C++ for Numerical Programming - lectures 7-8

Martin Robinson

2016

Consider the example of a linear solver the aims to solve the following linear system $Ax = b$. It would be useful we could write code in such a way that the variables A, x and b could be treated as *matrices* and *vectors*, rather than a generic `std::vector<double>`. We would like to have:

1. Internal variables representing, for example, the shape of each matrix/vector and its individual elements
2. An external interface that contains all of the operations necessary to solve the system (e.g. matrix-vector multiply)

This is possible, through the use of classes

A simple class

As an example we will develop a class of Premier League footballers Each footballer will have the following attributes:

- a first name;
- a surname;
- a club;
- a weekly wage;
- a weekly expenditure; and
- a number of appearances.

We will begin by writing a class with these attributes, and then develop the class further

This class may be coded as

```
#include <string>
class PremiershipFootballer
{
    public:
    std::string firstName, surname, club;
    int weeklyWage, weeklyExpenditure, appearances;
};
```

Don't worry about the term public – that will be explained later
Note the semicolon after the curly bracket at the end of the class

Save the code on the previous slide as
`PremiershipFootballer.hpp`
The class can then be used using the code on the following slide of variables in the line of code above

```cpp
#include <iostream>
#include "PremiershipFootballer.hpp"
int main()
{
    PremiershipFootballer wrooney;
    wrooney.firstName = "Wayne";
    wrooney.surname = "Rooney";
    wrooney.club = "Manchester United";
    wrooney.weeklyWage = 220000;
    wrooney.weeklyExpenditure = 4000;
    std::cout << "First name is "
              << wrooney.firstName << "\n";
    return 0;
}
```

Recall that we want to write functions that are associated only with the class

We will write a function `CalculateSpendingMoney` that takes an integer number of weeks as input and returns the amount of money that the footballer has available to spend during that period

We define the function inside the file

`PremiershipFootballer.hpp`

The body of the function is written inside another file

`PremiershipFootballer.cpp`

The files `PremiershipFootballer.hpp` and `PremiershipFootballer.cpp` are shown on the next two slides

Technically a function on an object is known as a *method* of the object

PremiershipFootballer.hpp

```
#include <string>
class PremiershipFootballer
{
public:
    std::string firstName, surname, club;
    int weeklyWage, weeklyExpenditure, appearances;
    int CalculateSpendingMoney(int numWeeks);
};
```

PremiershipFootballer.cpp

```
#include "PremiershipFootballer.hpp"
int PremiershipFootballer::
        CalculateSpendingMoney(int numWeeks)
{
    return (weeklyWage - weeklyExpenditure) * numWeeks;
}
```

Loosely speaking, a list of variables and functions is included in the `.hpp` file, and the functions are included in the `.cpp` file

On the previous slide note that the function written is associated with the class `PremiershipFootballer` through the statement

```
int PremiershipFootballer::
        CalculateSpendingMoney(int numWeeks)
```

This function may used outside the class by using statements such as

```
std::cout << wrooney.CalculateSpendingMoney(2) << "\n";
```

Some example code is given on the next slide

use class.cpp

```
#include <iostream>
#include "PremiershipFootballer.hpp"
int main()
{
    PremiershipFootballer wrooney;
    wrooney.firstName = "Wayne";
    wrooney.surname = "Rooney";
    wrooney.club = "Manchester United";
    wrooney.weeklyWage = 220000;
    wrooney.weeklyExpenditure = 4000;
    std::cout << "Two weeks spending money = "
        << wrooney.CalculateSpendingMoney(2) << "\n";
    return 0;
}
```

Compiling multiple files

Before we can compile the file `use_class.cpp` on the previous slide we first need to compile the `PremiershipFootballer` class. This is done by using the -c option when compiling:

```
$ g++ -Wall -O -c PremiershipFootballer.cpp
```

This produces an object file `PremiershipFootballer.o`. We can now compile use `class.cpp` by typing

```
$ g++ -Wall -O -o use_class use_class.cpp
                        PremiershipFootballer.o
```

The code may be run as before by typing

```
$ ./use_class
```

Using a Makefile

Suppose a code `UseClasses.cpp` uses two classes: `Class1` and `Class2`
It is easy to forget to compile the files `Class1.cpp` and `Class2.cpp` each time they are modified
Also, you only need to compile classes that have been changed since they were last compiled
Makefiles are a very efficient solution to these problems

Save the following code as Makefile

```
Class1.o : Class1.cpp Class1.hpp
    g++ -c -O Class1.cpp

Class2.o : Class2.cpp Class2.hpp
    g++ -c -O Class2.cpp

UseClasses : Class1.o Class2.o UseClasses.cpp
    g++ -O -o UseClasses Class1.o Class2.o UseClasses.cpp
```

The executable `UseClasses` may be created by typing

```
make UseClasses
```

When using a makefile, the line

```
UseClasses : Class1.o Class2.o UseClasses.cpp
```

means that the executable UseClasses depends on the files `Class1.o`, `Class2.o` and `UseClasses.cpp`
In turn, the line

```
Class1.o : Class1.cpp Class1.hpp
```

means that the object file `Class1.o` depends on the files `Class1.cpp` and `Class1.hpp`
The file `Class1.cpp` will only be compiled if `Class1.cpp` or `Class1.hpp` have changed since the last time `Class1.o` was created

Can use pattern rules and automatic variables to generalise Makefiles

```
%.o: %.cpp %.hpp
    g++ -c -O $<

UseClasses : Class1.o Class2.o UseClasses.cpp
    g++ -O -o $@ $^
```

