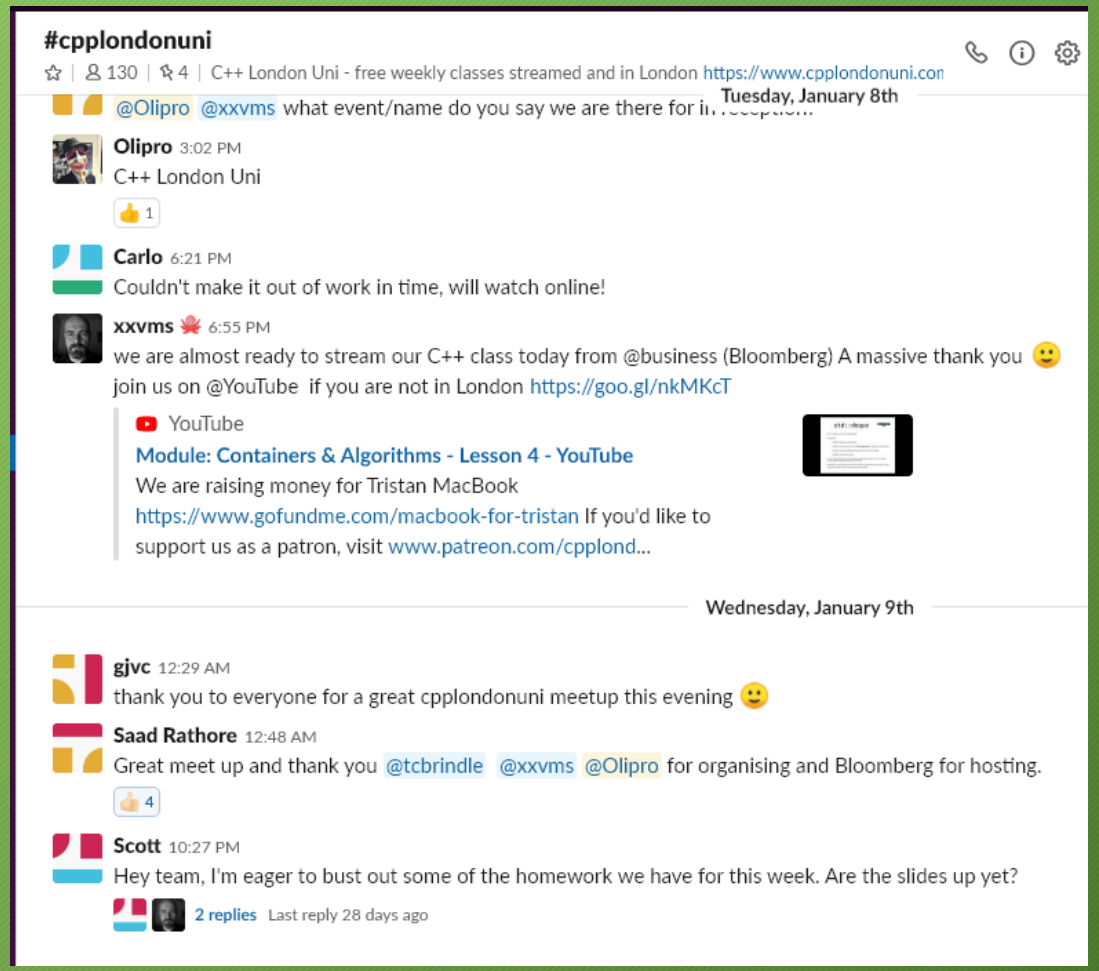
Getting to Know the Standard Library Session 5

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
   - Set algorithms
5. More standard-library algorithms

Associative containers in the standard library

- Recap and updates
- Introduction and an example problem
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - Looking up elements
  - Adding and removing elements
- Other associative containers
  - `std::unordered_map`
  - `std::set and std::unordered_set`
- Practice
- Summary
- Home exercise and additional content

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: Algorithms in the standard library

C++ London Uni

```
bool is_palindrome(std::string phrase) {
    phrase.erase(
        std::remove_if(phrase.begin(), phrase.end(),
                       [](unsigned char c) { return !std::isalnum(c); }),
        phrase.end());

    const size_t half_length = phrase.size() / 2;
    return std::equal(
        phrase.cbegin() , phrase.cbegin()  + half_length,
        phrase.crbegin(), phrase.crbegin() + half_length,
        [](unsigned char left, unsigned char right) {
            return std::tolower(left) == std::tolower(right);
        });
}
```

Home exercise: Anagrams

Write a function that given two strings tells that one is an anagram of the other taking into account only letters and digits

