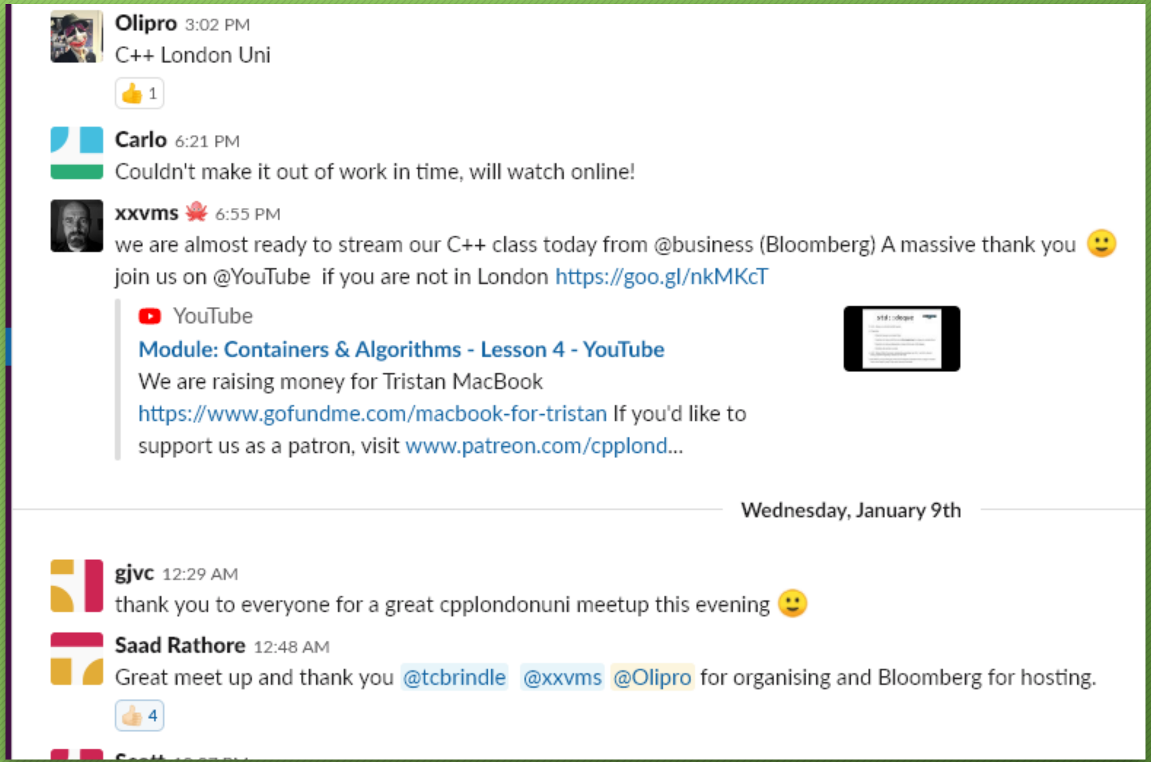
Custom Types — Session 5

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

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

This week

- Recap: member access control
- Recap: constructors
- Aggregate types
- Constructor overloading
- Default constructors
- Implicit conversions and explicit constructors

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*)
- 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.set_radius(-100.0f); // "Bad radius!"
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;
};
```

C++ London Uni

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

```
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;
};
```

```
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
}
```

An aside: aggregate types

- A struct (or class!) with has *no constructors* and which has *only public members* is called an *aggregate*
- Aggregate types are a bit special, and are a legacy of C compatibility
- Aggregates are constructed by *directly setting the values of their members*
- All the structs we have seen in previous sessions were secretly aggregates!
- Some style guides suggest that the `struct` keyword should be used for aggregates, and the `class` keyword otherwise

Aggregates example

```cpp
struct Example1 {
    int i;
    float f;
};

class Example2 {
public:
    std::string str;
    double d = 3.142;
};

class Example3 {
    std::string str;
    double d = 3.142;
};
```

```cpp
int main()
{
    Example1 ex1a = {1, 2.0f};
    // Okay, i = 1, f = 2.0f
    Example1 ex1b{1};
    // Okay, i = 1, f = 0.0f
    Example1 ex1c{};
    // Okay, i = 0, f = 0.0f;

