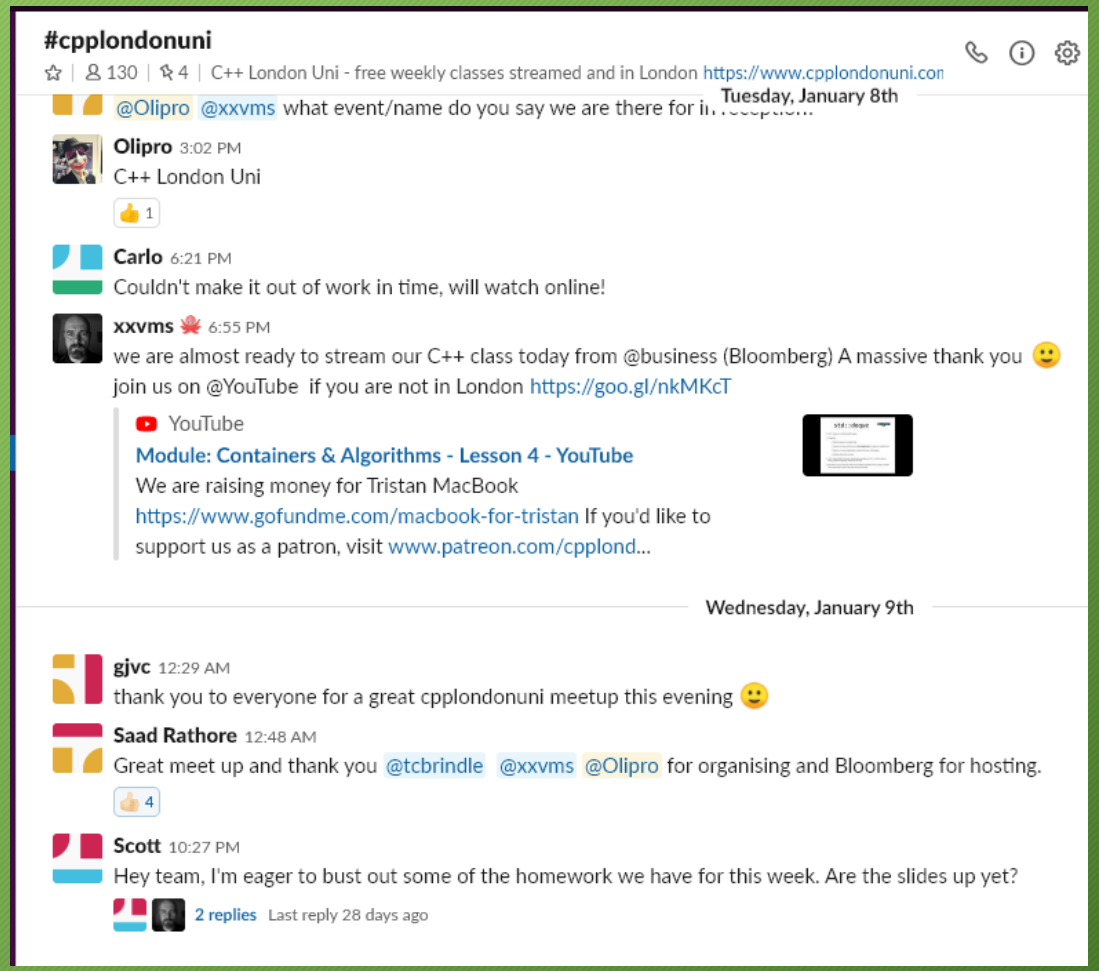
```
TEST_CASE("find anagrams -- test template") {
    const auto exciting_anagrams = find_anagrams(read_all_words());
    REQUIRE_FALSE(exciting_anagrams.most_anagrams.empty());
    REQUIRE_FALSE(exciting_anagrams.longest_anagrams.empty());

    // 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>{});
}
```

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