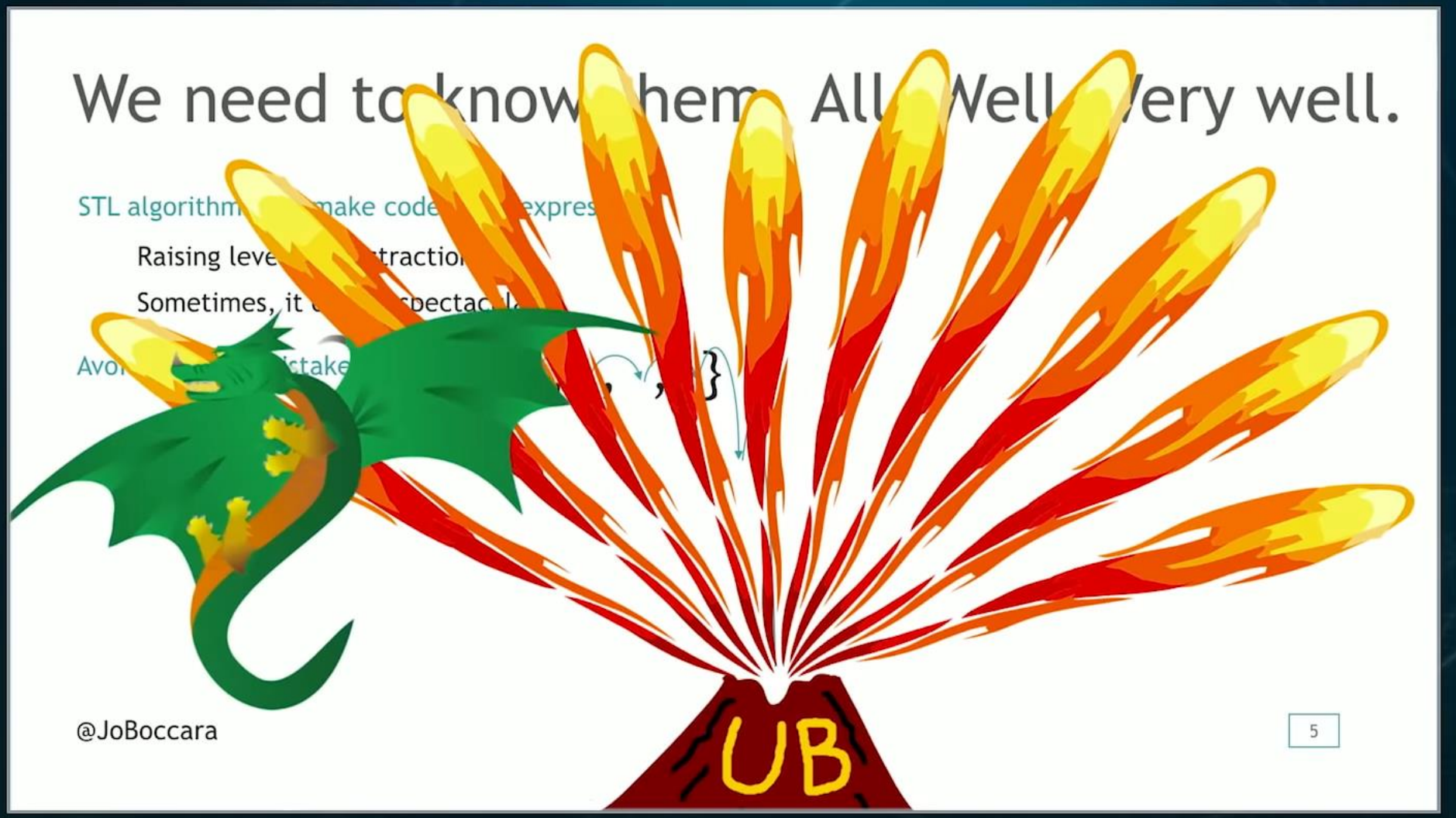
```
TEST_CASE("is anagram") {
    CHECK(is_anagram("Tom Marvolo Riddle", "I am Lord Voldemort"));
    CHECK(is_anagram("The Great Britain", "Tea? Bring it, heart!"));
    CHECK_FALSE(is_anagram("Tom Marvolo Riddle", "The Great Britain"));
}
```