    Example2 ex2a = {"Hello", 2.171};
    // Okay, str = "Hello", d = 2.172
    Example2 ex2b{"Hello"};
    // Okay, str = "Hello", d = 3.142
    Example2 ex3c{2.172};
    // ERROR, could not convert double to std::string

    Example3 ex3a = {"Hello", 2.171};
    // ERROR, Example3 is not an aggregate
    Example3 ex3b{};
    // Okay...
}
```

Constructor overloading

- Recall that in C++ it's possible to have multiple functions with the *same name*, but differing *parameter lists*
  - We call this *overloading*
- In the same way, it's possible (and common) to have *multiple overloaded constructors* for a class
- That is, we can have multiple constructors which have differing parameter lists
- When we create a new instance of a class, the compiler will choose which constructor best matches the arguments we supply

Constructor overloading example

```
class Circle {
public:
    // Constructor taking no parameters
    Circle()
    {
        std::cout << "Calling Circle()\n";
        radius = 0.0f;
    }

    // Constructor taking a float
    Circle(float initial_radius)
    {
        std::cout << "Calling Circle(float)\n";
        if (initial_radius >= 0.0f) {
            radius = initial_radius;
        }
    }

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

```
int main() {
    const Circle c1{};
    // prints "Calling Circle()"

    const Circle c2{3.0f};
    // prints "Calling Circle(float)"

    const std::string hello = "Hello";
    const Circle c3{hello};
    // ERROR, no matching constructor
}
```

Exercise

- Write an aggregate type called `Person` with two member variables, both of type `std::string`, called `first_name` and `surname`
- Write a class named `PersonList` which has a member variable of type `std::vector<Person>`. Add a getter function which returns (a reference to) the member variable.
- Add a *constructor* to your `PersonList` class taking a single argument of type `std::vector<Person>`, and use this to set the class member
- Add a second constructor to `PersonList` which take a single argument of type `Person`. Add this person to the member vector.

Solution

```
struct Person {
    std::string first_name;
    std::string surname;
};

class PersonList {
public:
    PersonList(const std::vector<Person>& vec) {
        people = vec;
    }

    PersonList(const Person& person) {
        people.push_back(person);
   }

    const std::vector<Person>& get_list() const {
        return people;
    }

private:
    std::vector<Person> people;
};
```

Default constructors

- A constructor taking no parameters is called a *default constructor*
- *If we don't write any constructors ourselves*, the compiler will automatically *generate a default constructor* for us!
- The compiler-generated default constructor will use *member initialisers* to set the values of its member variables
- If a member variable does not have an initialiser, it will itself be default constructed

Default constructor example

```cpp
struct Rectangle {
    float get_width() const { return width; }
    float get_height() const { return height; }
    /* ... setters omitted ... */
private:
    float width = 1.0f;
    float height;
};

class Window {
public:
    Window(std::string name_) { name = name_; }
    /* ... Getters and setters ... */
private:
    std::string name;
    Rectangle dimensions;
};
```

```cpp
int main()
{
    Rectangle rect{};
    // Okay, compiler supplies default constructor
    // rect.width = 1.0f, rect.height = 0.0f


    Window win1{"My application"};
    // Okay, dimensions is default-constructed

    Window win2{};
    // ERROR, no default constructor
}
```

Defaulted default constructors(!)

- The default constructor is one of the so-called *special member functions*
- It's special because the compiler can write one for us
- ...but it will only do so automatically if we *do not write any other constructors* of our own
- If we *do* have other constructors, we can still request that the compiler generate a default constructor by using the syntax `= default;` in place of the constructor body
- Note that this only works for special member functions!

Constructor overloading example

```cpp
struct Rectangle {
    float get_width() const { return width; }
    float get_height() const { return height; }
    /* ... setters omitted ... */
private:
    float width = 1.0f;
    float height = 1.0f;
};

class Window {
public:
    Window() = default;

    Window(std::string name_) { name = name_; }

private:
    std::string name;
    Rectangle dimensions;
};
```

```cpp
int main()
{
    Rectangle rect{};
    // Okay, compiler supplies default constructor
    // rect.width = 1.0f, rect.height = 0.0f


    Window win1{"My application"};
    // Okay, dimensions is default-constructed

    Window win2{};
    // Now okay
}
```

Default constructors exercise

- Add a *default constructor* to your PersonList class which adds a single Person named “Tom Breza” to its internal vector

- Request that the compiler generates the default constructor for you instead

- Does this do what you expect?

