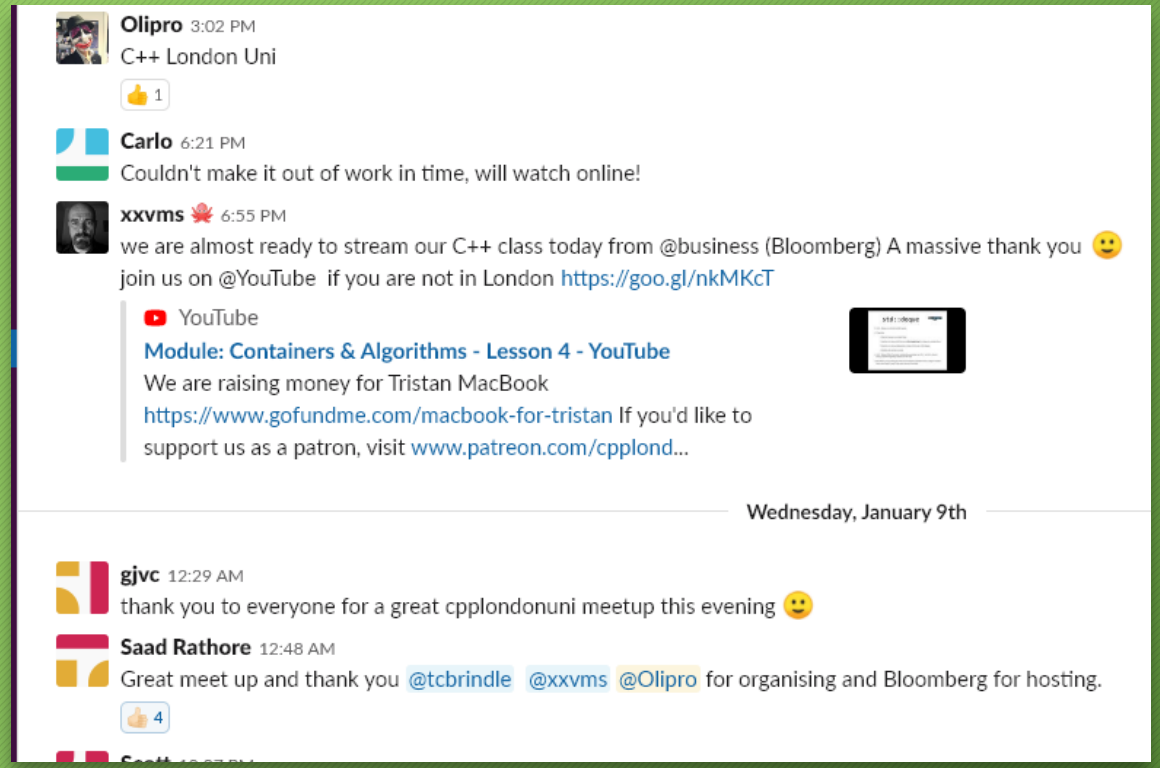
Custom Types — Session 2

Tristan Brindle

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.



Last week

- End-of-module C++ *questionnaire*

- Introduction to custom types
  - Defining simple structs
  - Creating variables of struct type
  - Passing structs to functions

This week

- Member functions
- Const member functions

Quiz results

- Hopefully everyone who completed last week's quiz now has their results
- Overall results were really good — well done everyone! 🎉
- If you have any questions about the quiz please speak to me or Oli

Last week's homework

- In the main.cpp file of a new CLion project, define a new struct called `Student`
- A Student should have two member variables, both of type `std::string`, named `first_name` and `surname`
- Write a function `void print_surname(const Student& s)` which prints the surname of the given student
- Check that your function works correctly
- Extension: create a `std::vector` of `Students`. Use a range-for loop to print the surname of each student

Solution

```
struct Student {
    std::string first_name = "";
    std::string surname = "";
};

void print_surname(const Student& s) {
    std::cout << s.surname << '\n';
}

int main() {
    const Student tom{"Tom", "Breza"};
    print_surname(tom);

    std::vector<Student> students{
            tom,
            {"Oli", "Ddin"},
            Student{“Tristan", "Brindle"}
    };

    for (const auto& s : students) {
        print_surname(s);
    }
}
```