Home exercise: Proposed solution

C++ London Uni

```
std::string to_alnum_lowercase(std::string phrase) {
    phrase.erase(std::remove_if(phrase.begin(), phrase.end(),
                                [](char c) { return !std::isalnum(c); }),
                 phrase.end());
    std::transform(phrase.begin(), phrase.end(), phrase.begin(),
                   [](unsigned char c) -> unsigned char {
                       return std::tolower(c);
                   });
    return phrase;
}

bool is_anagram(const std::string &left, const std::string &right) {
    const std::string left_alnum = to_alnum_lowercase(left);
    const std::string right_alnum = to_alnum_lowercase(right);
    return std::is_permutation(left_alnum.begin(), left_alnum.end(),
                               right_alnum.begin(), right_alnum.end());
}
```



105 STL Algorithms in Less Than an Hour

https://youtu.be/2olsGf6JIkU

Associative containers in the standard library

- Recap and updates
- **Introduction and an example problem**
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - Looking up elements
  - Adding and removing elements
- Other associative containers
  - `std::unordered_map`
  - `std::set` and `std::unordered_set`
- Practice
- Summary
- Home exercise and additional content

Associative containers in C++

The main goal: better-than-linear lookup of an element in the container

- `std::map` and `std::set`
  - O(logN) lookup time
  - elements are sorted
- `std::unordered_map` and `std::unordered_set`
  - O(1) lookup time
  - element order is not guaranteed

Problem: The most frequent word count

Write a function that given a vector of words returns the number of occurrences for the most frequent word in the vector

```
TEST_CASE("most frequent word count") {
    CHECK(most_frequent_word_count({}) == 0);
    CHECK(most_frequent_word_count({"banana"}) == 1);
    CHECK(most_frequent_word_count({"banana", "banana"}) == 2);
    CHECK(most_frequent_word_count({"banana", "banana", "apple"}) == 2);
    CHECK(
        most_frequent_word_count({"banana", "banana", "apple", "apple"}) == 2);
    CHECK(most_frequent_word_count(
        {"banana", "banana", "apple", "apple", "apple"}) == 3);
    CHECK(most_frequent_word_count(
        {"apple", "banana", "apple", "banana", "apple", "apple"}) == 4);
}
```

Most frequent word count: Proposed solution

C++ London Uni

```
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;
}
```

Associative containers in C++

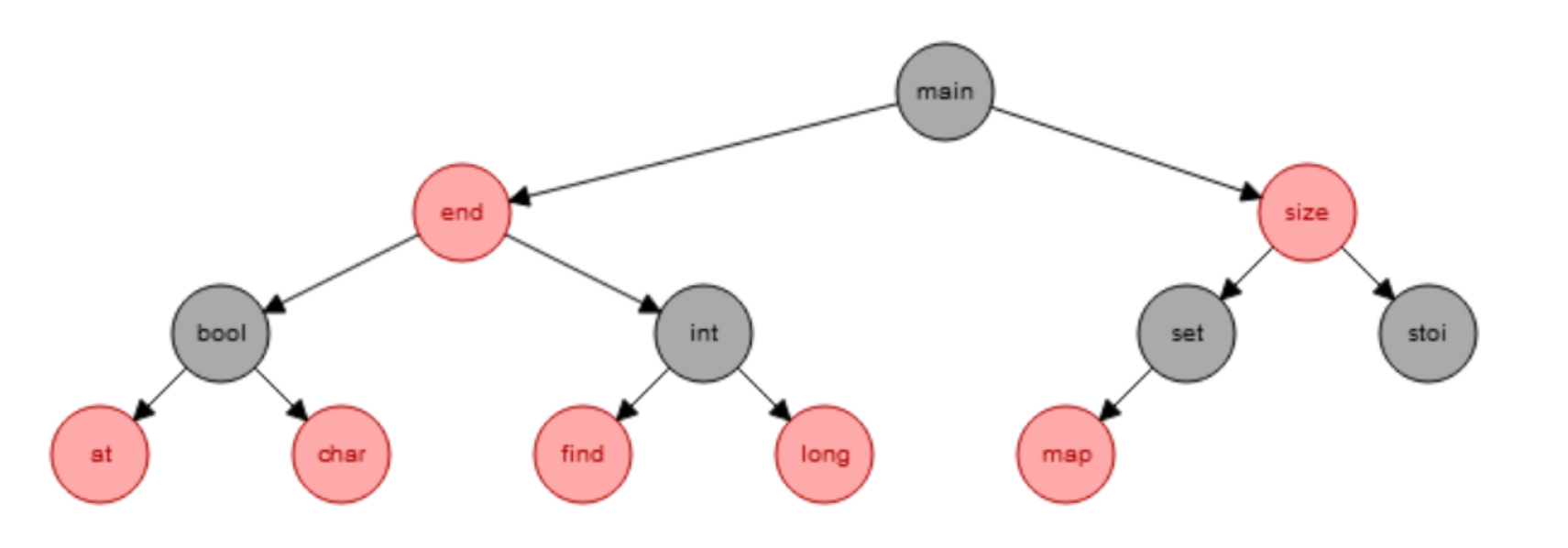
The main goal: better-than-linear lookup of an element in the container

