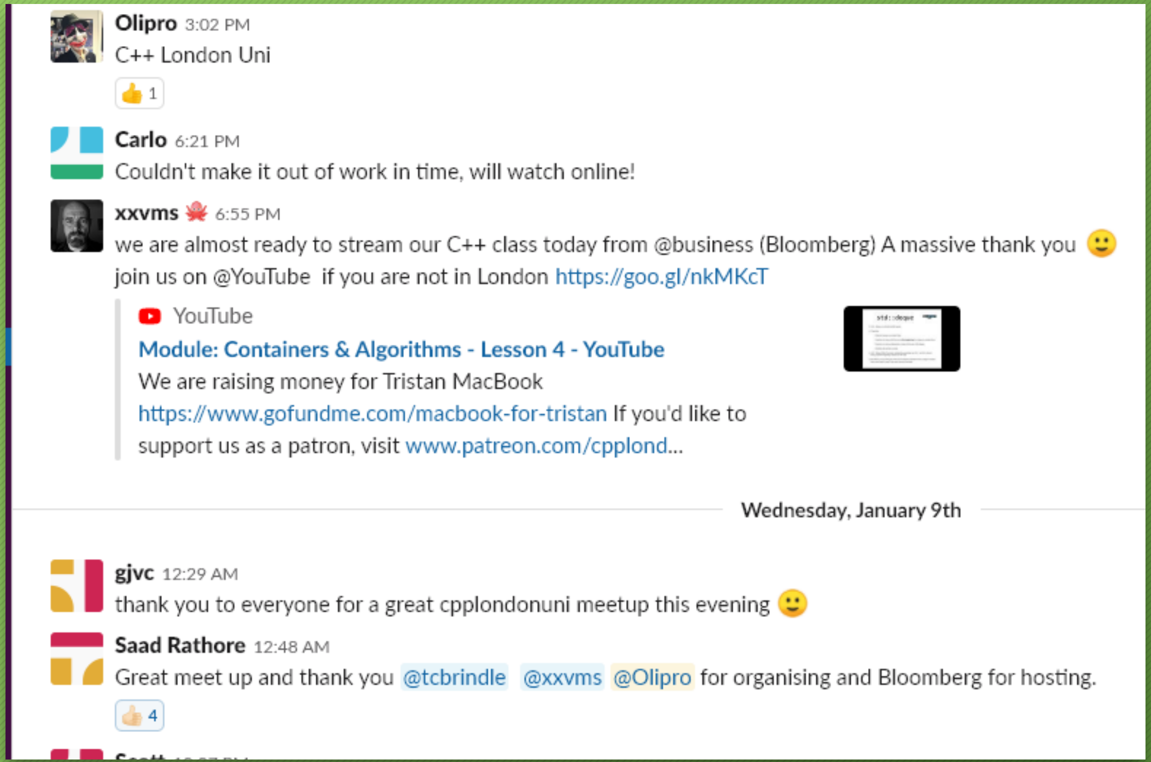
Custom Types — Session 6

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 time

- Aggregate types
- Constructor overloading
- Default constructors

This week

- Brief revision:
  - Aggregates
  - Constructor overloading
  - Default constructors and `= default;`
- Explicit constructors
- Member initialiser lists

Revision: 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

- In C++, 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
- (This uses the same *overload resolution* process as for function overloads)

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
}
```

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

```
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;
};
```

```
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
}
```

Revision: 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!

Defaulted 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 = 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;
    }
};
```

Explicit constructors

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

```
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;
};
```

```
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

```cpp
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;
};
```

```cpp
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`

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

class PersonList {
    std::vector<Person> people;

public:
    PersonList(const Person& person) {
        add_person(person);
    }

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

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

Solution

```
class PersonList {
    std::vector<Person> people;

public:
    explicit PersonList(const Person& person) {
        add_person(person);
    }

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

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

void print_names(const PersonList& names) {
    for (const auto& person : names.get_list()) {
        std::cout << person.first_name << " " << person.surname << '\n';
    }
}
```

Member initialiser lists

Member initialisation

```
class Rectangle {
public:
    Rectangle(int width, int height)
    {
        width_ = width;
        height_ = height;
    }

    /* ...getters and setters... */

private:
    int width_;
    int height_;
};
```

```
class Window {
public:
    Window(int width, int height, const std::string& name = “")
    {
        name_ = name;
        dimensions_ = Rectangle(width, height);
    }

    /* ...other member functions... */

private:
    std::string name_;
    Rectangle dimensions_;
};

int main()
{
    Window win{800, 600, "My App"};
    // ERROR: no matching function call to Rectangle::Rectangle() ?!
}
```

Member initialisation

- Why is the compiler trying to default-construct a Rectangle in the previous example?
- It turns out C++ has a strict rule: by the time we reach the body of a class constructor, **all of its member variables have already been constructed**
- Or, to put it another way, the program only proceeds to the constructor body once the member constructors have finished their work
- This means that a statement such as `rect_ = Rectangle(width, height);` in the constructor body is *assignment*, ***not** construction*!

