C++ Programming - Advanced

Author: Peter Tse (mcreng)

Most of you may find this section less useful than the previous ones. Note that this section is target to those who wish to have a better understanding the smart car libraries (*libsccc*) since previous members have use the following skills to construct them. This section would be revolving around the concept of *class* we introduced in *Intermediate* tutorial and build upon it.

Class

This is an example of a class in C++. Throughout this tutorial, we will build upon this class.

```
class Motor{
public:
  Motor(){} // constructor
  void SetPower(uint16_t power){
    OnSetPower(power); // the actions to really change the power in real life
    m_power = power;
  } // setter of power
  uint16_t GetPower() { return m_power; } // getter of power
  void SetClockwise(bool flag) {
    OnSetClockwise(flag); // the actions to really change the direction in real life
    m_is_clockwise = flag;
  } // setter of direction
  bool IsClockwise() { return m_is_clockwise; } // getter of direction

private:
  bool m_is_clockwise;
  uint16_t m_power;
  void OnSetPower(const uint16_t power);
  void OnSetClockwise(const bool flag);
};
```

Class Construction

As you may remember from previous tutorial, you may either initialize the parameters through assignment or member initialization list. We prefer using member initialization list, so the code looks like this.

```cpp
class Motor{
public:
  Motor(bool is_clockwise, uint16_t power) :
    m_is_clockwise(is_clockwise),
    m_power(power){} // constructor
  void SetPower(uint16_t power){
    OnSetPower(power); // the actions to really change the power in real life
    m_power = power;
  } // setter of power
  uint16_t GetPower() { return m_power; } // getter of power
  void SetClockwise(bool flag) {
    OnSetClockwise(flag); // the actions to really change the direction in real life
    m_is_clockwise = flag;
  } // setter of direction
  bool IsClockwise() { return m_is_clockwise; } // getter of direction

private:
  bool m_is_clockwise;
  uint16_t m_power;
  void OnSetPower(const uint16_t power);
  void OnSetClockwise(const bool flag);
};
```

The parameters of the constructor may look very clumsy as it may get a lot of initialization values for certain modules, so we prefer using a configuration `struct` to store all the settings.

```cpp
class Motor{
public:

  struct Config{
    bool is_clockwise = true;
    uint16_t power = 0;
  };

  Motor(const Config& config):
    m_is_clockwise(config.is_clockwise),
    m_power(config.power){} // constructor

  void SetPower(uint16_t power){
    OnSetPower(power); // the actions to really change the power in real life
    m_power = power;
  } // setter of power
  uint16_t GetPower() { return m_power; } // getter of power
  void SetClockwise(bool flag) {
    OnSetClockwise(flag); // the actions to really change the direction in real life
    m_is_clockwise = flag;
  } // setter of direction
  bool IsClockwise() { return m_is_clockwise; } // getter of direction

private:
  bool m_is_clockwise;
  uint16_t m_power;
  void OnSetPower(const uint16_t power);
  void OnSetClockwise(const bool flag);
};
```

There will be different `Config` for different modules in libsccc, and for you to not confuse yourself when you do your code, we should limit you to explicitly construct the class (i.e. implicit type conversion is not allowed).

```cpp
class Motor{
public:

  struct Config{
    bool is_clockwise = true;
    uint16_t power = 0;
  };

  explicit Motor(const Config& config):
    m_is_clockwise(config.is_clockwise),
    m_power(config.power){} // constructor

  void SetPower(uint16_t power){
    OnSetPower(power); // the actions to really change the power in real life
    m_power = power;
  } // setter of power
  uint16_t GetPower() { return m_power; } // getter of power
  void SetClockwise(bool flag) {
    OnSetClockwise(flag); // the actions to really change the direction in real life
    m_is_clockwise = flag;
  } // setter of direction
  bool IsClockwise() { return m_is_clockwise; } // getter of direction

private:
  bool m_is_clockwise;
  uint16_t m_power;
  void OnSetPower(const uint16_t power);
  void OnSetClockwise(const bool flag);
};
```

This would be the results.

```cpp
Motor::Config config;
Motor bad_motor = config; // invalid due to explicit specifier
Motor motor(config); // valid
```

Inheritance

Let's say now we wish to implement a *proportional* controller (part of *PID*) to the motor, hopefully in mechanical tutorials it has been covered, but nevertheless here is the recap. Suppose we have an encoder to read the motor output and found that there is an error to the target output $error$, using the proportional controller we can tell the motor to correct its power output in next duty cycle with new power $k_p \cdot error$.