- `std::map` and `std::set`
  - O(logN) lookup time
  - elements are sorted
- `std::unordered_map` and `std::unordered_set`
  - O(1) lookup time
  - element order is not guaranteed

```
#include <map>
#include <set>
#include <unordered_map>
#include <unordered_set>
```

Balanced binary-search tree

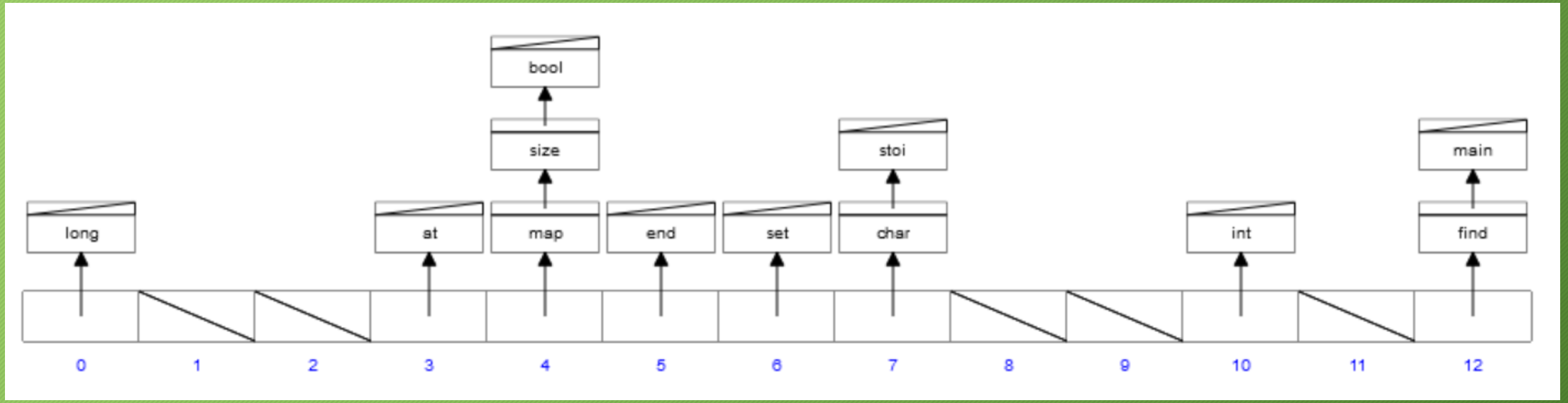
- Each node has up to two children
- All the values in the left subtree compare less than one in the current node
- All the values in the right subtree compare greater than one in the current node
- Balancing: maintaining similar depths for left and right subtrees of each node



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

Hash table (open hashing)

- Hash function maps any possible element into “bucket” index
  - Must be fast (it’s used on each lookup)
  - Must provide even distribution (otherwise lookup degenerates to linear)
- Number of buckets is adjusted according to number of elements



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

Associative containers in the standard library

- Recap and updates
- Introduction and an example problem
- **Detailed view of `std::map`**
  - Map-element type: `std::pair`
  - Looking up elements
  - Adding and removing elements
- Other associative containers
  - `std::unordered_map`
  - `std::set` and `std::unordered_set`
- Practice
- Summary
- Home exercise and additional content

std::map: Create, insert and lookup

```
TEST_CASE("std::map: very first use") {
    std::map<std::string, size_t> population;
    REQUIRE(population.empty());

    population["Birmingham"] = 1141374;
    REQUIRE(population.size() == 1);
    CHECK(population["Birmingham"] == 1141374);

    population.clear();
    CHECK(population.empty());
}
```

std::map elements: std::pair

C++ London Uni

```
TEST_CASE("map entries: std::pair") {
    std::pair<const std::string, size_t> birmingham_population{
        "Birmingham", 1141374};
    CHECK(birmingham_population.first == "Birmingham");
    CHECK(birmingham_population.second == 1141374);

    std::map<std::string, size_t> population;
    population.insert(birmingham_population);
    REQUIRE(population.size() == 1);
    CHECK(population["Birmingham"] == 1141374);
    CHECK(*population.begin() == birmingham_population);
    CHECK(population.begin()->first == "Birmingham");
    CHECK(population.begin()->second == 1141374);
}
```

