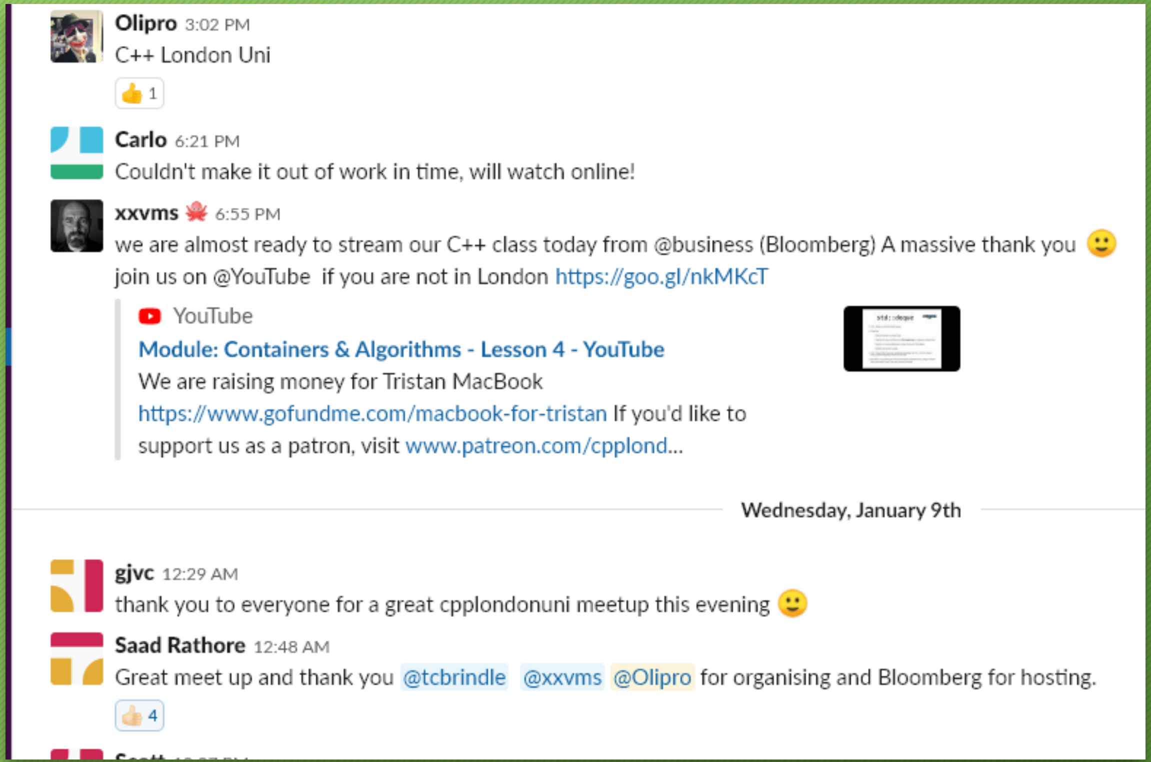
Custom Types — Session 3

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

- Structs revision
- Member functions
- Const member functions

This week

- Member function recap
- Function overloading in C++
- Intro to operator overloading

Last week's 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.

Solution

- https://github.com/CPPLondonUni/custom_types_week1_homework_soln

C++ London Uni

Revision: member functions

- C++ allows us to define member functions of structs/classes
- These are functions which belong to a struct, and operate on an instance of that struct
- Member functions are often called methods in other languages
- Ordinary, non-member functions are known as *freestanding functions* or just *free functions*
- We call a member function of an object using a ‘.’ for example

```
std::string str = "Hello World";
str.resize(5); // str is now "Hello"
```

Revision: 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
}
```

Revision: const member functions

- By default, member functions are able to change (mutate) the member variables of the instance on which they are operating
- This means that they cannot be called on immutable (`const`) objects
- Solution: we can mark member functions as being non-mutating (or read-only)
- Non-mutating member functions may be called on all instances
- But *only* non-mutating member functions may be called on `const` instances

Revision: writing const member functions

- To mark a member function as non-mutating, add the `const` keyword after the function parameter list
- For example:

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

- Within the body of a const member function, all member variables behave as if they were themselves declared `const`
  - That is, it is a compile error to attempt to modify them
  - The `mutable` keyword is an escape hatch, but use is rarely

Const member functions example

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

    float get_area() const { /* get_area() is read-only */
        return M_PI * radius * radius;
    }

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

int main() {
    Circle c1{2.0f};
    std::cout << c1.get_area() << '\n'; // okay
    c1.resize(0.5f); // okay

    const Circle c2{3.0f};
    std::cout << c2.get_area() << '\n'; // okay
    c2.resize(4.0f); // ERROR!
}
```

Function overloading

- Unlike many programming languages, in C++ we are allowed to have multiple different functions with the *same name*, but which have differing *parameter lists*
- This is known as *function overloading*
- Function overloading is very useful to provide equivalent functionality for a variety of different types
- Both member functions and non-member functions may be overloaded
    - We'll stick to talking about non-member functions for now

Function overloading

C++ London Uni

- For example, we could define two separate functions, both named `print`:

```
void print(int i) { std::cout << i << '\n'; }
void print(float f) { std::cout << f << '\n'; }
```

- We can now call print on both `int`s and `float`s:

```
print(99);      // calls print(int i)
print(1.234f); // calls print(float f)
```

- When we call an overloaded function, the compiler examines the *types of the arguments* we provide, and selects the most appropriate overload
- This selection process is called *overload resolution*
  - The overload resolution rules are extraordinarily complex (even for C++), but in general "do what you want"

