C++ London University Session 5

Tristan Brindle

Feedback

- Your feedback is vital
- Otherwise, we don’t know what you don’t know!
- If you don’t know, please **ASK**

Today's Lesson Plan

- Solutions to last week's exercises
- More about constructors and special member functions
- All about access specifiers

“Homework” questions from last week

- Last week’s “homework” was to complete the group exercises we started in class
- These were intended to be tough! So don’t worry if you didn’t manage them
- My solutions are available at

  https://github.com/CPPLondonUni/week4_group_exercises/tree/solutions

Constructors and special member functions

- For this section we'll be using the *constructor playground*
- You can find this at

https://github.com/CPPLondonUni/constructor_playground

Constructors

- A member function with the same name as the class is called a *constructor*
- A constructor's job is to make the object ready for use, by performing any initialisation of member variables (and base classes) that might be required
- Constructors do not have a return type, and may not be declared `const`

```
struct example {
    example(int i, float f); // constructor
};
```

Constructors (2)

- Constructors may declared in a header and defined in an implementation file, just like normal member functions
- Constructors may be overloaded, just like normal member functions
- A constructor which takes no arguments is called a *default constructor*, and is one of the six *special member functions*

Constructors (3)

- If you do not write any constructors for your class, a default constructor will be provided for you by the compiler if possible
- You can tell the compiler to generate a default constructor by using the syntax `= default`, i.e.

```
struct example {
    example() = default;
};
```

- You can tell the compiler **not** to generate a default constructor (when it otherwise would) by using the syntax `= delete`, i.e.

```
struct example {
    example() = delete;
};
```

Exercise

- Declare constructor in class example which takes an `int` as an argument. Which special member functions are now provided?

- Add a *compiler-generated default constructor* to example, in addition to the constructor you have just defined

- Try marking the default constructor as deleted. What happens?

- Try marking your new constructor as `= default`. What happens?

Explicit constructors

- A constructor which takes a single argument can be used as an *implicit conversion* in some circumstances

- For example:

```
struct example {
    example(int i);
};

void func(const example& e);

func(3); // Not an error!
```

Explicit constructors (2)

- Implicit conversions like these can have surprising effects, and are usually not desired
- This can be prevented by using the keyword `explicit` in front of the constructor
- Get into the habit of declaring all single-parameter constructors `explicit` by default

```
struct example {
    explicit example(int i);
};

void func(const example& e);

func(3); // Now a compile error
func(example{3}); // Okay
```

Explicit constructors (3)

- Like other functions, constructors can have *default arguments*
- This means that it’s not always obvious when a constructor can take a single argument, and therefore be a candidate for implicit conversion

Exercise

- In the file `example.cpp`, uncomment the lines in `test_example()`. Notice that we can pass an `int` to `function_taking_example()`. Try to work out what's happening when we do this.

- In `example.hpp`, mark the constructor taking an `int` as explicit. What happens when we now try to compile example.cpp?

- Comment out the `int` constructor and add a new constructor with signature

  `example(int i, double d = 0.0)`.

  Notice that we can now (again) compile `example.cpp` without any errors.

- Remove the default argument from the constructor you've just added, so it now requires two arguments. What happens if we mark it as `explicit`?

- Can you think of a situation where we would **not** want to mark a single-argument constructor as `explicit`?

Member initialisers

- A constructor's job is to make the class ready for use. This includes setting the initial values of member variables.

- To set the initial values, we can use a *member initialiser list*. This appears after the declaration, but before the body of the constructor

```
struct example {
    explicit example(int i, float f)
        : mem1{i}, mem2{f}
    {
        // assert(mem1 == i);
    }

    int mem1;
    float mem2;
};
```

Member initialisers (2)

- We can also provide initial values in our member variable declarations themselves, as we've seen in our `point` class

- The constructor initialiser list overrides the default values provided in the member declaration, if any

- **Always ensure that all member variables are initialised**, either with a default member initialiser or in the constructor initialiser list.

Exercise

- Add member variables `int i = 3` and `double d = 4.0` to our example class.
- Add member initialiser lists to the constructors of `example`, initialising the members appropriately
- In `test_example()`, print the values of the `i` and `d` members of class `example`.
- In `example.cpp`, experiment with initialising an `example` with different constructors, and notice how the member initialiser list overrides the default member initialisers.

Copy constructors

- A constructor which takes another object of the same type as an argument (as a const reference) is called a *copy constructor*

- A copy constructor is used by the compiler when a copy of a variable is required, for example when we pass variables to functions *by value*

```
void func(example e);

struct example {
    example(const example&); // copy constructor
};

example e1;
auto e2 = e1; // Uses copy constructor
func(e1); // Passes a copy of e1 to func() using copy constructor
```

Copy constructors (2)

- The copy constructor is another of the special member functions
- If you do not provide a copy constructor, the compiler will provide one for you, calling the copy constructor of each member variable in turn
- Most of the time, you do not need to write your own copy constructor

Exercise

- Add a copy constructor to class example
- What should the member initialiser list of the copy constructor contain?
- Try marking the copy constructor as `= default`. What happens?
- Try marking the copy constructor as `= delete`. What happens in `example.cpp`? Why?

