Tutorial: Constructing a Symmetric Matrix Class

Matteo Fasiolo

A Symmetric Matrix Class

We should start from `tut_symmatrix_template.cpp`.

We have previously seen how to construct a standard matrix class.

Here we will create a new class, called `symmmatrix` for storing symmetric matrices:

```
class symmmatrix: public matrix{

};
```

A symmetric matrix is a matrix -> inheritance makes sense here.

The line above makes so that `symmmatrix` inherits all the code we have written for the `matrix` class.

Problem 1

However, symmetric matrices are square, how do we enforce this constrain when constructing a new instance of the `symmmatrix`?

This needs to be enforced by the constructor.

Recall that constructors are **not** inherited by the child class.

To solve this problem very easily we can use **initialisation lists**.

Initialisation lists

Consider the code:

```
class Animal {
    double weight;
    int age;
    int lags;
    public:
    Animal(double w, int a, int l){
     weight = w;
     age = a;
     lags = l;
    };
};

int main() {
    Animal Oscar(3.3, 2, 4);
}
```

An alternative way of writing the constructor is:

```
Animal(double w, int a, int l) : weight(w), age(a) {
      lags = l;
};
```

where

- `: weight(w), age(a)` is the initialisation list
- `{ lags = l; }` is the function body

So we can initialise fields in the list or in the function body.

Here we can do as we like, but in some cases we need to use the list.

Initialising a Child Class

Consider

```
class Dog : public Animal {
 double height;
 public:
 Dog(double w, int a, int l, double h) : Animal(w, a, l){
  height = h;
 };
};
```

Here initialising the parents class can only be done using the list.

Now we are ready to solve Problem 1: initialising a `symmmatrix` while guaranteeing that the matrix is square.

Note: the initialiser of a `symmmatrix` should have only one argument!

Problem 2

The `void set_elem(int i, int j, int value)` method is available to the child `symmmatrix` class.

However, it does not impose symmetry, that is we can do:

```
a_sym_mat.set(1, 3, 1.5);
a_sym_mat.set(3, 1, -0.2);
```

which leads to an asymmetrix matrix.

Solution: add to the child `symmmatrix` class the alternative method

```
void set_elem(int i, int j, int value){
 \\ Here we should make sure that M(i, j) == M(j,i)
}
```

This is an example of **polymorphism**: the child class will have a method with the same interface as the parent class but with a different behaviour.

Testing code

The testing code should look something like this:

```cpp
int main()
{
    // construct B: a 3x3 symmetric matrix.
    symmmatrix B(3);

    for (int i = 0; i < 3; i++){
        for (int j = i; j < 3; j++){
            B.set_elem(i, j, rand() % 10);
        }
    }

    // print out the matrix: it must be symmetric
    B.print();
}
```

Approach 1: using “::”

The `set_elem` method of the `symmmatrix` can be implemented in several ways.

One approach implies doing something like this:

```
class Animal {
  public:
  void make_noise(){
   char noise[] = "This animal does: ";
   for (int i = 0; i < 19; i++) {
    printf("%c", noise[i]);
   }
 }
};

// continues ...
```

```
class Dog : public Animal {
  public:
  void make_noise(){
   Animal::make_noise(); // <- look at this
   char noise[] = "Woof Woof!";
   for (int i = 0; i < 10; i++) {
    printf("%c", noise[i]);
   }
 }
};

int main() {
    Dog Bellatrix;
    Bellatrix.make_noise();
} // Prints: "This animal does: Woof Woof!"
```

Here `Animal::` make so that you are calling the parent's method.

Task: define a `set_elem` method for the `symmmatrix` class which reuses the `set_elem` method of the `matrix` class.

Approach 2: Using “Protected” Fields

Alternatively we can re-write the `set_elem` method from scratch.

First note that this does not work:

```
class Animal {
  double weight;
  public:
   Animal(double w){
    weight = w;
  }
};

class Dog : public Animal {
  public:
  Dog(double w) : Animal(w) { }
  void set_weight(double w){
   weight = w;
  }
}; // continues ...
```

```
int main() {
    Dog Bellatrix(3.3);
    Bellatrix.set_weight(2.2);
}
// In member function ‘void Dog::set_weight(double)’:
// error: ‘double Animal::weight’ is private
// within this context
```

Problem: **child can not access parent’s private fields!**

Solution is to use the `protected` keyword:

```
class Animal {
  protected:
  double weight;
  public:
   Animal(double w){
    weight = w;
  }
}; // This works!
```

`protected` members of a class A are not accessible outside of A's code, but are accessible from the code of classes derived from A.

Try to use the `protected` keyword to define a `set_elem` method for the `symmatrix` without reusing any method from the `matrix` class.