C++ London University Session 11

Tristan Brindle

Feedback and Communication

- Your feedback is vital
- Otherwise, we don’t know what you don’t know!
- Please join the #ug_uk_cpplondonuni channel on the cpplang Slack — Go to https://cpplang.now.sh/ for an “invitation”

Today's Lesson Plan

- Inheritance in C++
- Virtual functions and polymorphism

Revision: Access specifiers

- Recall that C++ has three kinds of *access specifier* for class members: `private`, `protected` and `public`
- *Private* members are only accessible by other members of the same class
- *Protected* members are accessible by members of the same class, and by members of derived classes
- *Public* members are accessible by everyone

Revision: Access specifiers

- The `friend` keyword can be used to give other classes or functions access to private members

- For types declared with the `struct` keyword, the default access level is public

- For types declared with the `class` keyword, the default access level is private

Inheritance in C++

- Like many other languages, C++ allows class types to *inherit from* other class types
- This means (loosely) that a class “builds upon” the classes it inherits from
- Base classes can in turn inherit from other classes, forming an *inheritance hierarchy*.
- A base class can declare virtual functions which may be overridden in a derived class; (much) more on this later

Inheritance in C++

- To declare an inheritance relationship, we use the syntax

```
class Base {
    /* ...members... */
};

class Derived : [access-specifier] Base {
    /* ...members... */
};
```

- The access specifier is optional, and can be one of `private`, `protected` or `public`

- If the access specifier is omitted, it defaults to `public` when using the `struct` keyword, and `private` when using the `class` keyword.

Inheritance in C++

- Other languages use the terms *subclass* and *superclass* for describing an inheritance relationship. In C++, we usually talk about *base classes* and *derived classes*

- Unlike some other languages, in C++ there is no language-level distinction between base classes and interfaces — we only have classes

- Unlike some other languages, in C++ there is no "root object" type from which everything derives (e.g. `java.lang.Object`, `System.Object`, `NSObject` etc)

- Unlike some other languages, in C++ a type can have more than one base class — this is called *multiple inheritance*

Inheritance in C++

- C++ offers three levels of inheritance: `private`, `protected` and `public`
- Public inheritance is what we mean when we talk about "inheritance" without qualification
- 99% of the time public inheritance is what you want

Private Inheritance

- When using private inheritance, all `public` and `protected` members of the base class are accessible as `private` members of the derived class.

- Private inheritance can be thought of as modelling a “has a” relationship

- Another way of looking at it is that inheriting from `A` is purely an implementation detail of class `B`, invisible to the outside world

- Most of the time, you should prefer using a private member variable rather than private inheritance

Protected Inheritance

- When using protected inheritance, all `public` and `protected` members of the base class are accessible as `protected` members of the derived class.

- If anybody finds a use for protected inheritance, please let me know 🙂

Public Inheritance

- When using public inheritance, `public` members of the base class are accessible as `public` members of the derived class; `protected` members of the base class are accessible as `protected` members of the derived class

- Public inheritance models an “is a” relationship

- This is the “normal” kind of inheritance we’re used to thinking about in other languages

- When we talk about “inheritance” without qualification, we almost always mean public inheritance

Public inheritance example

```
struct Animal {
    void eat();
};

struct Mammal : public Animal {
    int get_num_legs();
};

struct Bird : public Animal {
    void flap_wings();
};

struct Dog : public Mammal {
    void say_woof();
};

int main()
{
    Dog d;

    d.say_woof(); // OK
    d.get_num_legs(); // OK -- Dog "is a" Mammal
    d.eat(); // OK -- Dog "is a" Animal

    d.flap_wings(); // ERROR -- Dog is not a Bird!
}
```

Base class construction and destruction

- If class B inherits from class A, then every instance of B *contains an instance* of class A. This “base class subobject” must be constructed and destroyed correctly.

- When constructing objects, base class constructors are called before derived class constructors

- When destroying objects, the order is reversed — derived class destructors are called, followed by base class destructors

- *Be very careful when calling virtual functions from within constructors*

Calling base class constructors

- From our derived class, we can specify a base class constructor to call using the member initialiser list. For example:

```cpp
class Base {
public:
    Base() = default; // default constructor

    Base(int, int); // Another constructor
};

class Derived : public Base {
public:
    Derived()
       : Base(3, 4) // Calls Base::Base(int, int)
    {}
};
```

- If you don't tell the compiler which base class constructor to use, it will (try to) use the default constructor

Base class references

- When we have an instance of a derived class, we can form a reference or pointer to its public base classes

- We can then call base class methods via this pointer or reference

- For example:

```
Dog d;

Animal& a = d;
a.eat();

Mammal* pm = &d;
pm->get_num_legs();
```

- If we call a *virtual function* via a base class pointer or reference, then the call will dispatch to the derived class implementation

- This is the basis behind polymorphism, which you'll hear more about later

Managing lifetimes via smart pointers

- When using polymorphic classes, we usually need to use dynamic allocation
- As we heard last time, smart pointers are key to managing dynamic lifetimes correctly
- We can use a `unique_ptr<Base>` to manage the lifetime of a Derived object
- Note that we still need `Base` to have a virtual destructor!

Managing lifetimes via smart pointers

```
struct Shape {
    virtual float get_area();
    virtual ~Shape() = default;
};

struct Circle : Shape {
    Circle(float radius) : r{radius} {}

    virtual float get_area() override {
        return M_PI * r * r
    };
};

struct Rectangle : Shape {
    Rectangle(float width, float height) : w{width}, h{height} {}

    virtual float get_area() override {
        return w * h;
    }
};

int main()
{
    std::unique_ptr<Shape> shape = std::make_unique<Circle>(3);

    std::cout << shape->get_area() << '\n'; // calls Circle::get_area()

    shape.reset(std::make_unique<Rectangle>(3, 4)); // shape now contains a Rectangle
}
```

Inheritance “gotchas”

- If a derived class contains a member function of the same name as a member function of the base class, then the base class member function is hidden
- This applies even if the member functions have different signatures!
- For example:

```
struct Base {
    void do_something(int i);
};

struct Derived : Base {
    void do_something();
};

int main() {
    Derived d;
    d.do_something(3); // Error -- no matching function call!
}
```

Inheritance “gotchas”

- We can avoid this by explicitly writing that we want to call the base class’s version, `Base::do_something(int)`:

```
int main() {
    Derived d;
    d.Base::do_something(3); // OK
}
```

- Alternatively, in the definition of Derived we can use a *using directive* to make the base class function accessible again:

```
struct Derived : Base {
    using Base::do_something;
    void do_something();
};

int main() {
    Derived d;
    d.do_something(3); // OK
}
```

Inheritance “gotchas”

- When using inheritance, we need to be careful about using types by value; this can lead to *slicing*:

```
Dog d;
Animal a = d; // Legal, but probably not correct
```

- Here, the variable `a` is copy-constructed from the `Animal` subobject inside `d`

- But only the `Animal` subobject has been copied! All information about `Dog` (and `Mammal`) has been *sliced* away

- This mostly occurs when passing objects to functions taking their arguments by value. Prefer to use reference arguments when dealing with polymorphic types.

Inheritance: guidelines

- Inheritance is one way of implementing polymorphism in C++. It is not the only tool available.
- Prefer composition over inheritance
- Use private inheritance rarely, only for special cases or to take advantage of EBO
- Avoid multiple inheritance except for pure interface classes
- Be wary of slicing — don’t take polymorphic types by value
- Use smart pointers to manage the lifetime of polymorphic objects

Exercise

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

It’s goodnight from me…