Interfaces

The need for interfaces

- The code for pricing a `CallOption` and the code for pricing a `PutOption` by Monte Carlo is identical apart from the specification of the type of the option.

The fantasy code

```
double MonteCarloPricer::price(
        const PathIndependentOption& option,
        const BlackScholesModel& model ) {
    double total = 0.0;
    for (int i=0; i<nScenarios; i++) {
        vector<double> path= model.
            generateRiskNeutralPricePath(
                option.getMaturity(),
                1 );
        double stockPrice = path.back();
        double payoff = option.payoff( stockPrice );
        total+= payoff;
    }
    double mean = total/nScenarios;
    double r = model.riskFreeRate;
    double T = option.getMaturity() - model.date;
    return exp(-r*T)*mean;
}
```

Explanation of the concepts

- Our code to price a `CallOption` only generates prices with one time step, so it is only valid for path-independent options.
- Wouldn't it be nice if we could price all path-independent options with the same code?
- Given that C++ requires us to specify the type of all parameters, we need to define a type that represents a general path-independent option.

Other examples

- An `ostream` is a general type.
  - We can write to a general `ostream`.
  - An `ostream` might be a file, a printer or the Internet.
- In the real world, a car is a general type.
  - We can steer to the right, we can steer to the left.
  - There are lots of makes of car, but we use them all the same way.
- In a computer game a monster is a general type.
  - We can draw a monster on screen.
  - We can ask it where it will move next.
  - Goblins and elves behave quite differently, but we call the same functions to paint them on the screen and to decide their next moves.
- In a trading simulation, a strategy is a general type.
  - We can use the strategy to determine what to buy.
  - There are lots of strategies, but a simulator should cope with all strategies.

The idea of an interface

- We need to separate interface from implementation.
- The interface defines how you can interact with an object.
- It defines a contract that must be fulfilled by the function

Example

A path-independent option must implement two functions:

- `double payoff( double finalStockPrice );`
- `double getMaturity();` This defines the interface of a path-independent option.

An interface class in C++

Example

A path-independent option must implement two functions:

- `double payoff( double finalStockPrice );`
- `double getMaturity();`

C++ equivalent is:

```
class PathIndependentOption {
public:
    /*  A virtual destructor */
    virtual ~PathIndependentOption() {}
    /*  Returns the payoff at maturity */
    virtual double payoff(
        double finalStockPrice) const = 0;
    /*  Returns the maturity of the option */
    virtual double getMaturity() const
        = 0;
};
```

How to write an interface

- Come up with a name for your class that describes the interface clearly. For example `PathIndependentOption`, `Monster` or `ostream`.
- Write a class declaration that contains all the functions you would like to present in the class.
- Add the keyword `virtual` at the front of every function declaration.
- Add the text =0 before the semi-colon at the end of the declaration.
- Add the mystical text

```
    virtual ~CLASS_NAME() {}
```

to the list of declarations.

What is this voodoo?

- The =0 means "this function is not implemented directly". This is the norm for interfaces. There is no general function you can write that will compute the `payoff` of all options, so that is why we do not implement it directly.
- We'll discuss *inheritance* later in the course and will see how to use the word `virtual` more generally. This will explain why we need to use it here.
- The mystical text:
  `virtual ~CLASS_NAME() {}`
  is called a virtual destructor. We'll explain the need for this when we discuss destructors later in the course.
- I'm teaching you how to do the right thing without explaining it fully. This is because it's much easier to learn what you should type than why you should type it!

How to implement an interface

- To specify that you implement an interface use the following construction:

```
class CallOption : public PathIndependentOption {
```

- This means that a `CallOption` *is a* `PathIndependentOption`. You can now pass a reference to a `CallOption` to any function that simply requires a reference to a `PathIndepdendentOption`.
- A `PutOption` is a path-independent option too, so we change the declaration of `PutOption` too.
- Our `MonteCarloPricer` now works equally well with put options and call options.

The general syntax

```
class CLASS_NAME : public INTERFACE_NAME {
... declarations for CLASS_NAME ...
};
```

- You will need to provide a new declaration and definition for every function defined in the interface `INTERFACE_NAME`.
- They must have exactly the same types, `const` declarations and so forth.

What has this bought us?

- You can now price ANY path-independent option using the `MonteCarloPricer`. This includes options you've never thought of before.
- The `MonteCarloPricer` code will remain unchanged even when you think of a new option you want to price.
- This makes the `MonteCarloPricer` *pluggable*. You can extend its functionality by simply plugging in new options. You don't need to rewrite the code.
- More generally, the use of interfaces means you can extend a trading system with new types of financial contract, without having to retest the whole thing.
- Notice that electrical plugs are "pluggable" precisely because they have a well-defined interface. You can use the same electricity for dishwashers, lamps and televisions!

