More sophisticated classes

Inlining member functions

- An `inline` function is copied by the compiler rather than called.
- Inline a member function by including the definition in the class declaration.
- Saves typing, but only use it when you really want to inline.

```
class Point {
public:
    double getX() const {
        return x;
    }
    // other members of Point
private:
    double x;
    double y;
};
```

The this keyword

- private variables are good for information hiding.
- Write getters and setters as shown.
- this is a pointer to the current instance of the object.

```
class Point {
public:
    double getX() const {
        return x;
    }
    void setX( double x ) {
        this->x = x;
    }
    // other members of Point
private:
    double x;
    double y;
};
```

Another use of this

- `price` method needs a reference to an option.
- `*this` is a reference to an option.

```
double UpAndOutOption::price(
        const BlackScholesModel& model ) const {
    MonteCarloPricer pricer;
    return pricer.price( *this, model );
}
```

Inheritance

- Inheritance allows you to implement interfaces more easily.
- Removes repetition of methods.

```
class ContinuousTimeOption {
public:
    /*  Virtual destructor */
    virtual ~ContinuousTimeOption() {};
    /*  The maturity of the option */
    virtual double getMaturity() const = 0;
    /*  Calculate the payoff of the option given
        a history of prices */
    virtual double payoff(
        const std::vector<double>& stockPrices
        ) const = 0;
    /*  Is the option path dependent */
    virtual bool isPathDependent() const = 0;
};
```

A base class

```
class ContinuousTimeOptionBase :
        public ContinuousTimeOption {
public:
    virtual ~ContinuousTimeOptionBase() {}
    double getMaturity() const {
        return maturity;
    }
    void setMaturity( double maturity ) {
        this->maturity = maturity;
    }
    double getStrike() const {
        return strike;
    }
    void setStrike( double strike ) {
        this->strike = strike;
    }
    //... more methods ...
private:
    double maturity;
    double strike;
};
```

- Base class provides basic implementations of boring methods common to most options.
- The base class has a virtual destructor. Any class used as a base class must have a virtual destructor.

Extending the base class

```
class PutOption : public ContinuousTimeOptionBase {
public:

    /*  Calculate the payoff of the option given
        a history of prices */
    double payoff(
        const std::vector<double>& stockPrices
        ) const;

    double price( const BlackScholesModel& bsm )
        const;

    bool isPathDependent() const {
        return false;
    };
};
```

- The PutOption extends the ContinuousTimeOptionBase. Same notation as used to implement an interface.
- It inherits the functions defined by this class.
- It inherits the variables `strike` and `maturity`.
- It inherits the interface `ContinuousTimeOption`.
- There is no need to write new `getMaturity` and `getStrike` functions.

The payoff is that it is now easy to write:

- `CallOption`,
- `DigitalCallOption`,
- `DigitalPutOption`,
- `UpAndOutOption`.

Terminology

- `ContinuousTimeOptionBase` is termed a *superclass* or a *parent class* of `PutOption`.
- `PutOption` is termed a *subclass* or a *child class* of `ContinuousTimeOptionBase`.
- `PutOption` *extends* from `ContinuousTimeOptionBase`.
- `PutOption` *inherits* from `ContinuousTimeOptionBase`.

Overriding methods

We give our base class a method to price the option:

```
class ContinuousTimeOptionBase
   : public ContinuousTimeOption {
public:
    /*  Price the option, by Monte Carlo or otherwise */
    double price(
        const BlackScholesModel& model ) const;
    // ... other members ...
};
```

Implement it using Monte Carlo:

```
double ContinuousTimeOptionBase::price(
        const BlackScholesModel& model ) const {
    MonteCarloPricer pricer;
    return pricer.price( *this, model );
}
```

Overriding methods continued

We don't want to use Monte Carlo for put options. Add the keyword `virtual` to the declaration of `price`.

```
class ContinuousTimeOptionBase
   : public ContinuousTimeOption {
public:
    /*  Price the option, by Monte Carlo or otherwise */
    virtual double price(
        const BlackScholesModel& model ) const;
    // ... other members ...
};
```

Overriding methods continued

We can now *override* the method in a subclass.

```
class PutOption : public ContinuousTimeOptionBase {
public:
    double price( const BlackScholesModel& bsm )
        const override;
    // ... other members ...
};
```

- The keyword `override` is optional.
- The parameter and return types must be identical including the `const` and `&` characters.

The keyword virtual

- `virtual` means *may* be overridden.
- in an interface no functions have definitions so all *must* be overidden. Therefore they must be virtual.
- All classes have a destructor. This must be declared as `virtual` in classes that are subclassed so that the correct destructor is called.
- Any class that is designed to be subclassed must have a virtual destructor.