C++ London Uni

Revision: structs

- A struct (or class) in C++ is a collection of *data members* (or *member variables*), along with *member functions* which operate on them
- We can define new struct types using the `struct` keyword
- Unlike some other languages, the keywords `struct` and `class` mean almost exactly the same thing in C++
  - The only difference is in which defaults you get
  - Today (and for the next few sessions) we'll be using the `struct` keyword, but I'll often use the two terms interchangeably

Revision: structs

- We can define a new struct type using the `struct` keyword
- Inside the struct definition we list its *data members*, similarly to how we define local variables in a function:

```
struct Point {
    int x = 0;
    int y = 0;
};
```

- A struct definition must always end with a semicolon!
- We can access the members of a struct using a `.` (dot), e.g. `p.x = 42;`

Revision: structs

- A struct definition always introduces a new type, *distinct from any other type in our program*. Two structs with different names are *different types*, even if they have the exact same members
- We can declare variables of struct type just as with fundamental (built-in) types like `int`s and `float`s
- Similarly, we can pass variable of struct type to functions, and return them from functions, just as with fundamental types
- Remember, C++ uses *value semantics* by default: data will be copied into a function, and the return value (in principle) copied out
- We can specify passing by (const) reference instead if we wish

Struct example

```
struct Example {
    std::string str = "default";
    int value = -1;
};

Example my_function(const Example& ex) {
    return ex;
}

int main() {
    Example ex1 = { "a", 1 };
    Example ex2{"b", 2}; // may omit the '='
    const Example ex3{};    // ex3.str == "default", ex3.value = -1

    auto ex4 = my_function(ex2);
    // ex4.str = "b", ex4.value = 2
}
```

Member functions

- As well as member variables, C++ allows us to define *member functions* of our structs/classes
- A member function belongs to a particular type, and *operates on an instance* of that type
- Member functions are often called *methods* in other programming languages
  - I'll sometimes use that terminology too
- Note for Java programmers: C++ methods are not overridable by default: you need to mark them as `virtual` to allow this (like C#)
  - We'll talk much, much more about this later in the course

Calling member functions

- We access members of a struct instance using a `.` (dot)
- This notation is also used to call member functions, for example

```
std::string str = "Hello world";
auto len = str.length();
```

- Here, `str.length()` calls the `length()` member function on the `std::string` instance `str`, returning the number of characters
- Like non-member (“free-standing” or “free”) functions, member functions take can take parameters, for example

```
str.resize(5); // string is truncated to 5 characters
```

Writing member functions

- We can add a member function to a struct by writing it inside the struct definition
- Here, struct `Rectangle` has a member function named `get_area()`, taking no parameters and returning a `float`
- Within the body of the `get_area()` function, we can refer to the `width` and `height` member variables of the `Rectangle` object on which the member function was called

```
struct Rectangle {
    float width = 0.0f;
    float height = 0.0f;

    float get_area()
    {
        return width * height;
    }
};
```

Member functions example

```
struct Circle {
    float radius = 0.0f;

    float get_area() {
        return M_PI * radius * radius;
    }

    void resize(float factor) {
        radius *= factor;
    }
};

int main() {
    Circle c1{2.0f};

    std::cout << c1.get_area() << '\n';
    // prints 4 * π ≈ 12.566

    c1.resize(0.5f);
    std::cout << c1.get_area() << '\n';
    // prints 1 * π ≈ 3.142
}
```

Member functions exercise

- Write a struct named Point, with two int members named x and y
- Add a member function named `equal_to` taking a second `Point` as a parameter, which returns `true` if the x and y coordinates of both points are the same
  - for example, if `Point p1 = { 1, 2 }` and `Point p2 = { 3, 4 }`, then `p1.equal_to(p2)` should return `false`
- In main(), write a few tests to make sure your member function works correctly
- Add a second member function named `not_equal_to`, again taking a Point, which returns the inverse result to the above. Again, test your new function in main()
- In main(), add a Point that is immutable. What happens when you call the equal_to() member on this Point?