Solution (1)

```
class PersonList {
public:
    PersonList() {
        vec.push_back(Person{"Tom", "Breza"});
    }

    PersonList(const std::vector<Person>& vec) {
        people = vec;
    }

    PersonList(const Person& person) {
        people.push_back(person);
   }

    const std::vector<Person>& get_list() const {
        return people;
    }

private:
    std::vector<Person> people;
};
```

Solution (2)

```
class PersonList {
    std::vector<Person> people{Person{"Tom", "Breza"}};

public:
    PersonList() = default;

    PersonList(const std::vector<Person>& vec) {
        people = vec;
    }

    PersonList(const Person& person) {
        people.clear(); // Note!
        people.push_back(person);
   }

    const std::vector<Person>& get_list() const {
        return people;
    }
};
```

Implicit conversions

- When we call a function, we usually try to make the arguments we supply match the types of the parameters the function is expecting
- If the type of the argument we supply is not an exact match, then the compiler will attempt to *convert* the argument into the expected type
- For example, if we have a function taking a `float`, we can call it with an `int`:

```
float halve(float f) { return f * 0.5f }
auto two = halve(4); // 4 is an int
```

- Here, the compiler is converting the `int` we supply into the `float` that the function is expecting

Implicit conversions

- If the argument and parameter types do not match exactly, the compiler will try ***really really hard*** to convert the arguments to the expected type
- One of the ways it will do this is by looking at the *constructors* of the parameter type
- If the parameter type has a matching constructor which can be *called with one argument*, then the compiler will use that to perform the conversion
- This is sometimes a good thing…

Implicit conversion example

```
void say_hello(std::string name)
{
    std::cout << "Hello, " << name << '\n';
}

int main()
{
    std::string tom = "Tom";
    say_hello(tom);

    say_hello(std::string{"Oli"});

    auto michael = "Michael"; // "string literal", *NOT* std::string
    say_hello(michael);
    // Argument type "const char*" is implicitly converted to std::string
    // equivalent to say_hello(std::string{"Michael"});
}
```

Implicit conversion example 2

```cpp
class Circle {
public:
    Circle(float initial_radius)
    {
        if (initial_radius >= 0.0f) {
            radius = initial_radius;
        }
    }

    float get_radius() const
    {
        return radius;
    }

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

```cpp
void print_radius(const Circle& c) {
    std::cout << c.get_radius() << '\n';
}

int main() {

    const Circle c{4.2f};

    print_radius(c); // Okay, as expected

    print_radius(Circle{4.2f}); // Okay

    print_radius(4.2f); // works?!

    print_radius(99); // works?!?!?!

}
```

Explicit constructors

- The fact that we can call `print_radius()` with a `float` or even an `int` is surprising an unexpected!
- This happens because the compiler is using the `Circle(float)` constructor to perform an *implicit conversion*
- We can prevent this by using the `explicit` keyword
- When a constructor is marked as `explicit`, the compiler will not attempt to use it to perform an implicit conversion
- Get into the habit of marking **all single-parameter constructors as** `explicit`

Explicit constructor example

```
class Circle {
public:
    explicit Circle(float initial_radius)
    {
        if (initial_radius >= 0.0f) {
            radius = initial_radius;
        }
    }

    float get_radius() const
    {
        return radius;
    }

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

```
void print_radius(Circle c) {
    std::cout << c.get_radius() << '\n';
}

int main() {

    const Circle c{4.2f};

    print_radius(c); // Okay

    print_radius(Circle{4.2f}); // Okay

    print_radius(4.2f);
    // ERROR, cannot implicitly convert

    print_radius(99);
    // ERROR, cannot implicitly convert
}
```

Exercise

- Write a function called `print_names()` which takes a const reference to a `PersonList`, and prints out every name on the list

- Prevent `print_names()` from being called with an argument of type `Person`

Thank You!

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