To do so, we can *inherit* the original class and add the controllers upon it.

```
class MotorController : Motor{
public:

  struct Config : Motor::Config{ // inherited the Config inside Motor
    float kp;
    float target;
  };

  MotorController(const Config& config) : // constructor
     m_kp(config.kp),
     m_target(config.target),
     Motor(config){} // to specify the Motor object constructor for inheritance

  void SetKp(float kp) { m_kp = kp; } // setter for Kp
  float GetKp() { return m_kp; } // getter for Kp

  void SetTarget(float target) { m_target = target; } // setter for target
  float GetTarget() { return m_target; } // getter for target

  void Control(float actual){ SetPower(Calc(actual)); } // controller

private:
  float m_target;
  float m_kp;
  float m_p;
  float Calc(float actual){ // P controller
    float error = m_target - actual;
    m_p = m_kp * error;
    return m_p;
  }
};
```

Here you may see two inheritance (specified with the `:` operator), one is for `struct` and one is for `class`. The meaning of inheritance is to copy the original `class` or `struct` and add things into it. For example, for `MotorController::Config`, it is different from `Motor::Config` by two members, `kp` and `target`. Both would also have the members `is_clockwise` and `power`. For `MotorController`, it would also keep the functions `SetPower()`, `GetPower()`, etc. Some terminologies, after inheritance, we may call `Motor` the *base class* and `MotorController` the *derived class*.

Inside the member initialization list, `Motor(config)` is specified for the compiler to call its corresponding constructor for `Motor`. If it is not specified, it would try to call `Motor()` as the constructor, and since there is no definition for it in class `Motor`, the compiler returns an error.

Public Inheritance vs Private Inheritance

As you may remember from Intermediate tutorial, the default visibility for the members in `struct` is `public` and that of `class` is `private`. Here we also have the same difference, in `config` it is defaulted to have `public` inheritance while for `class` it is private. From the table below you may understand the difference between the two.

Publicity in base class	Public	Private
Public Inheritance	Public	Private
Private Inheritance	Private	Private

That means, for `MotorController::Config`, all its members from its bass class would still be publicly accessible, and for `MotorController`, the members from its base class would all be private. This allows the `Config` to just extend itself and the `MotorController` to not use the original functions to mess up the setup. For the example above, only inside `MotorController` we can access the function `SetPower()`. If we wish to access it outside the class, we can force it to be a public inheritance.

```
class MotorController : public Motor{ // now public inheritance
  /* ... */
};
```

Overload

You may also wish to overload some functions in the base class.

```
class MotorContoller : public Motor{
  /* ... */
  void SetPower(uint16_t power); // overload of base class, objects of derived class will use this instead
  /* ... */
};
```

Polymorphism

In real life, we have two types of motor: alternate motors and direction motors, and they each work different in terms of how we change their directions and powers. So, they should have similar structures when representing both of them in C++ `class` syntax. Therefore, it is good to have some sort of *template* for both of the class, and let both of the class somehow inherit from the template. This is a concept of *polymorphism*, that we can build some classes with similar features with an *abstract class*.