Abstract Functions

- We say that a function has no implementation by writing =0.
- Such a function must be virtual.
- This is called an abstract function.
- Interfaces are classes where all functions are abstract.
- An abstract class is a class with at least one abstract function. For example `ContinuousTimeOptionBase` has an abstract `payoff` function.

Multiple layers

- You can build complex hierarchies of classes.
- `PutOption` has parent `ContinuousTimeOptionBase` and grandparent `ContinuousTimeOption`.
- We should insert a `PathIndependentOption` into our hierarchy:

```
class PathIndependentOption :
        public ContinuousTimeOptionBase {
public:
    /*  A virtual destructor */
    virtual ~PathIndependentOption() {}
    /*  Returns the payoff at maturity */
    virtual double payoff( double endStockPrice) const
        = 0;
    /*  Compute the payoff from a price path */
    double payoff(
            const std::vector<double>& stockPrices ) const {
        return payoff(stockPrices.back());
    }
    /*  Is the option path-dependent? */
    bool isPathDependent() const {
        return false;
    };
};
```

PathIndepdendentOption

`PathIndependentOption` does the following.

- It provides an implementation of `isPathDependent`.
- It has an abstract function to compute the payoff given only the final stock price.
- This means we can implement the `payoff` function that takes an entire path of stock prices.

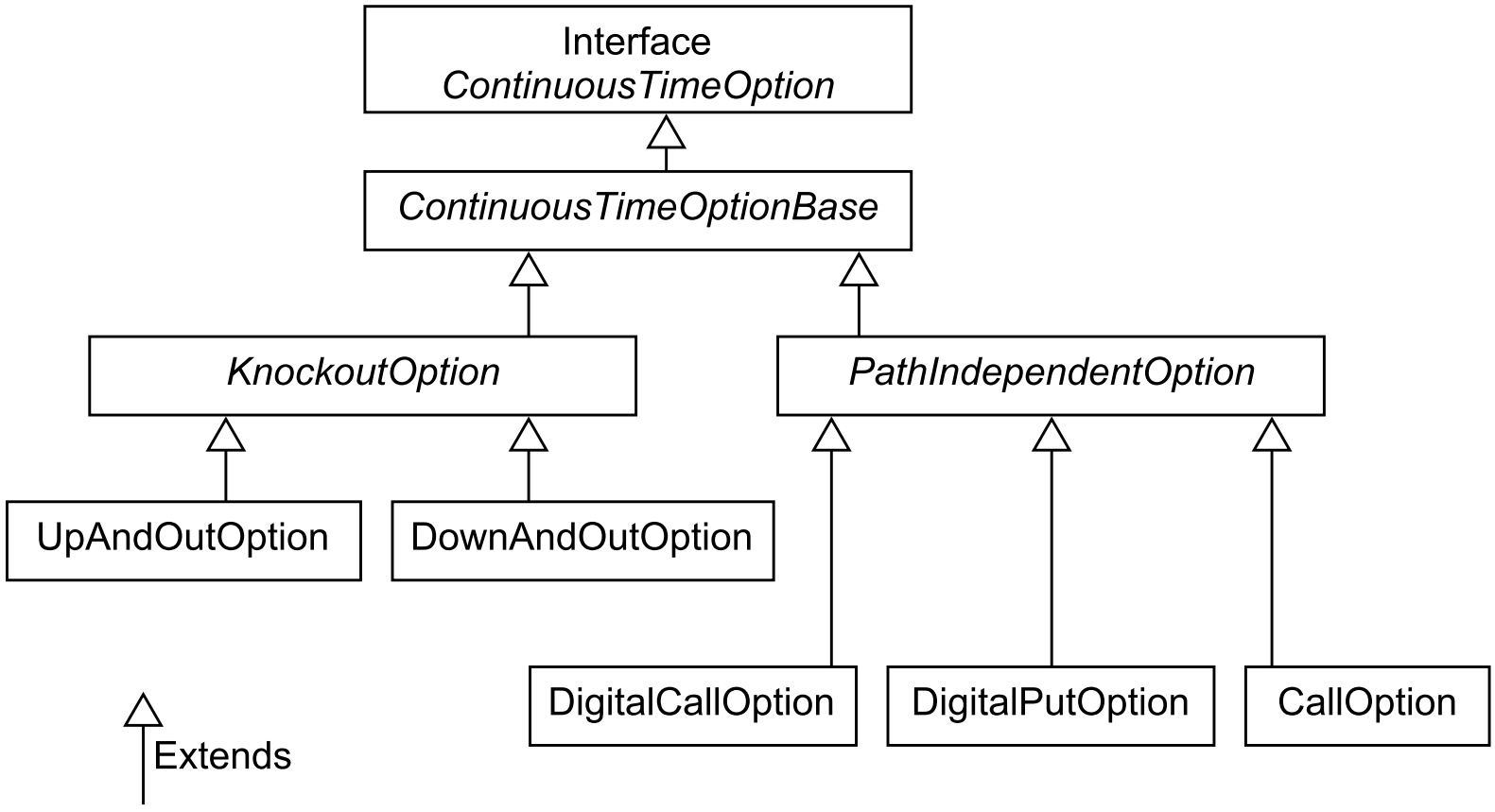
Extending PathIndependentOption

```
class CallOption : public PathIndependentOption {
public:

    double payoff( double stockAtMaturity ) const;

    double price( const BlackScholesModel& bsm )
        const;
};
```

