User-defined types

- We learned how to use
  - `vector<double>`,
  - `string`,
  - `fstream`,
  - `stringstream`.

  Using these new types has made as much more productive C++ programmers.
- Writing our own types is the next step.
- The main job of a programmer in an object-oriented language is writing their own types.

Class, object, instance

We already know the following:

- Every variable in C++ has a type.
- You can have lots of variables of a given type.

In the language of object-oriented programming we say:

- Every object in C++ has a class.
- You can have lots of instances of the same class.

Example

```
string s("To be, or not to be?");
```

In the code above `s` is an *object* of type `string`. It is an *instance* of the *class* `string`.

What is an object?

- An object is a bundle of data and functions to help with working with that data.
- The type of data and functions supported by an object depend only on its *class*. All instances of the same class have the same functions and the same types of data.

Example

An instance of the class `string` consists of:

- Data consisting of a sequence of characters.
- Functions such as `size`, `insert`, `erase` for working with this data.
- The *class* `string` is where all these functions and data types are defined.

An example class

Let us represent a point in two dimensions using a class.

```
class CartesianPoint {
public:
    double x;
    double y;
};
```

- This is called a `class` declaration.
- We want this class to be available to users of our library so it is declared in the `.h` file.
- The name of our class is `CartesianPoint`
- A Cartesian point contains two `double` values called `x` and `y`.
- Users of the library are allowed to use the values of `x` and `y` in their own code, so these are marked as `public`.

Using the class

```
CartesianPoint p;
p.x = 100;
p.y = 150;
cout << "Coordinates (";
cout << p.x ;
cout << ", ";
cout << p.y ;
cout << ")\n";

p.x *= 2;
p.y *= 2;
cout << "Rescaled cordinates (";
cout << p.x ;
cout << ", ";
cout << p.y ;
cout << ")\n";
```

- The name of the class is the name used when you want to create objects of the class.
- You use a dot `.` to access the data inside your class.
- You can only access the variables because they are `public:`. Delete the word `public:` and see what happens.

The syntax for declaring classes

```
class CLASS_NAME {
public:
    DATA AND FUNCTION DECLARATIONS
private:
    DATA AND FUNCTION DECLARATIONS
};
```

- You can define classes in header files or cpp files. Use header files if you want users of the library to use your class.
- You must remember the semi-colon at the end. You'll often get very confusing errors if you omit it.
- Like variable names, class names should contain no spaces or strange characters and should start with a letter. I recommend using camel case starting with a capital letter.
- We'll discuss `public` and `private` later.

Another example class

```
class PolarPoint {
public:
    double r;
    double theta;
};
```

Writing a function that uses our classes:

```
CartesianPoint polarToCartesian( const PolarPoint& p ) {
    CartesianPoint c;
    c.x = p.r*cos( p.theta );
    c.y = p.r*sin( p.theta );
    return c;
}
```

- CartesianPoints are considered big objects, so we pass them by reference for efficiency.
- We use the `const` keyword to ensure that the function isn't allowed to change the `r` and `theta` coordinates.

A test for both functions

```
static void testPolarToCartesian() {
    PolarPoint p;
    p.r = 2.0;
    p.theta = PI/2;
    CartesianPoint c = polarToCartesian( p );
    ASSERT_APPROX_EQUAL( c.x,0.0,0.001 );
    ASSERT_APPROX_EQUAL( c.y,2.0,0.001 );
}
```

- If you compare this against our previous test code, you will see that it is marginally simpler.
- Against this the code for the functions is marginally longer.
- When designing libraries, choose what is best for the user of the library, not the author of the library.
- In this case, the version with classes is better.

What have classes given us here?

- We have established a clear convention that x-coordinates are called `x` and y-coordinates are called `y`. This will make code written using CartesianPoint clearer than code that uses `x` some of the time, `X` some of the time, and `xcoord` the rest of the time.
- We've established similar conventions for polar coordinates.
- We are able to return multiple values without using pass by reference.
- We have made it easier for users of our library to work with Cartesian and polar coordinates. Making life easier for our users is our main aim when writing a library.

Adding functions to classes

```
class Circle {
public:
    double radius;
    double area();
    double circumference();
};
```

- The circle class comes with two functions you can call:
- `area()` returns the area of the circle,
- `circumference()` returns the circumference of the circle.
- This declares the functions, we'll have to define them in a C++ file.