Solution

```cpp
#include <cassert>

struct Point {
    int x = 0;
    int y = 0;

    bool equal_to(Point other) {
        return x == other.x && y == other.y;
    }

    bool not_equal_to(Point other) {
        return !equal_to(other);
    }
};
```

```cpp
int main()
{
    Point p1{1, 2};
    assert(p1.equal_to(p1));

    Point p2{3, 4};
    assert(!p1.equal_to(p2));
    assert(p2.not_equal_to(p1));

    Point p3{1, 4};
    assert(p3.not_equal_to(p1));
    assert(p3.not_equal_to(p2));

    const Point p4{100, 200};
    p1.not_equal_to(p4); // ?
    p4.not_equal_to(p1); // ?
}
```

Const member functions

- By default, member functions are able to change (mutate) the member variables of the instance on which they are operating
- For example:

```
struct Circle {
    float radius;
    void resize(float factor) {
        radius *= factor; // mutates radius
    }
};
```

- Problem: `const` (immutable) struct instances cannot be changed after they have been created
- Result: we cannot call member functions on immutable objects?!
- Actual result: we need to *mark which member functions are non-mutating* (read-only)

Const member functions

- To mark a member function as read-only, we add the `const` keyword after the function signature
- For example:

```
struct Circle {
    float radius;
    float get_area() const {
        return M_PI * radius * radius;
    }
};
```

- Here, the trailing `const` indicates that `get_area()` does not change the state of the `Circle` object it is called on
- Now the compiler will allow us to call get_area() on a `const Circle`

```
const Circle circle{3.0f};
float area = circle.get_area(); // okay
```

Const member functions

- Within the body of a const member function, all member variables *behave as if they had been declared* `const`
- That is, it is a **compile error** if we attempt to mutate them
- For example:

```
struct Circle {
    float radius;
    bool get_area_called = false;

    float get_area() const {
        get_area_called = true; // ERROR
        return M_PI * radius * radius;
    }
};
```

- The `mutable` keyword may be used to mark member variables which can be changed inside a const member function, but use it only in special cases

Exercise

- In the last exercise, we found that calling the equal_to() member of a `const Point` would not work
- Fix this problem by telling the compiler that `equal_to()` and `not_equal_to()` are a read-only functions
- Test that both member functions now work correctly on immutable Point instances
- What happens if you make `equal_to()` (only) a read-write function again?

“Solution”

```
struct Point {
    int x = 0;
    int y = 0;

    bool equal_to(Point other) /* not const */ {
        return x == other.x && y == other.y;
    }

    bool not_equal_to(Point other) const {
        return !equal_to(other);
    }
};


int main() {
    const Point p1{1, 2};
    const Point p2{3, 4};

    p1.not_equal_to(p2);
    // ERROR!
    // *const member functions may only call other const member functions*
}
```

Homework

- Your task for this week is to develop a simple record-keeping app for schools or universities. After each step, you should include tests to make sure everything works correctly.
- Define a new struct `Student` with three member variables: a `first_name` and a `surname` (both strings) and an id (which should be an `int`)
- Change the default initialiser for `Student::id` to use an incrementing counter. That is, the first `Student` instance you create should have id 1, the second id 2, and so on
- Add a `print()` member function to `Student` which should print out the first name, the surname and the id number, separated by spaces
- Define a new struct `ModuleRecord` with two member variables: a `Student` and an integer grade
- Define a new struct `Module` which has two member variables: a `std::string` containing the module name, and a `std::vector<ModuleRecord>` of the grades for the module
- Add an `add_record()` member function to your `Module` struct, which takes as arguments a `Student` and an integer grade. In the implementation of this member function, create a `ModuleRecord` and add it to the vector member.
- Add a `print()` member function to your `Module` struct. For each element in the member vector, print out the student's first name, surname and id number followed by their grade
- **Extension**: In the above `print()` function, print the records in descending order of their scores: that is, the highest-scoring student should have their name printed first, followed the second highest, and so on.

Thank You!

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