Associative containers in the standard library

C++ London Uni

- Recap and updates
- Introduction and an example problem
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - **Looking up elements**
  - Adding and removing elements
- Other associative containers
  - `std::unordered_map`
  - `std::set` and `std::unordered_set`
- Practice
- Summary
- Home exercise and additional content

std::map: Check if an element is present

C++ London Uni

```
TEST_CASE("presence in map") {
    std::map<std::string, size_t> population{
        {"Birmingham", 1141374},
        {"Liverpool",  494814}
    };

    const auto liverpool_it = population.find(/*key to find*/"Liverpool");
    REQUIRE(liverpool_it != population.end());
    CHECK(liverpool_it->first == "Liverpool");
    CHECK(liverpool_it->second == 494814);

    const auto bristol_it = population.find("Bristol");
    CHECK(bristol_it == population.end());
}
```

std::map: operator[] and .at()

C++ London Uni

```
TEST_CASE("[...] vs .at(...)") {
    std::map<std::string, size_t> population;
    REQUIRE_THROWS_AS(population.at("Liverpool"), std::out_of_range);
    REQUIRE(population.empty());

    // if key is not present, operator[] creates a new value
    // with default constructed mapped type
    CHECK(population["Liverpool"] == 0);
    CHECK(population.size() == 1);

    population["Liverpool"] = 494814;
    CHECK(population["Liverpool"] == 494814);
    CHECK(population.at("Liverpool") == 494814);

    const auto const_population = population;
    // const_population["Liverpool"]; doesn't compile
}
```

Associative containers in the standard library

- Recap and updates
- Introduction and an example problem
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - Looking up elements
  - **Adding and removing elements**
- Other associative containers
  - `std::unordered_map`
  - `std::set and std::unordered_set`
- Practice
- Summary
- Home exercise and additional content

```
TEST_CASE("flavours of adding elements") {
    std::map<std::string, size_t> population;

    SECTION("insert") {
        population.insert(
            std::pair<const std::string, size_t>{"Birmingham", 1141374});
        population.insert({"Liverpool", 494814});
        CHECK(population.size() == 2);
    }

    SECTION("insert over existing element") {
        population["Bristol"];
        CHECK(population.at("Bristol") == 0);

        population.insert({"Bristol", 463405}); // a pair
        CHECK(population.at("Bristol") == 0);

        population.insert_or_assign("Bristol", 463405); // 2 args
        CHECK(population.at("Bristol") == 463405);
    }
}
```

```cpp
class CityInfo {
public:
    CityInfo() = delete;
    explicit CityInfo(size_t population) : population_{population} {}
    size_t population() const { return population_; }
private:
    size_t population_;
};

TEST_CASE("inserting an object with non-default-constructible mapped type") {
    std::map<std::string, CityInfo> cities;

    // This won't compile because operator[] internally may use default
    // constructor of CityInfo

    // cities["Birmingham"] = CityInfo{1141374};

    // These ways don't have such limitation
    cities.insert({"Bristol", CityInfo{463405}});
    cities.insert_or_assign("York", CityInfo{209893});
    CHECK(cities.size() == 2);
}
```

std::map: Removing elements

```
TEST_CASE("erase") {
    std::map<std::string, size_t> population{{"York",      209893},
                                             {"Bristol",   463405},
                                             {"Liverpool", 494814}};
    SECTION("by key") {
        population.erase("Bristol");
        CHECK(population ==
              std::map<std::string, size_t>{{"York",      209893},
                                            {"Liverpool", 494814}});
    }

    SECTION("by iterator") {
        auto bristol_it = population.find("Bristol");
        population.erase(bristol_it);
        CHECK(population ==
              std::map<std::string, size_t>{{"York",      209893},
                                            {"Liverpool", 494814}});
    }
}
```

std::map: Removing elements while iterating

C++ London Uni

```
std::map<std::string, size_t>
remove_lower_than(std::map<std::string, size_t> population, size_t threshold) {
    for (auto it = population.begin(); it != population.end();) {
        if (it->second < threshold) {
            it = population.erase(it);
        } else {
            ++it;
        }
    }
    return population;
}
TEST_CASE("return value of erase()") {
    std::map<std::string, size_t> population{{"York",       209893},
                                             {"Bristol",    463405},
                                             {"Liverpool",  494814},
                                             {"Birmingham", 1141374}};
    CHECK(remove_lower_than(population, 300000).size() == 3);
    CHECK(remove_lower_than(population, 1000000).size() == 1);
}
```

