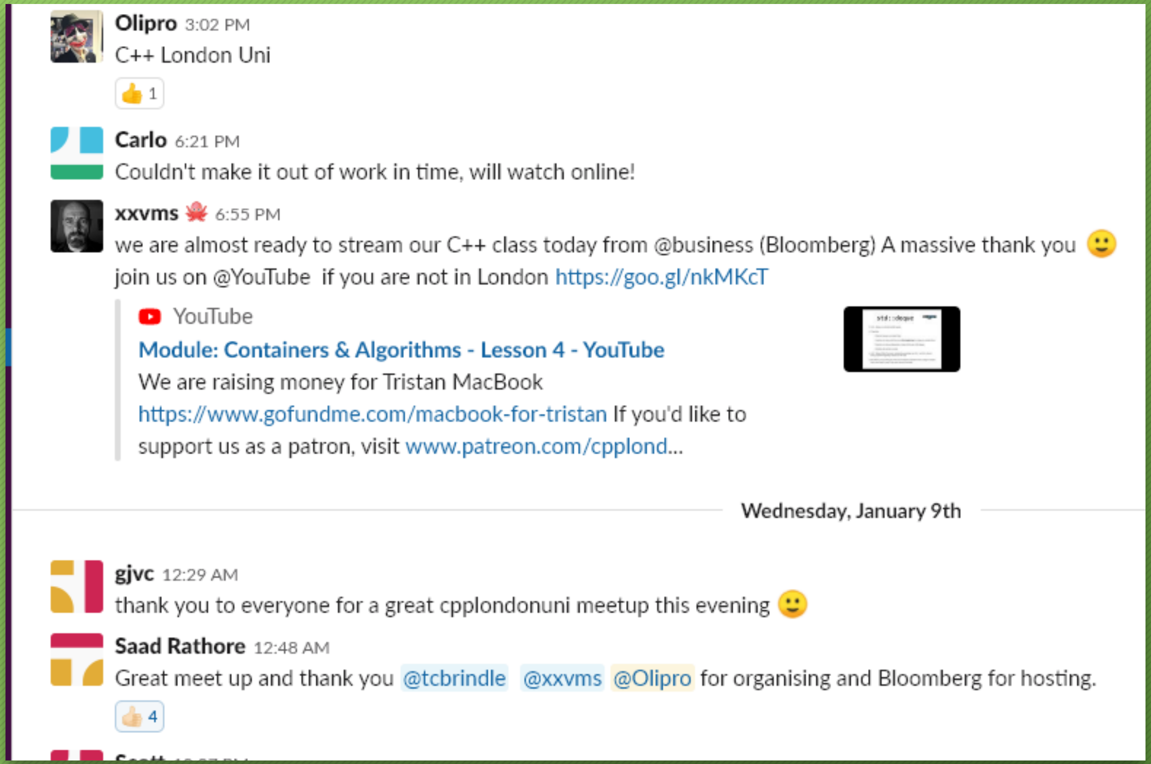
Custom Types – Session 4

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 session

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

This week

- Function overloading and operator overloading recap
- Class invariants and private data
- Introduction to constructors

Homework from last time

- 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 :)

Solution

- Solution is on the “ex1_solution” branch on Github
- https://github.com/CPPLondonUni/week12_point_exercise/tree/ex1_solution

C++ London Uni

Revision: 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

Revision: function overloading

- 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

```
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;
}
```

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

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

Revision: 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*
- 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

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”*!

C++ London uni

Access control

- When using our own types, it's often the case that only certain values "make sense" for our data members
- For example, consider our familiar `Circle` struct:

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

- Clearly, it doesn't make sense for a circle to have a negative radius!
- We say that our `Circle` type has an *class invariant* that its radius is always non-negative

Access control

- We can try to *maintain our class invariant* by writing a member function which checks that the new value is valid before using it
- (Member functions like these are often called setters, for obvious reasons)
- Problem: users of our `Circle` type can still set the radius directly!
- Solution: `private` data!

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

    void set_radius(float new_radius) {
        if (new_radius < 0.0f) {
            std::cerr << "Bad radius!\n";
        } else {
            radius = new_radius;
        }
    }
};

Circle c{};
c.set_radius(10.0f); // okay
c.radius = -100.0f; // Oh dear!
```

Access control

- We can use the `private` keyword to prevent “unauthorised access” to members of our type
- After `private:` appears in a struct definition, everything that follows is *private to the type we are defining*
  - Note that this is slightly different from the Java/C# syntax
- Private members can only be accessed from within member functions of the same struct
- Attempting to access a private member from anywhere else in the code is a compile-time error

Access control example

```cpp
struct Circle {
    // Setter
    void set_radius(float new_radius) {
        if (new_radius < 0.0f) {
            std::cerr << "Bad radius!\n";
        } else {
            radius = new_radius;
        }
    }

