Numerical Computing in C++ - lectures 5-6

Fergus Cooper

2019

Lecture 5 — An introduction to classes

Consider the example of a linear solver that 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 movies! Each movie will have the following attributes:

- a title;
- a list of actors;
- a budget;
- a year it was released; and
- a number of Oscars it won.

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

This class may be written as:

```cpp
class Movie
{
public:
  int year, oscars;
  double budget;
  std::string title;
  std::vector<std::string> actors;
};
```

- 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

The class can then be used using like so:

```
Movie m;
m.title = "The Matrix";
m.actors.push_back("Keanu Reeves");
m.actors.push_back("Laurence Fishburne");
m.actors.push_back("Carrie-Anne Moss");
m.budget = 63.2;
m.year = 1999;
m.oscars = 4;

std::cout << m.title << std::endl;
```

[< compiler explorer >]

We want to be able to write functions that are associated with this class.

We will write a function `PrintSummary()` that prints out a nicely formatted summary of the information held in the class.

We declare the function inside the class. We define the function either inside the class or, as before, underneath the main function.

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

```cpp
class Movie
{
public:
  int year, oscars;
  double budget;
  std::string title;
  std::vector<std::string> actors;

  void PrintSummary();
};
```

```cpp
void Movie::PrintSummary()
{
  std::cout << title << " (" << year << ")\n";
  std::cout << "Starring:\n";
  for (const auto& actor : actors){
    std::cout << "  " << actor << "\n";
  }
  std::cout << "It cost $" << budget << "m, and won "
            << oscars << " oscars.";
}
```

[< compiler explorer >]

Loosely speaking, a list of variables and functions are kept in the class declaration, and the function definitions are kept elsewhere.

On the previous slide the compiler knows that the function definition is associated with the class `Movie` through the statement

```
void Movie::PrintSummary()
```

Where `Movie::` associates the definition with the declaration inside the class. This function may used outside the class by using statements such as:

```
m.PrintSummary();
```

Setting variables

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

```cpp
void Movie::SetYear(int y)
{
  if (y < 1888 || y > 2025) {
    std::cout << "Error setting movie release year!\n";
  }
  else {
    year = y;
  }
}
```

First we add the function declaration to the class as we did with the function `PrintSummary()`:

```
void SetYear(int y);
```

Then we add the function definition on the previous slide below our `main()` function.

We can now set the variable `year` using the statement:

```
m.SetYear(1999);
```

Now we have written a function to set the variable `year` that checks that it takes an appropriate value, it seems sensible to **only** allow ourselves to assign a value to this variable through this function. We do this with **access privileges**.

Access privileges

An instance of a class is known as an **object**. In the previous slide, for example, `m` is an object of type `Movie`.

Variables and functions associated with a class - for example `year` and `PrintSummary()` - are known as class **members** and **methods**.

There are three degrees of access to class members and methods:

- `private` - only accessible to other class members and methods
- `public` - accessible to everyone
- `protected` - accessible to other class members and to derived classes

To make the variable `year` only accessible via the function `SetYear` we make `year` a `private` class member.

To highlight that it is now private we might also modify the name from `year` to `mYear`:

```
class Movie
{
private:
  int mYear;
public:
  int oscars;
  double budget;
...
```

The default for variables in a class is `private`.

For example, the following code is equivalent to the code on the previous slide:

```
class Movie
{
  int mYear;
public:
  int oscars;
  double budget;
...
```

Now, though, there is no way to access the member `mYear` from outside the class. We need to write a public class method to access it:

```
int Movie::GetYear() {
    return mYear;
}
```

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

```
std::cout << "The Matrix is " << 2019 - m.GetYear() << " years old.\n";
```

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

`[< compiler explorer >]`

Lecture 6 — More on classes

Constructors and Destructors

Each time an object of the class of `Movie` 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 Oscars to `0` automatically when a new object is created.

A **constructor** has the same name as the class, has no return type, and must be `public`.

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

```
class Movie {
private:
  int mYear, mOscars;

public:
  Movie();
...
```

and the function is written

```
Movie::Movie() {
    mOscars = 0;
}
```

[< compiler explorer >]

Copy constructors

Another function that is automatically generated is a **copy constructor**.

The line of code

```
Movie m2 = m;
```

will create another object `m2` with variables initialised to those of `m`.

This **copy constructor** may also be overridden in the same way as for the default constructor.

Other constructors

You can write as many **constructors** as you like.

For example, we might want a constructor that initialises the title and the year of the movie.

This allows code to be written such as:

```
Movie m("The Matrix", 1999);
```

This **constructor** is declared as:

```
Movie(std::string title, int year);
```

This **constructor** is then defined as:

```
Movie::Movie(std::string t, int y)
{
  title = t;
  SetYear(y);
}
```

`[< compiler explorer >]`

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, but in modern C++ this is much less common than it used to be.

Other special member functions

There are six special member functions in total.

`[< wikipedia >]`

Other than the constructor, if you explicitly declare **any** of them, you must declare **all** of them.

`[< compiler explorer >]`

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, including as a pointer or as a reference.

This is shown in the example code on the next slide.

```cpp
void SomeFunctionValue(Movie m);
void SomeFunctionPointer(Movie* pm);
void SomeFunctionReference(Movie& rm);

int main() {
  Movie m("The Matrix", 1999);

  SomeFunctionValue(m);
  SomeFunctionPointer(&m);
  SomeFunctionReference(m);

  return 0;
}
```

```cpp
void SomeFunctionValue(Movie m);
void SomeFunctionPointer(Movie* pm);
void SomeFunctionReference(Movie& rm);
```

As before, if the object `m` is passed **by value** (`SomeFunctionValue`), the function receives a copy of the original object.

If `m` is passed **as a pointer** (`SomeFunctionPointer`) or **by reference** (`SomeFunctionReference`), the underlying object *can* be altered.

```cpp
void SomeFunctionPointer(Movie* pm) {
  (*pm).budget = 63.2;
}
void SomeFunctionReference(Movie& rm) {
  rm.budget = 63.2;
}
int main() {
  Movie m("The Matrix", 1999);

  SomeFunctionPointer(&m);
  SomeFunctionReference(m);

  return 0;
}
```

The line of code on the previous slide

```
(*pm).budget = 63.2;
```

is a little clumsy. An equivalent statement is

```
pm->budget = 63.2;
```

Tip: step-through debuggers

- Debuggers can be extremely useful tools.
- Demo of CLion debugger…