Associative containers in the standard library

- Recap and updates
- Introduction and an example problem
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - Looking up elements
  - Adding and removing elements
- Other associative containers
  - **`std::unordered_map`**
  - `std::set` and `std::unordered_set`
- Practice
- Summary
- Home exercise and additional content

std::unordered_map

Everything is the same as for `std::map`

```
TEST_CASE("unordered map has the same interface") {
    std::unordered_map<std::string, size_t> population{{"York",      209893},
                                                       {"Bristol",   463405},
                                                       {"Liverpool", 494814}};

    REQUIRE(population.size() == 3);
    CHECK(population["York"] == 209893);
    CHECK(population.at("Bristol") == 463405);
    population["Birmingham"] = 1141374;
    CHECK(population.at("Birmingham") == 1141374);
}
```

Associative containers in the standard library

C++ London Uni

- Recap and updates
- Introduction and an example problem
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - Looking up elements
  - Adding and removing elements
- Other associative containers
  - `std::unordered_map`
  - **`std::set and std::unordered_set`**
- Practice
- Summary
- Home exercise and additional content

```
TEST_CASE("std::set") {
    std::set<std::string> visited_cities{
        "Budapest", "Milton Keynes", "Brussels", "Berlin"};
    CHECK(visited_cities.size() == 4);
    CHECK(visited_cities.find("Berlin") != visited_cities.end());

    visited_cities.insert("Denver");
    std::vector<std::string> trip_to_italy{
        "Rome", "Bologna", "Florence", "Milan"};
    visited_cities.insert(trip_to_italy.begin(), trip_to_italy.end());
    CHECK(visited_cities.size() == 9);

    visited_cities.insert("Brussels");
    CHECK(visited_cities.size() == 9);

    std::vector<std::string> set_content{visited_cities.begin(),
                                         visited_cities.end()};

    CHECK(set_content == std::vector<std::string>{
        "Berlin", "Bologna", "Brussels", "Budapest", "Denver", "Florence",
        "Milan", "Milton Keynes", "Rome"});
}
```

```
TEST_CASE("std::unordered_set") {
    std::unordered_set<std::string> visited_cities{
        "Budapest", "Milton Keynes", "Brussels", "Berlin"};
    CHECK(visited_cities.size() == 4);
    CHECK(visited_cities.find("Berlin") != visited_cities.end());

    visited_cities.insert("Denver");
    std::vector<std::string> trip_to_italy{
        "Rome", "Bologna", "Florence", "Milan"};
    visited_cities.insert(trip_to_italy.begin(), trip_to_italy.end());
    CHECK(visited_cities.size() == 9);

    visited_cities.insert("Brussels");
    CHECK(visited_cities.size() == 9);

    std::vector<std::string> set_content{visited_cities.begin(),
                                         visited_cities.end()};
    std::sort(set_content.begin(), set_content.end());
    CHECK(set_content == std::vector<std::string>{
        "Berlin", "Bologna", "Brussels", "Budapest", "Denver", "Florence",
        "Milan", "Milton Keynes", "Rome"});
}
```

Associative containers in the standard library

- Recap and updates
- Introduction and an example problem
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - Looking up elements
  - Adding and removing elements
- Other associative containers
  - `std::unordered_map`
  - `std::set` and `std::unordered_set`
- **Practice**
- Summary
- Home exercise and additional content

Practice: Count unique entries in a vector

Write a function that given a vector of words returns how many words in the vector were unique

```
TEST_CASE("count unique") {
    CHECK(count_unique({}) == 0);
    CHECK(count_unique({"apple"}) == 1);
    CHECK(count_unique({"apple", "banana"}) == 2);
    CHECK(count_unique(
        {"apple", "banana", "apple", "apple", "persimmon", "banana"}) == 3);
}
```

Practice: Proposed solution

```cpp
size_t count_unique(const std::vector<std::string> &elements) {
    return std::unordered_set<std::string>{
        elements.begin(), elements.end()}.size();
}

TEST_CASE("count unique") {
    CHECK(count_unique({}) == 0);
    CHECK(count_unique({"apple"}) == 1);
    CHECK(count_unique({"apple", "banana"}) == 2);
    CHECK(count_unique(
        {"apple", "banana", "apple", "apple", "persimmon", "banana"}) == 3);
}
```

Associative containers in the standard library

C++ London Uni

- Recap and updates
- Introduction and an example problem
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - Looking up elements
  - Adding and removing elements
- Other associative containers
  - `std::unordered_map`
  - `std::set` and `std::unordered_set`
- Practice
- **Summary**
- Home exercise and additional content