Another example

- It is easy to write a function that integrates $\exp -x^2$ from a to b using the rectangle rule. But how can we write a function that can integrate any real-valued function from a to b using the rectangle rule?
- Here is our fantasy code:

```
double integral( RealFunction& f,
                 double a,
                 double b,
                 int nPoints ) {
    double h = (b-a)/nPoints;
    double x = a + 0.5*h;
    double total = 0.0;
    for (int i=0; i<nPoints; i++) {
        double y = f.evaluate(x);
        total+=y;
        x+=h;
    }
    return h*total;
```

RealFunction

- I've introduced a type called `RealFunction`. This name describes the kinds of object I can integrate.
- All we need from a `RealFunction` is that it can compute a value at a given point $x \in \mathbb{R}$. We define the interface by requiring that it has a function `evaluate` that computes the desired value.

```
class RealFunction {
public:
    /*  A virtual destructor */
    virtual ~RealFunction() {};
    /*  This method is abstract, there is
        no definition */
    virtual double evaluate( double x ) = 0;
};
```

- We've followed all the voodoo rules about `virtual` and =0.

Implement RealFunction

- Every time we want to integrate a real-valued function we need to write an appropriate implementation class.

```
class SinFunction : public RealFunction {
    double evaluate( double x );
};

double SinFunction::evaluate( double x ) {
    return sin(x);
}
```

```
static void testIntegral() {
    SinFunction integrand;
    double actual = integral( integrand, 1, 3, 1000 );
    double expected = -cos(3.0)+cos(1.0);
    ASSERT_APPROX_EQUAL( actual, expected, 0.000001);
}
```

Local classes

```
static void testIntegralVersion2() {

    class Sin : public RealFunction {
    public:
        double evaluate( double x ) {
            return sin(x);
        }
    };

    Sin integrand;
    double actual = integral(integrand, 1, 3, 1000 );
    double expected = -cos(3.0)+cos(1.0);
    ASSERT_APPROX_EQUAL(actual, expected, 0.000001);
}
```

Local classes

- You can define a “throwaway” class inside a function.
- Notice that the definition is provided for each member function not just the declaration.
- Java and C# fans will be disappointed to learn that you can't access local variables of the containing function.

How to spot when an interface is needed 1

Do you have an ever growing list of parameters to your function to handle different cases. For example if your function takes:

- A strike,
- A maturity,
- A barrier level,
- A `bool` specifying whether this is a knock-out or a knock-in option,
- A `bool` specifying whether this is a put or a call,
- A `bool` specifying whether this is a digital or vanilla option,
- A `bool` specifying whether this is a European or an American option
- ...

this is a clue that you are missing an interface. Lots of `bool` parameters are a particularly strong clue.

How to spot when an interface is needed 2

Do you keep changing the same function? For example, if every time a trader comes up with a new investment strategy you find you have to rewrite the function `backTestInvestmentStrategy` this is a sign that you should introduce an in interface.
Perhaps you need to design classes `MarketData` and `Portfolio` and a general interface for strategies like this:

```
class InvestmentStrategy {
    virtual ~InvestmentStrategy() {};
    virtual Portfolio rebalancePortfolio(
        const MarketData& marketData,
        const Portfolio& oldPortfolio ) =0;
};
```

How to spot when an interface is needed 3

- Whenever you need to know the *type* of the data to solve the problem. E.g., I can only price an option if I know *what type* of option it is.
- This is why C++ makes such a big deal of the type of all parameters. By specifying types, we are also specifying behaviour.
- Whenever you find yourself using a `switch` statement or an enormous `if`, `else if`, `else if ...`statement.
- Whenever you feel you want to pass functionality around. For example if you want the user to be able to select:
  - The algorithm to use—for example the random-number generation algorithm.
  - The methodology to use to solve a problem—for example, should we measure risk using `VAR` or `CVAR`?
  - The action to perform—for example, what should we do when we discover an arbitrage opportunity, alert the trader or just go ahead and trade?

How to spot when an interface is needed 4

- Experience!
  - There are many books available on object-oriented *design patterns*.
  - A *design pattern* is a technique someone has used in the past to successfully solve a problem.
  - By becoming familiar with design patterns, you can use the same ideas in your own code.
  - For example, we have seen a design pattern to write a generic `integral` function. Could you write a generic `differentiate` function?
  - Learning design patterns is a way of consciously learning from other people's experience.

Summary

- By defining interfaces we can write code that will work with inputs we haven't even thought of yet.
- This is essential to writing real code for the financial industry where new products are invented every day.
- Interfaces are common in every day life: for example cars, doors, guitars and plug sockets all give familiar examples of interfaces.
- Learning how to use interfaces is probably the most important skill in object-oriented design.