Member initialisation

```
class Rectangle {
public:
    Rectangle(int width, int height)
    {
        width_ = width;
        height_ = height;
    }

    /* ...getters and setters... */

private:
    int width_;
    int height_;
};
```

```
class Window {
public:
    Window(int width, int height, const std::string& name = “")
    {
        name_ = name;
        dimensions_ = Rectangle(width, height);
    }

    /* ...other member functions */

private:
    std::string name_;
    Rectangle dimensions_;
};

int main()
{
    Window win{800, 600, "My App"};
    // ERROR: no matching function call to Rectangle::Rectangle() ?!
}
```

Member initialisation

- The problem in the example is that the compiler needs to construct the `Rectangle` member before it runs the body of the `Window` constructor
- There are a couple of ways we could fix this:
  - Add a default constructor to the Rectangle class
  - Use an in-class initialiser for the `dimensions_` member:

```
Rectangle dimensions_{0, 0};
```

- In either case however, we are still constructing a `Rectangle` (and `std::string`) with "dummy" values, only to immediately overwrite them. Can we avoid this somehow?

Member initialiser lists

- C++ allows us to specify a *member initialiser list* for our constructors
- The *member initialiser list* allows us to tell the compiler how it should construct our member variables
  - …and base classes, as we'll see later
- The member initialiser list appears *after* the constructor declaration, but *before* the constructor body
- Remember, by the time we reach the constructor body, **all of our members have already been constructed**

Member initialiser list example

```
class Window {
public:
    Window(int width, int height,
           const std::string& name = "")
        /* */
        : name_(name),
          dimensions_{width, height}
        /* */
    {
    }

    /* ...other member functions */

private:
    std::string name_;
    Rectangle dimensions_;
};
```

Member initialiser lists

- Using a *member initialiser list*, we can **directly** construct our `string` and `Rectangle` members with the desired values
- Notice that in this example the constructor body is empty — we have done all the work we needed to do in the initialiser list
- This is an extremely common pattern!
- **Whenever possible**, prefer using a member init list rather than performing (default-)construction followed by assignment

Member initialiser lists

- The precedence rules for member initialisation go like this:
  - If the member name appears in a constructor’s member init list, then that gets used
  - Otherwise, if the member has an *in-class initialiser*, than that gets used
  - Otherwise, the member is *default-constructed* if possible

Member initialiser lists cont.

```cpp
struct Example {
    explicit Example(const std::string& s)
        : str(s),
          i(42)
    {}

    Example(const std::string& s, int i_)
        : str(s),
          vec{1, 2, 3}
    {}

    Example() = default;

private:
    std::string str = "Hello!";
    std::vector<int> vec;
    int i = 0;
};
```

```cpp
int main()
{
    Example ex1{"Goodbye"};
    // str = "Goodbye", vec = {}, i = 42

    Example ex2{"Farewell", 99};
    // str = "Farewell", vec = {1, 2, 3}, i = 0

    Example ex3{};
    // str = "Hello!", vec = {}, i = 0
}
```

Order of initialisation

- There is one last very important rule to be aware of
- In C++, member variables are **always** constructed *in the order that they appear in the class definition*
- ***Not*** in the order in which they appear in the constructor's init list!
- To avoid confusion (and possible problems), ***always*** write the names in the member init list in the ***same order*** that they appear in the class definition
- (Compilers will usually warn you if you don't)

Order of initialisation example

```cpp
struct Example2 {
    explicit Example2(const std::string& s)
        : str(s),
          vec{str}
    {
        std::cout << vec[0] << '\n';

    }

private:
    std::vector<std::string> vec;
    std::string str;
};

int main() {
    Example2 ex{"Hello"};
}
```

Exercise

- Add a constructor to the `Person` class which takes two strings, and uses them to directly initialise the `first_name` and `surname` members
- Use a member initialiser list in the `PersonList` constructor to initialise the `people` member variable
- Implement the missing `PersonList::add_person` overload. Can you do this without constructing a temporary `Person` object?

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

class PersonList {
    std::vector<Person> people;

public:
    explicit PersonList(const Person& person) {
        add_person(person);
    }

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

   void add_person(const std::string& first_name,
                            const std::string& surname);

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

Thank You!

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