Setting variables

We can use functions to set variables - this allows us to check that the values assigned are sensible. For example

```
void PremiershipFootballer::SetWeeklyWage(int w)
{
    if (w > 0 && w < 500000)
    {
        weeklyWage = w;
    }
    else
    {
    std::cout << "Error in setting weekly wage\n";
    }
}
```

First we add the function definition to the file `PremiershipFootballer.hpp` as we did with the function `CalculateSpendingMoney`:

```
void SetWeeklyWage(int w);
```

Then we add the function on the previous slide to the file `PremiershipFootballer.cpp`
This allows us to set the variable weeklyWage using statements such as

```
wrooney.SetWeeklyWage(220000);
```

Now we have written a function to set the variable `weeklyWage` that checks that it takes an appropriate value, it seems sensible only to allow ourselves to assign a value to this variable through this function We now need to think about *access privileges*

Access privileges

An instance of a class is known as an *object*. For example in the previous slide `wrooney` is an object of the class
Variables and functions associated with a class - for example `weeklyWage` and `CalculateSpendingMoney` - are known as class *members* and *methods*
There are three degrees of access to class members

- `private` - these class members are only accessible to other class members, unless `friend` is used
- `public` - these class members are accessible to everyone
- `protected` - these class members are accessible to other class members and to derived classes

To make the variable `weeklyWage` only accessible from outside the class through the function `SetWeeklyWage` we make `weeklyWage` a private class member
To highlight that it is now private we might also modify the name from `weeklyWage` to `mWeeklyWage`:

```
class PremiershipFootballer
{
private:
    int mWeeklyWage;
public:
    std::string firstName, surname, club;
    int weeklyExpenditure, appearances;
...
```

The reserved words `private` and `public` may be used as often as desired. For example the following is acceptable code

```
class PremiershipFootballer
{
public:
    std::string firstName, surname, club;
private:
    int mWeeklyWage;
public:
    int weeklyExpenditure, appearances;
...
```

The default for variables in a class is `private` For example, in the following code the variables `firstName`, `surname`, `club` are actually private

```
class PremiershipFootballer
{
    std::string firstName, surname, club;
private:
    int mWeeklyWage;
public:
    int weeklyExpenditure, appearances;
...
```

But now we can't access the variable `mWeeklyWage` outside the class. We need to write a public class member to access this variable

```
int PremiershipFootballer::GetWeeklyWage()
{
    return mWeeklyWage;
}
```

and this function may be used in the main code as follows

```
std::cout << "Weekly wage = "
    << wrooney.GetWeeklyWage() << "\n"
```

It is good software engineering practice to access as many variables as possible in this way

Constructors and Destructors

Each time an object of the class of Premiership footballers is created the program calls a function that allocates space in memory for all the variables used

This function is called a *constructor* and is automatically generated

This default constructor can be overridden if desired – for example in our class we may want to set the number of appearances to 0 when a new object is created

This function has the same name as the class, takes no arguments, has no return type and must be `public`

An overridden default constructor function is included in the class shown below

```
class PremiershipFootballer
{
private:
    int mWeeklyWage;
public:
    PremiershipFootballer();
...
```

and the function is written

```
PremiershipFootballer::PremiershipFootballer()
{
    appearances = 0;
}
```

Copy constructors

Another constructor that is automatically generated is a copy constructor
The line of code

```
PremiershipFootballer wrooney_copy(wrooney);
```

will create another object `wrooney_copy` with variables initialised to those of `wrooney`
This constructor may also be overridden in the same way as for the default constructor

Other constructors

Other constructors may be written.
For example you can write a constructor that initialises the first name and surname of a new object
This allows code to be written such as

```
PremiershipFootballer gsouthgate("Gareth", "Southgate");
```

The constructor must be added to the list of class members in the file `premiership_footballer.hpp`:

```
PremiershipFootballer(std::string first, std::string last)
```

The constructor is then written as

```
PremiershipFootballer::
    PremiershipFootballer(std::string first,
                          std::string last)
{
    firstName = first;
    surname = last;
}
```

This code is added to the file `premiership_footballer.cpp`
You can write as many constructors as you like

Destructors

When an object leaves scope it is destroyed
A *destructor* is automatically created that deletes the variables associated with that object
Destructors can also be overridden – there will be an example later

Use of pointers to classes

An instance of a class can be sent to a function in the same way as data types such as `double`, `int`, etc.
This is shown in the example code on the next slide

```
#include "PremiershipFootballer.hpp"
void SomeFunction(PremiershipFootballer player);
int main()
{
    PremiershipFootballer wrooney;
    SomeFunction(wrooney);
    return 0;
}

void SomeFunction(PremiershipFootballer player)
{
    some code
}
```

Note on the previous slide that as the function `SomeFunction` was called without the argument `wrooney` being a pointer, `SomeFunction` is unable to change any of the members of `wrooney` outside the body of the function

If you do want to change members of `wrooney` outside the body of the function, the function should be written to accept a pointer to wrooney as shown on the next slide

```cpp
#include "PremiershipFootballer.hpp"
void SomeFunction(PremiershipFootballer* pPlayer);
int main()
{
    PremiershipFootballer wrooney;
    SomeFunction(&wrooney);
    return 0;
}

void SomeFunction(PremiershipFootballer* pPlayer)
{
    (*pPlayer).appearances = 10;
}
```

The line of code on the previous slide

```
(*pPlayer).appearances = 10;
```

is a little clumsy. An equivalent statement is

```
pPlayer->appearances = 10;
```

Tip: step-through debuggers

- GCCs command-line debugger is `gdb`
- Most IDEs (Visual Studio, Eclipse, CLion etc.) have their own integrated step-through debuggers
- Other graphical front-ends to gdb exist (e.g. ddd).
- Note: Executable files suitable for debugging need to be compiled with `-g` (debug) rather than `-O` (optimized).