

# C++ for Numerical Methods

Damon McDougall

University of Warwick, UK

Lecture 11: Friendship and Inheritance

# Friendship and Inheritance

This will be a rather more different lecture covering three disjoint but important topics:

- Header and source files;
- Friendship of functions and classes;
- Class inheritance.

# Problems of a single source file

As your project gets larger with multiple class definitions it becomes almost necessary to break the code into multiple files:

- A single source file becomes too large to manage and edit effectively.
- All classes are bundled into a single file - difficult to reuse the class elsewhere.
- The compiler must compile ALL the code every time.

The solution, then, is to split up the source code into multiple files.

# Split up the source file

Let's simply split up the source file. The easiest thing to do would be to store one class per file.

- Put the declaration and definition for each class into its own file.
- Name the file the same as the class.
- Leave just the `main()` routine in the original file.
- Use `#include "<filename>"` to include each class file before the `main()` routine.
- So, for example, to store two classes called `MyClassA` and `MyClassB`, we would have three files: `main.cpp`, `MyClassA.cpp` and `MyClassB.cpp`.

But there are still problems.

# Split up the source file

What are the issues?

- This solves the problem of large source files - but only from the editing perspective.
- Compiling this code takes just as long - e.g. compile time of hours after changing just one line in Microsoft Windows.
- A class cannot be distributed as object code (a company might want to protect the source code).
- Problems of multiple inclusion of the same functions.

# Header and Source files

The solution to all these problems is to split up each class file into two separate files:

- A **header** file which contains the class declaration.
- A **source** file which contains the implementation of the class.

So for example:

- Point.h:

```
class Point {  
    Point ();  
    ...  
};
```

- Point.cpp:

```
#include "Point.h"  
Point::Point () {  
    ...  
}
```

# Multiple inclusion

We need to do one last thing to protect against multiple inclusion of a header file. We wrap the entire contents of the header file in a *preprocessor conditional statement*, to ensure the contents is only included once.

```
#ifndef CLASS_POINT
#define CLASS_POINT

class Point {
    Point ();
    ...
};

#endif
```

**Note:** The preprocessor variable `CLASS_POINT` must be unique to that particular header file.

# Compilation of multiple files

Now that the implementation of each class is completely independent from the `main()` routine, we must tell the compiler about the new `.cpp` files we've created.

- The compiler will then compile each `.cpp` file into object code `.o`.
- A *linker* then puts all the `.o` files together with any other libraries to make the executable.
- If a single line in one of the `.cpp` files is changed, only that file needs to be recompiled.

**Note:** In Dev-C++ you should create a project in which you put your header and source files. It will then handle compiling all the files automatically.

# Forward Declarations

To reduce the number of chain-included files the compiler must process, classes may instead be declared by means of a **forward declaration**.

## Forward declaration

A forward declaration indicates that a particular class exists without describing its complete definition.

- We use this **instead** of including the relevant header file.
- It can only be used if the only reference to that class in the file is in the form of a *pointer* or *reference*.
- Almost always used in header files - implementation files should include the full header file.
- Can also be used to solve cross-dependency issues; e.g. if `ClassA` uses `ClassB` and vice versa.

# Forward Declarations

## Example: Forward Declarations

### Contents of **Line.h**:

```
class Point;           // Forward declaration
class Line {
    ...
    Point *p1;         // Only pointers
    Point *p2;
};
```

### Contents of **Line.cpp**:

```
#include "Point.h"    // Full header for class members
double Line::getLength() {
    double x = p2->getX() - p1->getX();
    double y = p2->getY() - p1->getY();
    return sqrt(x*x + y*y);
}
```

## Friendship

Friends to a class are functions or other classes which are permitted to directly access the private members of that class.

- This breaks the concept of data encapsulation.
- A friend is specified using the `friend` keyword.
- You should avoid using it 99% of the time!

# Friend Functions

- The friendship of a function to a class is specified in the class definition.
- As an example, casting your minds back to last week, recall that we defined an external operator overloading function:

```
bool operator==(double pVal, const Point& pSrc);
```

```
class Point {  
    ...  
    friend bool operator==(double pVal, const Point& pSrc);  
};
```

```
bool operator==(double pVal, const Point& pSrc) {  
    return (pVal == pSrc.mX)  
}
```

# Friend Classes

The friendship of a class to another class is similarly specified in the class definition. For example:

```
class Point {  
    ...  
    friend class Line;  
};
```

- Friendship is **not** transitive ( $A \rightarrow B, B \rightarrow C \not\Rightarrow A \rightarrow C$ ).
- There are very few circumstances where it is necessary to break the axiom of encapsulation and use `friends`.
- If you think you need to use `friends`, you're probably doing something wrong!
- One exception is for numerical efficiency - it is often better to access the arrays in another class directly.

# Inheritance

Inheritance allows the creation of classes which are derived from other classes - they become a specialisation of another class.

- The specialised class is called the *derived class*.
- The class from which it inherits is called the *base class*.

A derived class **extends** the functionality of a base class.

- All members of the base class which are `public` or `protected` are available in the derived class.
- Additional members can be added to the derived class to extend the functionality of the base class.

# Using Inheritance

## Example: Inheritance

Given a base class:

```
class BaseClass {  
    ...  
};
```

create a derived class from this as follows:

```
class DerivedClass : public BaseClass {  
    ...  
};
```

Now `DerivedClass` has inherited all of the `public` and `protected` members of `BaseClass`.

# Inheritance Access

## Inheritance

```
class <name> : <access-specifier> <basename> {  
  
};
```

The `<access-specifier>` has different implications for the access level of the members in the base class. There are only two options for `<access-specifier>` here:

- `public`: Members of the base class which were public remain publically accessible in the derived class.
- `private`: Members of the base class which were public become *private* in the derived class.

Usually, `public` is the sensible choice, but sometimes `private` is more appropriate if the derived class needs to manage access to the base class members.

# Inheritance Example

## Example: Polygons (regular)

```
class Polygon {
    public:
        Point getPoint(unsigned int vIdx);
    private:
        std::vector<Point> mCorners;
};

class Triangle : public Polygon {
    public:
        double getArea();
};
```

# Polygon close up

Let's look a little more closely at `Polygon` and `Triangle`.

- The base class, `Polygon`, represents a general polygon with a set of corners (stored as `private` data).
- The derived class, `Triangle`, inherits the storage framework provided by the `Polygon` class and adds some specialisation (in this case computing the area).
- Bizarrely, however, any new functions in `Triangle` won't have access to `mCorners` directly because they were declared as `private` inside `Polygon`.

# Protected Access

There is a slightly better way of designing the inheritance above - by giving `protected` access to the variable `mCorners`.

- `protected` access means the member is *private* in general.
- However, it is now available to derived classes of the `Polygon` class.
- From the perspective of the `Triangle` class.
- Protected access is *transitive*. A class derived from `Triangle` will still have access to `mCorners`.

Next week we will look at overriding virtual functions in the base class and multiple inheritance.

- Know how to declare a `friend` class.
- Understand the concepts of *inheritence* and deriving one class from another.
- Implement and build the `TriMatrix` class.
- Write a basic PDE solver to solve the heat equation in one dimension using the explicit Forward Euler time-stepping scheme.