Using the functions

Before we implement these functions, let's be test-driven developers and write tests for them.

```
static void testAreaOfCircle() {
    Circle c;
    c.radius = 4;
    ASSERT_APPROX_EQUAL( c.area(), 16*PI, 0.01 );
}

static void testCircumferenceOfCircle() {
    Circle c;
    c.radius = 2;
    ASSERT_APPROX_EQUAL( c.circumference(), 4*PI, 0.01 );
}
```

- You use a dot to call a function on a particular object instance. For example `c.area()` computes the area of the circle `c`.
- You don't need to pass the radius to the function area the circle instance "knows" its own area.

Defining the functions

As usual for C++ functions, we write the definitions in a `.cpp` file.

```
double Circle::area() {
    return PI*radius*radius;
}
```

```
double Circle::circumference() {
    return 2*PI*radius;
}
```

- When you define a function for a class, you must always specify the name of the class in the definition. The syntax is: `CLASS_NAME::FUNCTION_NAME`. If you forget to do this you'll get linker errors saying you've forgotten to define the function.
- The `Circle` functions are able to access the *member variable* `radius` and the *global variable* `PI`.

Member variables and functions

- A *member variable* of a class is a variable defined in the class.
- A *member function* of a class is a function defined in the class.
- You can access member variables and functions when writing function definitions.
- You must specify the class name using `::` when writing the definition of a member function.

The const keyword

If a member function doesn't change the object, you should mark it as const. You do this by writing const at the end of the declaration and definition.

```
class Circle {
public:
    double radius;
    double area() const;
    double circumference() const;
};

/*
 *   Computes the area of a circle
 */
double Circle::area() const {
    return PI*radius*radius;
}
```

A Black–Scholes Model class

```
class BlackScholesModel {
public:
    double stockPrice;
    double volatility;
    double riskFreeRate;
    double date;
};
```

- Notice that this class only contains the variables associated with the model.
- We'll specify different option contracts in different classes
- We might add functions to do things like generate simulated option prices.
- We're specifying the date as a `double`. You wouldn't do this in real code, but I don't want to waste time on thinking about calendars. We'll measure dates in years since 0 A.D.. So January the first 2014 would be represented as 2014.0.

A Call Option class

```
class CallOption {
public:
    double strike;
    double maturity;

    double payoff( double stockAtMaturity ) const;

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

- The call option has a strike and maturity, but knows nothing about the current market data. This means a `CallOption` object will remain unchanged as the market changes.
- It has a function to compute the payoff at maturity.
- It has a function to compute the price given some hypothetical market data in the form of a `BlackScholesModel`.
- Note the `const` statements.

The definition of payoff

```
double CallOption::payoff(
         double stockAtMaturity ) const {
    if (stockAtMaturity>strike) {
         return stockAtMaturity-strike;
    } else {
         return 0.0;
    }
}
```

- All the const keywords exactly match the declaration.
- Calling payoff doesn't change the option in anyway. This is why it is declared const.

The definition of price

This computes the price using the Black–Scholes formula.

```
double CallOption::price(
        const BlackScholesModel& bsm ) const {
    double S = bsm.stockPrice;
    double K = strike;
    double sigma = bsm.volatility;
    double r = bsm.riskFreeRate;
    double T = maturity - bsm.date;

    double numerator =
        log( S/K ) + ( r + sigma*sigma*0.5)*T;
    double denominator = sigma * sqrt(T );
    double d1 = numerator/denominator;
    double d2 = d1 - denominator;
    return S*normcdf(d1) - exp(-r*T)*K*normcdf(d2);
}
```

- `price` takes as parameter a `BlackScholesModel` called `m`. This is passed by constant reference. It is passed by reference because this is more efficient than pass by value. It is passed as a `const` reference because `price` doesn't change its value.
- Calling `price` doesn't change the option contract, so the function declaration ends with `const;`. Correspondingly we have the `const` appearing before `{` in the definition.
- The use of `const` and pass by reference must exactly match in the declaration and definition.

Using the CallOption class

Our unit tests provides a helpful example of usage:

```
static void testCallOptionPrice() {
    CallOption callOption;
    callOption.strike = 105.0;
    callOption.maturity = 2.0;

