Modern C++ Programming

24. Software Design II [DRAFT]

Design Patterns and Idioms

Federico Busato

2025-01-19

Table of Contents

C++ Idioms

Rule of Zero

The **Rule of Zero** is a rule of thumb for C++

> Utilize the *value semantics* of existing types to avoid having to implement *custom* copy and move operations

Note: many classes (such as `std` classes) manage resources themselves and should not implement copy/move constructor and assignment operator

```cpp
class X {
public:
    X(...); // constructor
            // NO need to define copy/move semantic
private:
    std::vector<int>      v; // instead raw allocation
    std::unique_ptr<int> p; // instead raw allocation
};                          // see smart pointer
```

Rule of Three

The **Rule of Three** is a rule of thumb for C++(03)

> If your class needs <u>any</u> of
>
> - a copy constructor `X(const X&)`
> - an assignment operator `X& operator=(const X&)`
> - or a destructor `~X()`
>
> defined explicitly, then it is likely to need <u>all</u> three of them

Some resources <u>cannot</u> or <u>should not</u> be copied. In this case, they should be declared as deleted

```
X(const X&) = delete
X& operator=(const X&) = delete
```

Rule of Five

The **Rule of Five** is a rule of thumb for C++11

If your class needs any of

- a copy constructor `X(const X&)`
- a move constructor `X(X&&)`
- an assignment operator `X& operator=(const X&)`
- an assignment operator `X& operator=(X&&)`
- or a destructor `~X()`

defined explicitly, then it is likely to need all five of them

Design Pattern

Singleton

Singleton is a software design pattern that restricts the instantiation of a class to one and only one object (a common application is for logging)

```
class Singleton {
public:
    static Singleton& get_instance() {  // note "static"
        static Singleton instance { ..init.. } ;
        return instance;   // destroyed at the end of the program
    }                      // initiliazed at first use

    Singleton(const& Singleton)      = delete;
    void operator=(const& Singleton) = delete;

    void f() {}
private:
    T _data;

    Singleton(..args..) { ... }  // used in the initialization
```

Pointer to IMPLementation (PIMPL) - Compilation Firewalls

Pointer to IMPLementation (PIMPL) idiom allows decoupling the interface from the implementation in a clear way

`header.hpp`

```cpp
class A {
public:
    A();
    ~A();
    void f();
private:
    class Impl;  // forward declaration
    Impl* ptr;   // opaque pointer
};
```

NOTE: The class does not expose internal data members or methods

PIMPL - Implementation

`source.cpp` (`Impl` actual implementation)

```
class A::Impl { // could be a class with a complex logic
public:
    void internal_f() {
        ..do something..
    }
private:
    int   _data1;
    float _data2;
};

A::A()     : ptr{new Impl()} {}
A::~A()     { delete ptr; }
void A::f() { ptr->internal_f(); }
```

PIMPL - Advantages, Disadvantages

Advantages:

- ABI stability
- Hide private data members and methods
- Reduce compile type and dependencies

Disadvantages:

- Manual resource management
  - `Impl* ptr` can be replaced by `unique_ptr<impl> ptr` in C++11
- Performance: pointer indirection + dynamic memory
  - dynamic memory could be avoided by using a reserved space in the interface e.g. `uint8_t data[1024]`

PIMPL - Implementation Alternatives

What parts of the class should go into the `Impl` object?

- *Put all private and protected members into* `Impl`:
  Error prone. Inheritance is hard for opaque objects

- *Put all private members (but not functions) into* `Impl`:
  Good. Do we need to expose all functions?

- *Put everything into* `Impl`*, and write the public class itself as only the public interface, each implemented as a simple forwarding function*:
  Good

https://herbsutter.com/gotw/_100/

The **Curiously Recurring Template Pattern (CRTP)** is an idiom in which a class `X` derives from a class template instantiation using `X` itself as template argument

A common application is *static polymorphism*

```cpp
template <class T>
struct Base {
    void my_method() {
        static_cast<T*>(this)->my_method_impl();
    }
};

class Derived : public Base<Derived> {
//  void my_method() is inherited
    void my_method_impl() { ... } // private method
};
```

Curiously Recurring Template Pattern


```cpp
#include <iostream>
template <typename T>
struct Writer {
    void write(const char* str) {
        static_cast<const T*>(this)->write_impl(str);
    }
};
class CerrWriter : public Writer<CerrWriter> {
    void write_impl(const char* str) {  std::cerr << str;  }
};
class CoutWriter : public Writer<CoutWriter> {
    void write_impl(const char* str) {  std::cout << str;  }
};
CoutWriter x;
CerrWriter y;
x.write("abc");
y.write("abc");
```

```
template <typename T>
void f(Writer<T>& writer) {
    writer.write("abc);
}

CoutWriter x;
CerrWriter y;
f(x);
f(y);
```

Virtual functions cannot have template arguments, but they can be emulated by using the following pattern

```
class Base {
public:
    template<typename T>
    void method(T t) {
        v_method(t);    // call the actual implementation
    }
protected:
    virtual void v_method(int t)    = 0; // v_method is valid only
    virtual void v_method(double t) = 0; // for "int" and "double"
};
```

Actual implementations for derived class `A` and `B`

```
class AImpl : public Base {
protected:
    template<typename T>
    void t_method(T t) { // template "method()" implementation for A
        std::cout << "A " << t << std::endl;
    }
};

class BImpl : public Base {
protected:
    template<typename T>
    void t_method(T t) { // template "method()" implementation for B
        std::cout << "B " << t << std::endl;
    }
};
```

Template Virtual Function


```cpp
template<class Impl>
class DerivedWrapper : public Impl {
private:
    void v_method(int t) override {
        Impl::t_method(t);
    }
    void v_method(double t) override {
        Impl::t_method(t);
    } // call the base method
};

using A = DerivedWrapper<AImpl>;
using B = DerivedWrapper<BImpl>;
```

```cpp
int main(int argc, char* argv[]) {
    A a;
    B b;
    Base* base = nullptr;

    base = &a;
    base->method(1);   // print "A 1"
    base->method(2.0); // print "A 2.0"

    base = &b;
    base->method(1);   // print "B 1"
    base->method(2.0); // print "B 2.0"
}
```

`method()` calls `v_method()` (pure virtual method of `Base`)

`v_method()` calls `t_method()` (actual implementation)