Introduction to Object Orientated Programming in C++ — Session 1

Tristan Brindle

Register now for C++ on Sea!

https://cpponsea.uk/

Feedback

- We love to hear from you!
- The easiest way is via the *cpplang* channel on Slack — we have our own chatroom, *#ug_uk_cpplondonuni*
- Go to https://cpplang.now.sh/ for an “invitation”

About these sessions

- We’re adopting a more modular approach to C++ London Uni

- This is the first session of a 4-week “block” introducing object-orientated programming in C++

- This is very much just an introduction!

What is object-orientated programming?

- *Object-orientated programming* (OOP) is programming style (or “paradigm”) that is based around the idea of self-contained *objects* which interact with each other

- Became popular in the 80s and 90s — many languages around this time were designed around OOP

- OOP is particularly well-suited to modelling real-world objects and their relationships

- Typically, OO languages (including C++) have a notions of *classes* and *class inheritance*, and provide mechanisms for *abstraction* and *encapsulation*

OOP fundamentals

- In object-orientated programming, we model the world as consisting of *objects* which interact with each other

- An object is an *instance* of a particular *class*

- Classes define the *state* that an object has and the *actions* that it can perform

- For example, a `DeskLamp` class might have two states, *on* or *off*, and an action (the switch) which changes its state

- An individual *instance* of the `DeskLamp` class is an object that represents a physical lamp

OOP fundamentals

- In C++, we model the *state* of an object using *member variables*

- Member variables can themselves be instances of other classes — this is called *composition*

- For example, a `Bicycle` class might have two member variables of type `Wheel`

- Composition models a "*has-a*" relationship: a bicycle *has a* wheel

OOP fundamentals

- In C++, we model the *actions* that an object can perform using *member functions*

- For example, a `Bicycle` class might have actions such as *pedal*, *brake* and *steer*

- Actions often modify the *state* of an object. For example, the *pedal* action may increase the *speed* of a `Bicycle`

- However, we may have actions which are merely *observers* and do not modify the state, such as a *get_speed* action on a `Bicycle`

Worked example

Any questions before we move on?

Encapsulation

- In strict OOP, we do not allow external actions to directly read or modify the state of an object

- Rather, all interactions with an object's state should be done via its actions (that is, member functions)

- Keeping an object's state *private* allows us to change the *implementation* of an object without affecting its external *interface*

- This is known as *encapsulation*

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*

Private 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 only 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 variables

Protected member access

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

- We'll be taking more about protected members when we discuss inheritance next week

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://classroom.github.com/a/XdHs3td2

Next time

- Inheritance in C++
- Virtual functions

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…)