

C++ for Numerical Methods

Damon McDougall

University of Warwick, UK

Lecture 12: Polymorphism and Multiple Inheritance

- Revise *inheritance*;
- Discuss *polymorphism*;
- *Multiple inheritance* and the *diamond problem*.
- Project.

Inheritance (revision)

Inheritance allows us to build specialisations of classes, *inheriting* the functionality of the general class.

- *Base class*: general class.
- *Derived class*: specialisation of a general class.

Declare a derived class as:

```
class <derived-name> : <access> <base-name> {  
  
};
```

Assuming *public* inheritance base class members translate as:

- *public* members → *public*.
- *protected* members → *protected*.
- *private* members are **NOT** accessible to the derived class.

Inheritance and Constructors

Constructors exist to initialise the class members when a new instance is instantiated. However, we must be careful when using inheritance.

- A constructor initialises members of the class.
- The constructor of the **derived class** can only initialise members which the derived class can access.
- A derived class could never initialise *private* members of the **base class**.
- Thus, we must use the constructor of the base class to initialise those members.
- So, finally, the base class constructor should be called **before** initialising the members of the derived class.

Inheritance and Constructors

We could call the base class constructor from our derived class constructor implementation as:

```
<derived>::<derived> (<derived-parameter-list>) {  
    <base-name>::<base-name> (<base-parameter-list>);  
}
```

but a better way is to put this as the first item in the initialisation list:

```
<derived>::<derived> (<derived-parameter-list>)  
    : <base-name> (<base-parameter-list>),  
    <derived-init-list> {  
}
```

Therefore base class members are all initialised first, followed by any elements of the derived class, followed by any statements in the constructor.

Constructors: A Quick Example

Example: Inheritance and Constructors

```
class Polygon {
    public:
        Polygon(vector<Point> pts);
    private:
        vector<Point> mPoints;
};

class Triangle : public Polygon {
    public:
        Triangle(Point a, Point b, Point c);
};

Triangle::Triangle(Point a, Point b, Point c) {
    vector<Point> points(a, b, c);
    Polygon::Polygon(points);
};
```

Inheritance Example

To clarify things, let us look at an example. We will consider the classes used in the example last week:

- A **base class** called `Shape`.
- **derived classes** called `FilledShape` and `Triangle`.
- `Shape` stores a set of ordered nodes and a line colour.
- `FilledShape` adds to this a fill colour.
- `Triangle` specialises `Shape` to exactly 3 nodes and provides a calculation of area.

See example code on website!

Polymorphism

Polymorphism

Polymorphism of objects mean those objects may behave differently in different contexts.

- Consider a pointer to a derived class:

```
Triangle *T = new Triangle ();
```

- A pointer to a derived class is type-compatible with a pointer to its base class. So we can do:

```
Shape *S = T;
```

- We can still do

```
S -> printMessage ();
```

But what if we were to **change** the functionality of `printMessage()` in the derived class `Triangle`?

Virtual Class Members

Virtual function

A function declared as **virtual in a base class** is one which can be **redefined in a derived class**.

To declare a function as virtual it should be preceded with the `virtual` keyword.

- The base class implements the function as normal.
- The derived class can *reimplement* the function (with the same name and parameter list).

Virtual class functions and polymorphism

Example: Virtual functions

```
class Shape {
    virtual void setLineColour (string pColour) {
        mColour = pColour;
    }
};

class Triangle {
    void setLineColour (string pColour) {
        if (pColour != ``Red``) pColour = ``Black``;
        mColour = pColour;
    }
};
```

```
Triangle *T = new Triangle();
Shape     *S = new Shape(), *R = T;
R -> setLineColour("Red");
T -> setLineColour("Red");
S -> setLineColour("Red");
```

Virtual class functions and polymorphism

The choice of version to run does **not** depend on the datatype of the pointer, but on the datatype of the original instantiated object.

- S is a Shape **so** runs

`BaseClass::setLineColour()`.

- T is a Triangle **so** runs

`DerivedClass::setLineColour()`.

- R is **still** a Triangle, **so** runs

`DerivedClass::setLineColour()`, even though the pointer is a Shape.

However, note that we could not call a solely `Triangle` member function of triangle if our pointer is of type `Shape`.

Virtual destructors

It is considered good practice to always make the destructor virtual. Suppose we call `delete T;`:

- A is of type `Triangle`, so the `Triangle` destructor is called.
- C++ automatically calls the `Shape` destructor as part of the process.

Now suppose instead we called `delete C;`:

- If the `Shape` constructor is **not** virtual, the `delete` statement calls the `Shape` destructor only.
Members of `Triangle` may not be cleaned up!
- Making the destructor virtual resolves the issue, calling the `Triangle` destructor as required.

Pure virtual functions

In `Shape`, we have no `getArea()` function. So if we have pointer of type `Shape*`, even if it is a `Triangle`, we cannot call it since the `Shape` class contains no such member.

- Suppose it doesn't mean anything to implement `getArea()` in `Shape`.
- But every class derived from `Shape` should have area.

We could create a virtual function with an empty implementation but this would not make sense with an instance of `Shape`. The better option is to create a *pure virtual function*.

Pure virtual function

```
virtual <return-type> <name> (<parameter-list>) = 0;  
virtual double getArea() = 0;
```

Pure virtual functions

If we implement `getArea()` as a pure virtual function:

- There is no implementation in `Shape`.
- `Shape` is called an *abstract base class*.
- Because there is no implementation, we cannot instantiate an object of class `Shape`.

Abstract base classes are useful for creating functionality which by itself is meaningless, but which can be bolted onto other classes.

See example code!

Multiple Inheritance

Multiple Inheritance

C++ allows a class to derive from more than one class. This is called multiple inheritance.

Suppose we have another class, `FilledTriangle`.

```
class FilledTriangle : public Triangle,  
                      public FilledShape {  
  
};
```

However, there are a few things to be careful of:

- If two or more base classes have a member with the same name, this is fine. Their scope is different.
- However, if you wish to use one of those members, you must identify which one with its *scope*.
- Use constructors correctly.
- The *Diamond Problem*.

Constructors and Multiple Inheritance

In the same way as we called the base class constructor from the initialisation list, we do the same for multiple inheritance. For our `FilledTriangle` class:

Constructors and Multiple Inheritance

```
FilledTriangle::FilledTriangle :  
    Triangle (), FilledShape () {  
    ...  
}
```

The two items in the initialisation list initialise the members of the base classes before the derived class is initialised.

The “Diamond Problem”

The previous example had a problem: both `Triangle` and `FilledShape` were derived from the `Shape` base class.

- Referencing members of `Triangle` or `FilledShape` was fine since they were unique with respect to scope.
 - However, C++ follows independent inheritance routes and so referencing a member of `Shape` causes a problem: Does the compiler want
 - `Shape::<member>` via `Triangle`
- OR**
- `Shape::<member>` via `FilledShape`

This is known as the **Diamond Problem**. The solution is to use *virtual inheritance* of `Shape` so the compiler takes care to create implementations for `Shape` members only once.

See example code!

This session you should aim to:

- Implement and build an implicit PDE solver.
- Modify your basic PDE solver to solve the heat equation using the *theta* method. This means that by setting the value of the variable θ , we can choose the weighting of the implicit and explicit methods, where

$$\theta = \begin{cases} 0, & \text{explicit Euler} \\ 1, & \text{implicit Euler} \\ \frac{1}{2}, & \text{Crank-Nicolson} \end{cases}$$