Function overloading example

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

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

float get_area(const Circle& circle)
{
    return M_PI * circle.radius * circle.radius;
}

float get_area(const Rectangle& rect)
{
    return rect.width * rect.height;
}
```

```cpp
int main()
{
    Circle circle{2.0f};
    std::cout << get_area(circle) << '\n';

    Rectangle rect{3, 4};
    std::cout << get_area(rect) << '\n';
}
```

Function overloading

- It's possible (and quite common) to provide function overloads with different numbers of parameters:

```
void my_func(); // no parameters
float my_func(int i); // one parameter
bool my_func(int i, float f); // two parameters
```

- Note that the *return type* of a function is not relevant for overloading purposes
- In particular, it is not possible to overload two functions which differ only in their return type:

```
float do_stuff(int i, float f);
int   do_stuff(int i, float f); // ERROR, not a valid overload
```

Function overloading exercise

- Given the struct

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

- Write a non-member function named `equal` which compares two Circles to see whether the have the same radius, returning a `bool`
- Now add the struct

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

- Write an *overload* of `equal` which compares two Rectangles to see if they have the same width and the same height
- What happens if you try to call `equal` with one circle and one rectangle?

Solution

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

bool equal(const Circle& c1, const Circle& c2) {
    return c1.radius == c2.radius;
}

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

bool equal(const Rectangle& r1,
           const Rectangle& r2) {
    return r1.width == r2.width &&
             r1.height == r2.height;
}
```

```cpp
int main()
{
    const Circle c1{2.0f};
    assert(equal(c1, c1));

    const Circle c2{1.0f};
    assert(!equal(c1, c2));

    const Rectangle r1{3.0f, 4.0f};
    assert(equal(r1, r1));

    const Rectangle r2{99.0f, 101.0f};
    assert(!equal(r1, r2));

    equal(c1, r1); // ERROR, no matching overload
}
```

Operator overloading

- When using built-in types like `int` and `float`, we can perform operations such as addition and subtraction using the common mathematical symbols + and –
- The + and – symbols are examples of *operators*
- Some operators take two arguments, for example the equality operator `a == b`
- Other operators take one argument, for example the pre-increment operator `++i`
- A few operators have both *unary* (single argument) and *binary* (two argument) forms

Operator overloading

- C++ allows us to *define the meaning* of various operators for our own custom types
- This is called *operator overloading*
- With suitable operator overloads we can use the same “natural” syntax for our own types as we do for built-in types
- Many types in the C++ standard library make use of operator overloading, for example
  - We can compare `std::vectors` for equality using the == operator
  - We can concatenate `std::strings` using the + operator
  - We can print various types to `std::cout` using the << operator (!)

Operator overloading

- To implement an operator for a custom type, we write a function named `operator`*XX*, where *XX* is the symbol for the operator
- For example, if we defined a function named `operator+` with two parameters both of type `Matrix`, then we would be able to use the natural mathematical notation of `matrix1 + matrix2` in our code
- Note that unlike some languages, we cannot define new operators in C++

Operator overloading example

```cpp
struct Vec2 {
    float x = 0.0f;
    float y = 0.0f;
};

// Vector addition
Vec2 operator+(const Vec2& lhs, const Vec2& rhs) {
    return Vec2{lhs.x + rhs.x, lhs.y + rhs.y}
}

// Scalar multiplication
Vec2 operator*(float lhs, const Vec2& rhs) {
    return Vec2{lhs * rhs.x, lhs * rhs.y};
}

int main() {
    Vec2 a = { 1.0f, 2.0f };
    Vec2 b = { 3.0f, 4.0f };

    Vec2 c = a + b; // calls operator+(Vec2, Vec2)

    Vec2 d = 4.0f * c; // calls operator*(float, Vec2)
}
```

Operator overloading

- Operator overloading is an important way in which way can make our custom types behave like built-in types
- However, it also opens the door to doing many questionable things!

```
void operator==(const MyType& m1, const MyType& m2) {
    fire_the_missiles(); // !!!
}
```

- (Or, say, using the << operator for printing…)
- *Golden rule*: only provide operator overloads when there is a “natural” meaning of that operator for your type
    - “Do as the ints do”!

Exercise

- Using the `Circle` and `Rectangle` structs from the last exercise:
  - Write an appropriate operator overload so that you can compare two `Circle` instances using ==
  - Write an appropriate operator overload so that you can compare two `Rectangle` instances using ==
  - Write additional overloads so that you can use the `!=` operator on `Circle`s and `Rectangle`s as well

Solution

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

bool operator==(const Circle& c1,
                const Circle& c2) {
    return c1.radius == c2.radius;
}

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

bool operator==(const Rectangle& r1,
                const Rectangle& r2) {
    return r1.width == r2.width &&
           r1.height == r2.height;
}
```

```cpp
bool operator!=(const Circle& c1, const Circle& c2) {
    return !(c1 == c2);
}

bool operator!=(const Rectangle& r1,
                const Rectangle& r2) {
    return !(r1 == r2);
}

int main() {
    const Circle c1{2.0f};
    assert(c1 == c1);

    const Circle c2{1.0f};
    assert(c1 != c2);

    const Rectangle r1{3.0f, 4.0f};
    const Rectangle r2{99.0f, 101.0f};
    assert(r1 != r2);

    c1 == r1 // ERROR, no matching overload
}
```

Homework

- Grab the starter code from
- https://github.com/CPPLondonUni/week12_point_exercise
- (Don’t worry about the bit that talks about “last week”!)

- Complete exercise 1 from the README file

- Solutions are on Github if you need them, but try not to cheat :)

Thank You!

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