Destructors

- A destructor is another *special member function* that runs when an object's lifetime ends
- A destructor's job is to clean up any resources that the object may be using
- You declare a destructor with a ~ in front of the class name. A destructor takes no arguments and has no return type. For example:

```
struct example {
    ~example();
};

{
    example e1{};
} // <- destructor is called here
```

Destructors (2)

- After the destructor body has run, the compiler will call the destructor of each member variable in turn.
- As with the other special members, the compiler will provide a destructor if you do not write one yourself.
- As with the other special member functions, you normally do not need to write your own destructor

Exercise

- Add a destructor to class example. What should it do?
- Try marking the destructor as = `default`. What happens?
- Try marking the destructor as = `delete`. What happens? Why?

Copy assignment operator

- The final special member that we'll be talking about today is the copy assignment operator

- This is an overload of `operator=` that takes a const reference to an object of the same type, returning a non-const reference to the assigned object

- The assignment operator is used when assigning to an *already constructed* variable. For example:

```
struct example {
    example& operator=(const example&);
};

example e1{};
example e2{};
e2 = e2; // calls assignment operator
example e3 = e2; // Copy constructor, not assignment
```

Copy assignment operator

- As with the other special members, the compiler will provide a copy assignment operator if you do not define an assignment operator yourself
- The compiler-provided version will simply assign each member variable in turn
- As with the other special members, you normally do not need to write your own copy assignment operator

Exercise

- Add a copy assignment operator to class example, marked = `default`. What happens?

- What happens if we mark the copy assignment operator as = `delete`?

- Implement the copy assignment operator yourself. In `example.cpp`, make sure that your operator is working correctly.

The Rule of Three

- The “rule of three” says that if you implement one of the special member functions (destructor, copy constructor, copy assignment operator), you almost certainly need to provide all three

The Rule of ~~Three~~ Five

- The “rule of three” says that if you implement one of the special member functions (destructor, copy constructor, copy assignment operator), you almost certainly need to provide all three

- C++11 added two new special member functions, a *move constructor* and *move assignment operator* (which we’ll talk about in a future session), turning this into the “rule of five”

The Rule of ~~Three~~ ~~Five~~ Zero

- The “rule of three” says that if you implement one of the special member functions (destructor, copy constructor, copy assignment operator), you almost certainly need to provide all three
- C++11 added two new special member functions, the move constructor and move assignment operator (which we’ll cover in a future session), turning this into the “rule of five”
- But much better is the rule of zero: in general, you should not provide any of the special member functions (other than perhaps a default constructor) yourself, but use the compiler-provided versions

Any questions before we move on?

Member access

- In C++ there are three access levels for member functions and member variables of classes: `public`, `private`, and `protected`

- Member access levels are used to provide *encapsulation*, by controlling how and when an object's value may change

- The main difference between the `struct` and `class` keywords in C++ is that in a `struct` members are *public* by default, and in a `class` members are *private* by default.

Public member access

- We can use the keyword **public:** within a class/struct definition to signify that all the members that follow (until the next access specifier) are publicly accessible.

- For example:

```
class example {
public:
    void public_member_function();

    int public_member_variable = 0;
};
```

Public member access

- Public members have no access restrictions
    - Other functions and classes can call public member functions
    - Other functions and classes can read from and write to public member variables
- The public members of a class define its *public interface*

Public member access

- We can use the keyword **private:** within a class/struct definition to signify that all the members that follow (until the next access specifier) are privately accessible.

- For example:

```
class example {
private:
    void private_member_function();

    int private_member_variable = 0;
};
```

Private member access

- Private members may only be accessed from within member functions of the same class
  - Other functions and classes may not call private member functions
  - Other functions and classes may not read from or write to private member functions

Protected member access

- Protected members are only accessible by members of the same class (with with private members), and by members of derived classes

- We’ll be taking more about protected members when we discuss inheritance

Friends

- We can use the keyword friend to allow unrelated functions and classes access to a type's private and protected members.

- For example

```
void other_function();

class other_class;

class example {
public:
    friend void other_function();

    friend other_class;
};
```

Friends

- Granting friendship to a function means that that function can access our private (and protected) members without restriction

- Granting friendship to another class means that that class's members can access our private (and protected) members without restriction

- One common use of friend functions is to allow an output stream operator overload to access private member variables, in order to print their value

Exercise

- https://github.com/CPPLondonUni/course_materials/tree/master/week5/member_access

Next week

- Inheritance
- Virtual functions and polymorphism
- Pointers and smart pointers

“Homework”

- Finish the member access exercise
- TBA!

Online Resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)

Thanks for coming!

C++ London University:

- Website: cpplondonuni.com
- Github: github.com/CPPLondonUni

Where to find Tom Breza:

- On Slack: cpplang.slack.com #learn #cpplondon
- E-mail: tom@PCServiceGroup.co.uk
- Mobile: 07947451167

My stuff:

- Website: tristanbrindle.com
- Twitter: @tristanbrindle
- Github: github.com/tcbrindle

See you next time! 🙂