    BlackScholesModel bsm;
    bsm.date = 1.0;
    bsm.volatility = 0.1;
    bsm.riskFreeRate = 0.05;
    bsm.stockPrice = 100.0;

    double price = callOption.price( bsm );
    ASSERT_APPROX_EQUAL( price, 4.046, 0.01);
}
```

What has using classes bought us (so far)?

- Easier programming. If we have a single function `blackScholesPrice` that takes 5 double parameters, it is almost impossible to remember what the correct order for the parameters is.
- Easier debugging. If you have a function that takes 5 double parameters, its almost impossible to spot if someone has accidentally put the parameters in the wrong order.
- Consistency. If we use the same `BlackScholesModel` class to price put options, Asian options, knock-out options etc. we'll have a library that is much easier to use.

Recommended programming conventions

- Whenever possible, don't put class declarations in header files. Put them in cpp files. You should try to hide information if possible.
- If you decide to put a class in a header file, define only one class in each header file.
- Name that class the same as the header file.
- Give classes names that are nouns: for example `CartesianPoint` or `BlackScholesModel`.
- Use upper case for the first letter of a class name.

The files `BlackScholesModel.h` and `CallOption.h` demonstrate these conventions. The file `geometry.h` breaks them.

The `const` keyword

- The `const` keyword is an optional feature in C++. You can choose to use it if you want the compiler to guarantee that you never accidentally change an object when you didn't mean to.
- To use the `const` keyword at all, you need to use it *everywhere*.
- Deciding whether to use it is a software architecture question: it affects the whole project. In practice, this means you'll need to ask your boss whether she wants you to use it or not.
- If you are writing a library, you'll need to use the `const` keyword if your library users want to use the `const` keyword.

A more sophisticated example

```
class PieChart {
public:
    void setTitle( const std::string& title );
    void addEntry( const std::string& label,
                   double value );
    void writeAsHTML( std::ostream& out ) const;
    void writeAsHTML( const std::string& file ) const;
private:
    std::string title;
    std::vector<std::string> labels;
    std::vector<double> values;
};
```

Private members

- The data variables are declared as `private`. This means only member functions of `PieChart` can see those variables.
- This is considered good programming style.
  - It means it is *impossible* for `labels` and `values` to contain a different number of elements.
  - It means you can change the implementation details (i.e. how data is stored) without users of your class being affected in anyway.
- The `private` keyword allows you to perform more sophisticated information hiding than just choosing what to put in the header file.
- Not putting a class in a header file gives even more information hiding than making things `private`.
- You can make helper functions `private`, as well as making data `private`.

Encapsulation

- Encapsulation refers to two things:
  - The bundling together of related items into a single object.
  - Preventing direct messing with the internal data of an object.
- Encapsulation is common in real world design.
  - All the lighting controls of a car are put on the dashboard and are clearly separated from the controls for the windows and the seats.
  - You can control a car through standard functions (turn left, turn right etc.). The internal workings are hidden from the user completely.
- Just as these design principles make cars easier to use, so the same principles make objects in object-oriented programs easy to use.

Encapsulation Example

Example

How does the `vector<double>` class store it's data, given that it can't be using a `vector<double>` itself?

Solution

You don't need to know! Most of the time you don't care and your life is easier for not having to think about this.

Example

What are the internal data members of an `fostream`?

Solution

Who cares?

Using a Pie Chart

```
static void testPieChartClass() {
    // just checks that the class compiles etc.
    PieChart pieChart;
    pieChart.addEntry("Mushrooms",200);
    pieChart.addEntry("Salami",100);
    pieChart.addEntry("Spinach",150);
    pieChart.setTitle("Pizza Toppings");

    pieChart.writeAsHTML( "PizzaPie.html" );
}
```

- A key belief behind object orientation is that you should find this code very easy to understand.

Writing the Pie Chart implementation

The code to define the functions is very similar to the code we wrote before we introduced objects. You can see the full listings in `FMLib`. Here is an excerpt:

```
void PieChart::setTitle( const std::string& t ) {
    title = t;
}

void PieChart::addEntry( const string& label,
                         double value ) {
    labels.push_back( label );
    values.push_back( value );
}
```

- Functions like `writeTopBoilerPlateOfPieChart` are left almost unchanged. They're written as ordinary functions. We've just added a `title` parameter.
- Notice that when you write a member function's definition, you have convenient access to all the member variables and member functions.

Constructors

Initialization of local variables

- If you declare a `double` in C++ but do not set its value, C++ does not guarantee the value.
- In practice, the computer will just grab some free memory and use whatever values happen to be there.

```
int main() {
    double d;
    cout << "What is the value of d?\n";
    cout << d;
}
```

- The C++ standard doesn't say what this program should do.
- If we compile with warnings as errors, we get an error message.

Initialization of member variables

This will fail for similar reasons.

```
class Point {
public:
    double x;
    double y;
};

int main() {
   Point p;
   cout << "What is the value of x?\n";
   cout << p.x;
   return 0;
}
```

Constructors

- You can give your C++ classes *constructors*.
- A constructor performs the initialization of an object to leave it in a sensible state.
- To construct a vector of length 100 initialized with zeros:

```
vector<double> v(100,0.0);
```

- To construct a string with characters `Some text`.