Summary: Associative containers

- Associative containers can store arbitrary types and provide great element lookup time:
  - O(1) for std::unordered_map and std::unordered_set
  - O(logN) for std::map and std::set
- By default prefer the unordered_ versions, although it's more typing
- Beware of surprises:
  - Map operator[] side effects that also break it for non-default-constructible types
  - Map insert() won't override the value if the key is already in the map

Problem revision

Write a function that given a vector of words returns the number of occurrences for the most frequent word in the vector

```
TEST_CASE("most frequent word count") {
    CHECK(most_frequent_word_count({}) == 0);
    CHECK(most_frequent_word_count({"banana"}) == 1);
    CHECK(most_frequent_word_count({"banana", "banana"}) == 2);
    CHECK(most_frequent_word_count({"banana", "banana", "apple"}) == 2);
    CHECK(
        most_frequent_word_count({"banana", "banana", "apple", "apple"}) == 2);
    CHECK(most_frequent_word_count(
        {"banana", "banana", "apple", "apple", "apple"}) == 3);
    CHECK(most_frequent_word_count(
        {"apple", "banana", "apple", "banana", "apple", "apple"}) == 4);
}
```

Solution revision

```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;
}
```

Associative containers in the standard library

- Recap and updates
- Introduction and an example problem
- Detailed view of `std::map`
  - Map-element type: `std::pair`
  - Looking up elements
  - Adding and removing elements
- Other associative containers
  - `std::unordered_map`
  - `std::set` and `std::unordered_set`
- Practice
- Summary
- **Home exercise and additional content**

Home exercise: Route finding

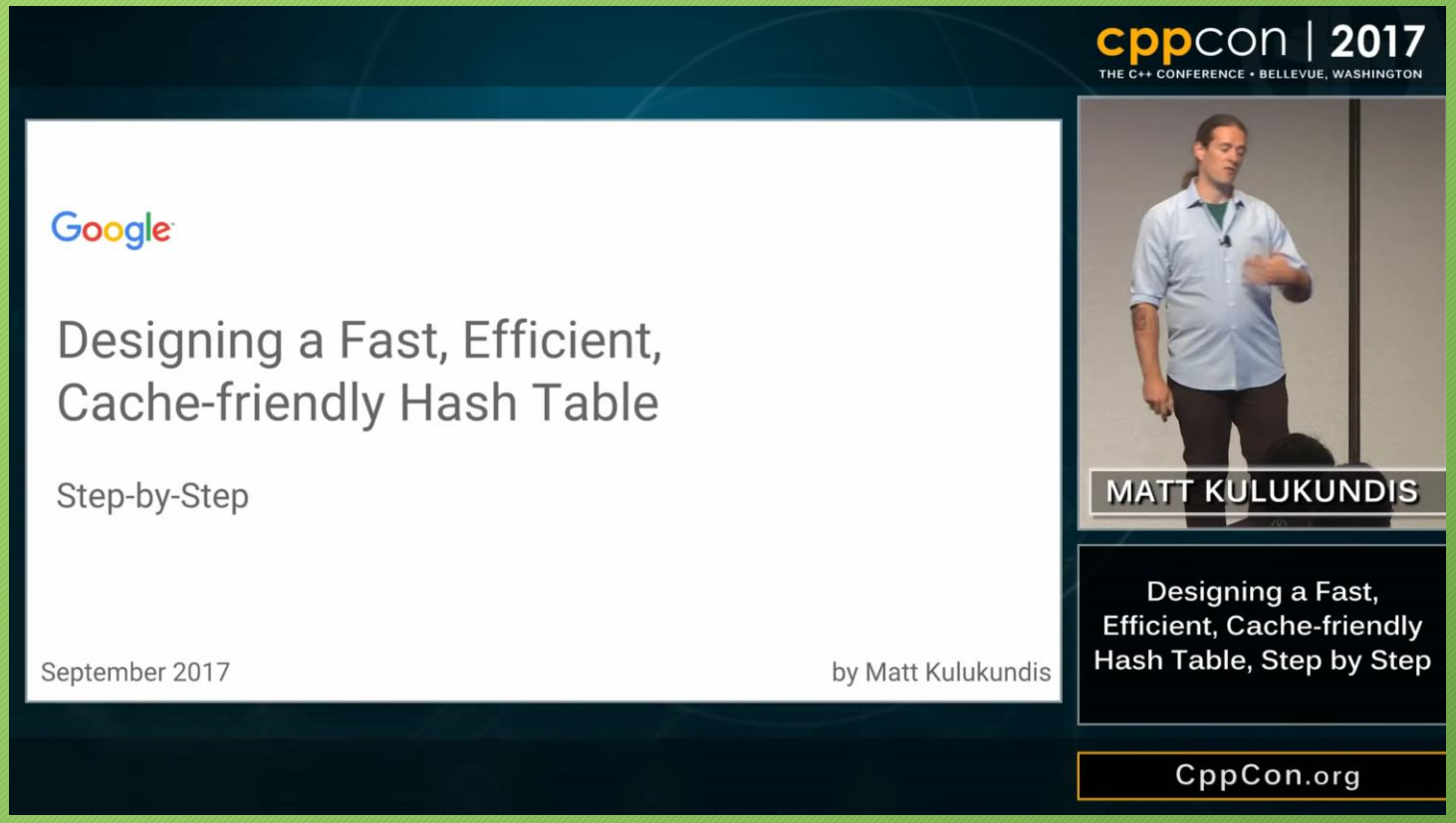
Given a vector of couples of directly connected airports,
find if there is a route with transfers between two particular airports

```
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"}));
}
```