- We must override the abstract function `payoff`.
- We choose to override the function `price` with a more efficient version.

UML

A UML diagram shows our option hierarchy:



Another hierarchy

- A class `Shape` that represents any finite shape in the plane. It has the following methods:
  - A method `area` to compute the area.
  - A method `contains` to test if a point is in the shape.
  - A method `boundingRectangle` that returns a `Rectangle` containing the entire shape.
- The class `Circle` is one implementation of `Shape`.
- The class `Rectangle` is another implementation of `Shape`.
- The class `HyperCircle` (the shape $x^4 + y^4 < 1$) is another implementation of `Shape`.
- The class `Shape` has a default implementation for `area` that uses Monte Carlo.
- `Circle` and `Rectangle` override `area`.

Discussion

- A graphics library where you couldn't write your own Shape classes would be pretty useless
- A pricing library where you can't write your own options would similarly be useless
- Object-oriented programming makes our library pluggable.

Multiplie inheritance

- It is possible to extend more than one class, but the rules are complex.
- Recommended that you only extend one normal class, but you may extend multiple interfaces.

```
class DerivativeWithStrike {
public:
    ~DerivativeWithStrike();
    virtual double getStrike() const = 0;
};
```

Inheriting from two parents

```
class ContinuousTimeOptionBase :
    public ContinuousTimeOption,
    public DerivativeWithStrike {
```

Calling superclass methods

- Sometimes you want to call a superclass's implementation of a function
- e.g. an UpAndOutOption overrides `price` to check if the stock price is over the barrier. If it is, return; otherwise, use superclass's method.

```
double price(
    const BlackScholesModel& model) const {
    if (model.stockPrice >= getBarrier())
        return 0;
    return KnockoutOption::price(model);
}
```

Forward declarations

- A `Shape` has a function which returns a `Rectangle`.
- But `Rectangle` extends `Shape`.
- Solution is called a forward declaration.

```
class CartesianPoint;
class Rectangle;

class Shape {
public:
    /*  Does the point lie in the shape */
    virtual bool contains( const CartesianPoint& point )
        const = 0;
    /*  A rectangle bounding the shape */
    virtual Rectangle boundingRectangle() const = 0;
    /*  By default area is computed by Monte Carlo */
    virtual double area() const;
};
```

Static variables and functions

```
class CallCountedSin : public RealFunction {
public:
    static int getNumberOfCalls();
    double evaluate( double x ) {
        numCalls++;
        return sin(x);
    }
private:
    static int numCalls;
};
```

Need to initialize the static variable.

```
int CallCountedSin::numCalls = 0;
```

- Static variables are global variables shared by all instances.
- To work with static variables you often use static functions.

```
int CallCountedSin::getNumberOfCalls() {
    return numCalls;
}
```

We've seen a static variable in `mt19337`.

```
mersenneTwister.seed(mt19937::default_seed);
```

Advantages over global variables and functions:

- static variables and functions can use private data;
- they are organised by class.

Protected

- `public` means everyone can see the member.
- `private` means accessible by your class only.
- `protected` means accessible by subclasses.

Summary

- Write getters and setters and use `private` data where possible.
- You can inline functions by writing the definition in the class.
- The `this` pointer makes writing setters easy. You can use it if you need a reference to the current instance.
- Build hierarchies of classes in order to inherit functionality.
- Use the `virtual` keyword to mean that a method can be overridden.
- Write `=0` to mean that a function has no implementation and so create an abstract class.
- Interface classes are a special case of inheritance.
- Use forward declarations to deal with circular class declarations.
- Use `static` variables in classes instead of global variables. Use `static` functions in classes to write functions that are associated with a class in general, rather than any particular instance of the class.