Here we rewrite the class `Motor` to serve this exact purpose.

```cpp
class Motor{
public:

  struct Config{
    bool is_clockwise = true;
    uint16_t power = 0;
  };

  explicit Motor(const Config& config):
    m_is_clockwise(config.is_clockwise),
    m_power(config.power){} // constructor

  void SetPower(uint16_t power){
    OnSetPower(power); // the actions to really change the power in real life
    m_power = power;
  } // setter of power
  uint16_t GetPower() { return m_power; } // getter of power
  void SetClockwise(bool flag) {
    OnSetClockwise(flag); // the actions to really change the direction in real life
    m_is_clockwise = flag;
  } // setter of direction
  bool IsClockwise() { return m_is_clockwise; } // getter of direction

private:
  bool m_is_clockwise;
  uint16_t m_power;
  virtual void OnSetPower(const uint16_t power) = 0; // pure virtual function
  virtual void OnSetClockwise(const bool flag) = 0; // pure virtual function
};
```

When marking some certain functions in a class with `virtual` keyword and declare it being `0`, we have an abstract class in place. We cannot construct abstract classes.

```cpp
Motor::Config config;
Motor motor(config); // illegal
```

Then, we can inherit this abstract class and overload the functions `OnSetPower()` and `OnSetClockwise()` since they work differently in both types of motor.

```cpp
class AlternateMotor : public Motor{
public:
  AlternateMotor(const Motor::Config& config) : Motor(config){}
private:
  void OnSetPower(const uint16_t power) { /* ... */ }
  void OnSetClockwise(const bool flag) { /* ... */ }
};

class DirMotor : public Motor{
public:
  DirMotor(const Motor::Config& config) : Motor(config){}
private:
  void OnSetPower(const uint16_t power) { /* ... */ }
  void OnSetClockwise(const bool flag) { /* ... */ }
};
```

Both of them would have all the functions defined in `Motor`, with each has different implementations of `OnSetPower()` and `OnSetClockwise()`.

Abstract Class Pointer

For the `MotorController()` class, inheritance is no longer an option since `Motor` is now an abstract class. If you really wish to use inheritance for this, you can inherit once for each type of motor. Yet, it is actually better include a pointer to the motor objects and control the motor through this.

```cpp
class MotorController{
public:
  /* ... */
private:
  /* ... */
  Motor* p_motor = nullptr;
};
```

Here we can just use `Motor*` to represent all possible types of motor class that inherit `Motor`.

```cpp
Motor::Config config;
AlternateMotor m1(config);
DirMotor m2(config);
// note that m1 and m2 have different type
Motor* p_motor = nullptr;
p_motor = &m1; // valid, Motor* can point towards AlternateMotor
p_motor = &m2; // also valid, Motor* can point towards DirMotor
```

Generic Programming

Inside `MotorController`, we used `float` to store all k_P, error and target value. However, sometimes we may require `double`, and sometimes `int` would do. To make the class as easy to alter as it could, we may use *generic programming* for this. In short, generic programming is about making the variable types changeable.

```
template <typename T> // or template <class T>, here we define some type T, and we can replace all float with T
class MotorController{
public:
  /* ... */

  void SetKp(T kp) { m_kp = kp; } // setter for Kp
  T GetKp() { return m_kp; } // getter for Kp

  void SetTarget(T target) { m_target = target; } // setter for target
  T GetTarget() { return m_target; } // getter for target

  void Control(T actual){ SetPower(Calc(actual)); } // controller

private:
  T m_target;
  T m_kp;
  T m_p;
  T Calc(T actual){ // P controller
    T error = m_target - actual;
    m_p = m_kp * error;
    return m_p;
  }

  /* ... */
};
```

Codes that are related to template (generic programming) are sometimes stored in `.tcc` files under `\inc` (same place with `.h` files).

For the `MotorController` that we just updated, we can construct one of these objects with syntax

```
MotorController<float> motor_controller(...);
```

In fact, you can define more than two types.

```
template <typename T, typename U>
```

and the types that we feed in will be in order.

You may define default type for the template with

```
template <typename T = int>
```

and you may construct the class with

```
MotorController<> motor_controller(...); // <> must be present
```

to specify it is constructed under default type.

You may even apply the concept of generic programming into the definition of functions.

```
#include <iostream>

template <typename T>
T max(T a, T b) { return a > b ? a : b; }

int main() {
    std::cout << max(1, 2) << " " << max('a', 'S');
    return 0;
} // output: 2 a
```