Introduction to Object Orientated Programming in C++ — Session 3

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* group on Slack — we have our own channel, *#cpplondonuni*
- Go to https://cpplang.now.sh/ for an “invitation”

This session

- Revision from last week
- Virtual functions in C++
- Pure virtual functions and abstract classes

Revision

- Like many other languages, C++ allows class types to *inherit from* other class types
- Unlike most other languages, C++ has three kinds of inheritance: `public`, `private` and `protected`
- *Public inheritance* is almost always what you want
- As with member access, `struct`s default to *public inheritance*, `class`es default to *private inheritance*

Revision

- Some languages use the terms *subclass* and *superclass* to talk about inheritance relationships; in C++, the usual terminology is *base class* and *derived class*
- When we *publicly inherit* from a base class B, our derived class D (literally) contains an instance of B, where:
  - All of B's `public` members are `public` in D
  - All of B's `protected` members are `protected` in D
  - B's `private` members *cannot be accessed* in D

Revision

- C++ does not have a language-level distinction between base classes and interfaces

- Instead, C++ allows *multiple inheritance* from an arbitrary number of base classes

- Unlike other languages, there is no universal “root object” type in C++ (e.g. `java.lang.Object`, `System.Object`, `NSObject` etc)

Revision

- To call a base class constructor from a derived class constructor, we use the initialiser list:

```
struct Base {
    explicit Base(std::string s);
};

struct Derived : Base {
    explicit Derived(int i)
        : Base(std::to_string(i))
    {}
};
```

- If we do not specify which constructor to use, the compiler will (attempt to) use the *default constructor* of the base class

- **Be careful** when using pass-by-value and inheritance: it usually won't do what you want

Last week's exercise

- https://github.com/CPPLondonUni/inheritance_oop_example

Any questions before we move on?

Virtual functions


```cpp
// generator.hpp
struct Generator {
    int generate() { return 0; }
};

void print_number(Generator& gen);
```

```cpp
// generator.cpp
#include "generator.hpp"

void print_number(Generator& gen)
{
    std::cout << gen.generate() << '\n';
}
```

```cpp
// two_generator.hpp
#include "generator.hpp"

struct TwoGenerator : Generator {
    int generate() { return 2; }
};
```

```cpp
// main.cpp
#include "two_generator.hpp"

int main()
{
    TwoGenerator two_gen{};
    int i = two_gen.generate();
    std::cout << i << '\n';
    // prints 2
    print_number(two_gen);
    // prints 0!
}
```

Virtual functions


```cpp
// generator.hpp
struct Generator {
    virtual int generate() { return 0; }
};

void print_number(Generator& gen);
```

```cpp
// generator.cpp
#include “generator.hpp"

void print_number(Generator& gen)
{
    std::cout << gen.generate() << '\n';
}
```

```cpp
// two_generator.hpp
#include "generator.hpp"

struct TwoGenerator : Generator {
    int generate() override { return 2; }
};
```

```cpp
// main.cpp
#include "two_generator.hpp"

int main()
{
    TwoGenerator two_gen{};
    int i = two_gen.generate();
    std::cout << i << '\n';
    // prints 2
    print_number(two_gen);
    // now prints 2
}
```

Virtual functions

- A *virtual function* in C++ is a member function whose behaviour can be *overridden* by derived classes

- The C++ runtime ensures that the derived class's version of a virtual function will *always be used*, no matter how the function is called

- This allows us to write code which uses only base class *interfaces*, without knowing the details of derived class *implementations*

Virtual functions

- To declare a virtual function in a base class, we use the keyword `virtual` in front of the function in its *declaration*
- For example

```
struct Animal {
    virtual void speak();
    // speak() is a virtual function
};
```

Virtual functions

- To *override* a virtual function in a derived class, we can simply write a member function with the same signature:

```
struct Dog : public Animal {
    void speak() { std::cout << "Woof\n"; }
};

struct Cat : public Animal {
    void speak() { std::cout << "Meow\n"; }
};
```

Example


```
struct Animal {
    virtual void speak();
};

void say_hello(Animal& a)
{
    a.speak();
}
```

```
struct Dog : public Animal {
    void speak() { std::cout << "Woof\n"; }
};

struct Cat : public Animal {
    void speak() { std::cout << “Meow\n"; }
};

int main()
{
    Dog d;
    Cat c;

    say_hello(d); // prints "Woof"
    say_hello(c); // prints "Meow"
}
```

Virtual functions

- When implementing a virtual function, we need to be careful: if the signature does not match **exactly**, then we are *declaring a new function*, not overriding!

```cpp
struct Animal {
    virtual void speak() const; // Note, const!
};

struct Dog : public Animal {
    void speak() { std::cout << "Woof\n"; }
    // Not an override!
};
```

Virtual functions

- To avoid this, we can use the `override` “keyword” after the function declaration in the derived class:

```
struct Animal {
    virtual void speak() const;
};
```

```
struct Dog : public Animal {
    void speak() override; // Error!
};
```

- Now the compiler will give us an error message if we are not correctly overriding a virtual function:

```
error: 'void Dog::speak()' marked 'override', but does not override
```

- **Always** use `override` (or `final`) when implementing virtual functions!

Exercise

- https://github.com/CPPLondonUni/oop_logging_exercise

- Please complete exercise 1

Any questions before we move on?

Pure virtual functions

- A *pure virtual function* is a virtual function which we *must* override in a derived class

- Pure virtual functions are often called *abstract methods* in other languages

- We can declare a pure virtual function by adding `= 0` to the end of the declaration in the base class

```
struct Animal {
    virtual void speak() = 0;
    // speak() is pure virtual
};
```

Abstract classes

- A class with at least one pure virtual function is called an *abstract class*
- Because pure virtual functions must be overridden in a derived class, the compiler will prevent us from directly creating an instance of an abstract class:

```cpp
struct Animal {
    virtual void speak() const = 0;
};

struct Dog : Animal {
    void speak() const override;
};

int main()
{
    Animal a{}; // error
    Dog d{}; // okay
}
```

```
<source>:11:12: error: variable type 'Animal' is an abstract class
    Animal a{}; // error
           ^
<source>:2:18: note: unimplemented pure virtual method 'speak' in 'Animal'
    virtual void speak() const = 0;
```

Interface classes

- A common pattern is to declare a base class consisting only of pure virtual member functions, and no data members

- This is sometimes called an *interface class*, and is functionally equivalent to an `interface` in Java or C#

- The usual advice is to inherit from any number of interface classes, but at most one non-abstract base class

- This pattern is enforced by the language in Java and C#

Exercise

- https://github.com/CPPLondonUni/oop_logging_exercise

- Please complete exercises 2 and 3

- If you did not manage to finish exercise 1, a solution can be found in the **ex1_solution** branch

Next time

- Virtual destructors
- Dynamic lifetimes and `unique_ptr`
- Module test

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…)