```
string s("Some text");
```

Default Constructor

- Most classes have a default constructor that initializes the object in a sensible default state.
- To create an empty vector:

```
vector<double> v;
```

  The default constructor is automatically called.

- To create an empty string:

```
string s;
```

To write a default constructor

```
class Point {
public:
    Point(); // declare default constructor
    double x;
    double y;
};

Point::Point() {
    x=0.0;
    y=0.0;
}

int main() {
   Point p;
   cout << "What is the value of x?\n";
   cout << p.x;
   return 0;
}
```

Rules for writing a default constructor

- The constructor declaration looks like a function declaration except:
  - There is no return type.
  - There are no parameters.
  - Instead of the function name you have the name of the class.
- The constructor definition looks like a function definition except:
  - There is no return type.
  - There are no parameters.
  - Instead of the function name you have the name of the class.

How constructors work

- Think of a constructor as a function that is automatically called before anyone is allowed to see the object.
- Set all `int`, `double` etc. fields to sensible default values.
- You should ensure that the object is in a consistent state before anyone ever sees it. Perform whatever processing is required to achieve this.

Alternative syntax

```
class Point {
public:
    Point(); // declare default constructor
    double x;
    double y;
};

Point::Point() :
    x(0.0),
    y(0.0) {
}

int main() {
   Point p;
   cout << "What is the value of x?\n";
   cout << p.x;
   return 0;
}
```

Notes on the alternative syntax

```
Point::Point() :
    x(0.0),
    y(0.0) {
}
```

- The list of statements $x(0.0), y(0.0)$ is called an initialization list.
- Note that this comes after a colon character `:` and before the `{`.
- Use this to call constructors of all the fields inside your object.

Which syntax should you use?

- Experienced C++ programmers prefer an initialization list because it is faster.
- This is because calling the default constructor and then performing assignment will be slower than using the right value first time.

```
string myString;
myString="Some text";
```

is slower than

```
string myString("Some text");
```

Other constructors

```
class CallOption {
public:
    double strike;
    double maturity;
    CallOption(); // default constructor
    CallOption(double strike, double maturity);//alternative
};
// default constructor implementation
CallOption::CallOption() :
    strike(0.0),
    maturity(0.0) {
}
// alternative constructor implementation
CallOption::CallOption(
        double s,
        double m ) :
    strike(s),
    maturity(m) {
}
```

Designing with constructors

- To create an option with strike 100 and maturity 2.0:
  `CallOption myOption( 100, 2.0 );`
- This seems convenient, but this is probably not a very good design:
  - By introducing classes we got rid of the problem of having to remember what order to put parameters when calling functions.
  - By having a constructor with multiple parameters, we've reintroduced this problem!
- By contrast:
  - Being able to construct a `vector` of given size and default value *is* a good use of constructors.
  - Being able to construct a `string` with the desired text is also a good use of constructors.

Single parameter constructors

- `string` has a constructor that takes raw text data.

```
string s("Some raw text");
```

- C++ will automatically convert raw text data to strings.

```
plot( "myPlot.txt", xVec, yVec );
```

- This works even though the first parameter of a plot is declared to be a `const string&`.
- If you write a single parameter constructor, C++ will perform similar automatic conversions.

Sounds cool, but it's a terrible idea

- Suppose you had added a constructor to `BlackScholesModel` where you provide just the stock price:

```
class BlackScholesModel {
public:
    double stockPrice;
    double data;
    double volatility;
    double riskFreeRate;
    BlackScholesModel();
    BlackScholesModel( double stockPrice ); // key line
};
```

- C++ will now automatically convert doubles into `BlackScholesModel` instances.
- This is weird and undesirable.

The explicit keyword

- Mark constructors that take one parameter as `explicit`.

```
class BlackScholesModel {
public:
    double stockPrice;
    double data;
    double volatility;
    double riskFreeRate;
    BlackScholesModel();
    explicit BlackScholesModel( double stockPrice );
};
```

- This prevents automatic conversion.
- In the very rare event that automatic conversion is useful (as for strings) you can drop the `explicit`.
- If you forget the `explicit` it isn't the end of the world.

Summary

- We've learned how to write classes that group data together.
- We've learned how to add functions to our classes.
- We've seen that using classes makes life easier for users of our library.
- We've learned how to use the keywords `public` and `private` to hide data from users of our library.
- We've learned the buzzword "encapsulation".
- We've learned how to use the `const` keyword to indicate whether or not an object is changed by calling one of its member functions.
- We've learned how to give classes constructors.