Additional content

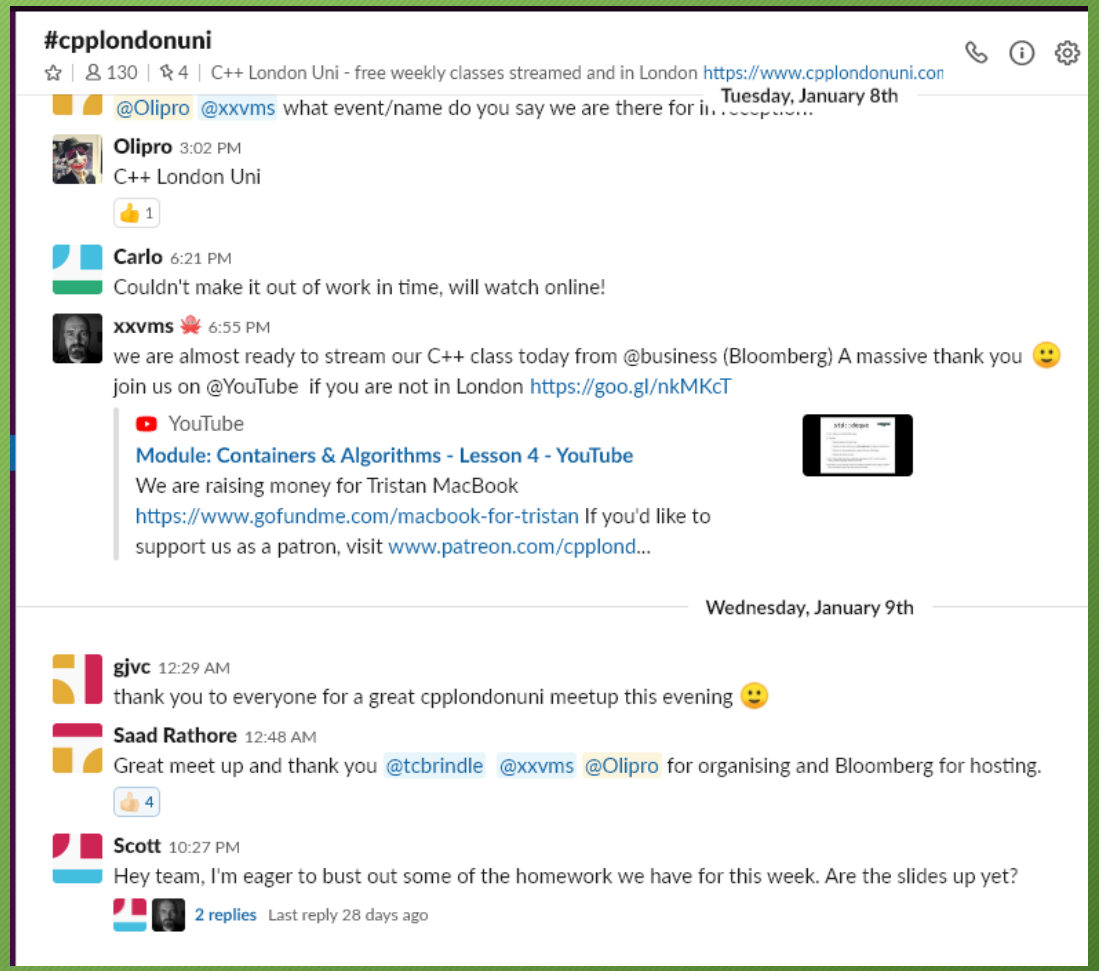


A lot of details about hash map implementation, design choices and performance implications

https://www.youtube.com/watch?v=ncHmEUmJZf4

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