    // Getter
    float get_radius() const {
        return radius;
    }

private:
    float radius = 0.0f;
};
```

```cpp
int main() {
    Circle c{};

    c.set_radius(10.0f);
    // Okay
    c.radius = 10.0f;
    // ERROR, radius is a private!

    std::cout << c.get_radius() << '\n';
    // Okay
    std::cout << c.radius << '\n';
    // ERROR, radius is still private!
}
```

Access control

- There is a corresponding `public:` access level which says that the members that follow are accessible by everyone
- There is a third access level in C++, `protected`, which says that those members are only accessible from derived classes
  - We'll talk more about protected members later in the course
- We can mix and match sections with different access levels as much as we like
  - However, it's most common to list public members, followed by protected members (if any), followed by private members

Structs and classes

- Remember how I said there was almost no difference between structs and classes in C++?
- The main difference is this:
  - `struct`s default to `public` access (until we say otherwise)
  - `class`es default to `private` access (until we say otherwise)
- (The only other difference relates to the default inheritance level, which we'll talk about later)
- It's up to you whether you prefer the `struct` or `class` keyword!

Structs and classes example

```cpp
struct Circle {

    // Setter
    void set_radius(float new_radius) {
        if (new_radius < 0.0f) {
            std::cerr << "Bad radius!\n";
        } else {
            radius = new_radius;
        }
    }

    // Getter
    float get_radius() const {
        return radius;
    }

private:
    float radius = 0.0f;
};
```

```cpp
class Circle {
public:
    // Setter
    void set_radius(float new_radius) {
        if (new_radius < 0.0f) {
            std::cerr << "Bad radius!\n";
        } else {
            radius = new_radius;
        }
    }

    // Getter
    float get_radius() const {
        return radius;
    }

private:
    float radius = 0.0f;
};
```

Exercise

- Write a struct `Student` with two (public) members, `first_name` and `surname`, both of type `std::string`
- Your Student type should have the invariants that *both the first_name and surname fields should be non-empty and start with a capital letter*
- Write getter and setter member functions both fields (four member functions in total)
- In the setters, reject any input where the first letter is not uppercase (HINT: the standard library function `std::isupper` may be useful here)
- Use the `public` and `private` keywords appropriately to prevent violation of the class invariants
- Change `Student` from a `struct` to a `class`. Update your use of public and private as appropriate.

Solution

```
class Student {
public:
    std::string get_first_name() const { return first_name; }

    void set_first_name(const std::string& new_name) {
        if (new_name.size() > 0 && std::isupper(new_name[0])) {
            first_name = new_name;
        }
    }

    std::string get_surname() const { return surname; }

    void set_surname(const std::string& new_name) {
        if (new_name.size() > 0 && std::isupper(new_name[0])) {
            surname = new_name;
        }
    }

private:
    std::string first_name = "Tom";
    std::string surname = "Breza";
};
```

Constructors

- We can use private data to prevent other parts of our code from breaking our class invariants
- Problem: we can no longer set the radius when we initially construct a `Circle`:

```
Circle c{1.0f}; // Error, radius is private!
```

- Double problem: if we have a `const Circle`, we can never call `set_radius()` either!

```
const Circle c{}; // Okay
c.set_radius(1.0); // ERROR!
```

- Solution: add a *constructor* to our Circle class

Constructors

- A *constructor* is a special kind of member function which is automatically called when creating an instance of a type
- We can use constructors to *establish our class invariants*
- A constructor is written as a member function with the *same name* as the enclosing class, and *no return type*
- Within the body of the constructor, we can *modify member variables*, even if the object will be immutable once constructed

Constructor example

```cpp
class Circle {
public:
    // Constructor
    Circle(float initial_radius)
    {
        std::cout << "Calling constructor!\n”;
        if (initial_radius >= 0.0f) {
            radius = initial_radius;
        }
    }

    /* ... getter and setter as before ... */

private:
    float radius = 0.0f;
};
```

```cpp
int main() {
    Circle c{1.0f};
    // prints "Calling constructor"

    std::cout << c.get_radius() << '\n'
    // prints 1

    const Circle c2{-1.0};
    // prints "Calling constructor"

    std::cout << c2.get_radius() << '\n'
    // prints 0
}
```

Exercise

- Write a constructor for your `Student` class taking two two parameters, both of type `std::string`
- In the body of your constructor, set the first name and surname fields using the passed-in arguments
- Test your code to make sure you can construct a `const Student`
- Update the body of your new constructor to ensure that you are correctly *establishing your class invariants*

MERRY CHRISTMAS FROM EVERYONE AT C++ LONDON UNI!

Thank You!

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