C++学习总结

1. 引用折叠、完美转发、move、forward

a. 引用折叠

i. T& & = T& T& && = T& T&& & = T& T&& && = T&&，编译器任何情况下都不允许手动写下int& &&的代码，必须指明是左值引用还是右值引用，但编译器可以自己推导，引用折叠多用在编译器自动类型推导的过程，在指明模板参数的时候也会用到引用折叠。

ii. 万能引用：既能接受左值类型的参数，也能接受右值类型的参数。本质是利用模板推导和引用折叠的规则，生成不同的实例化模板来接收传递进来的参数。

iii. 非模板函数是没有自动类型推导的，是左值就是左值，是右值引用就是右值引用，类型不匹配是会报错的！！！。

```
// 万能引用格式
// 模板函数参数为 T&& param,也就是说，不管T是什么类型，T&&的最终结果必然是一个引用类型。
// 如果T是int, 那么T&& 就是 int &&;
// 如果T为 int &，那么 T &&(int& &&) 就是&;
// 如果T为&&,那么T &&(&& &&) 就是&&。
// 很明显，接受左值的话，T只能推导为int &。
template<typename T>
ReturnType Function(T&& parem)
{
    // 函数功能实现
}
```

b. 完美转发 - forward

i. 为什么要用完美转发：任何函数内部，对形参的直接使用，都按照左值来进行的。

ii. 什么是完美转发：左值引用还是左值引用，右值引用还是右值引用，在进行参数传递的时候使用。

iii. forward需要指明模板参数，不指明编译报错！！！T forword T&& T& forward T& T&& forward T&&

c. 代码示例

```
#include <iostream>
#include <type_traits>

// remove reference
// 用模板和类结合的方式，使用模板特化左值引用和右值引用版本
// 在struct中定义基本类型，就可以返回所有基本类型
```

```cpp
// 核心思想在于模板的特化6
template <typename T>
struct my_remove_reference {
  typedef T type;
};

template <typename T>
struct my_remove_reference<T&> {
  typedef T type;
};

template <typename T>
struct my_remove_reference<T&&> {
  typedef T type;
};

void test01() {
  std::cout << "test01" << std::endl;
  std::cout << std::is_same<int, my_remove_reference<int>::type>::value
            << std::endl;
  std::cout << std::is_same<int, my_remove_reference<int&>::type>::value
            << std::endl;
  std::cout << std::is_same<int, my_remove_reference<int&>::type>::value
            << std::endl;
}

// move
// 自己实现move
// move的本质是模板推导、引用折叠、强制类型转换、类型检测
template <typename T>
constexpr typename my_remove_reference<T>::type&& my_move(T&& t) noexcept {
  return static_cast<typename my_remove_reference<T>::type&&>(t);
}

void test02() {
  std::cout << "test02" << std::endl;
  int a = 1;
  std::cout << std::is_same<int&&, decltype(my_move(a))>::value << std::endl;
  std::cout << std::is_same<int&&, decltype(my_move<int&>(a))>::value
            << std::endl;
```

```cpp

  int& b = a;
  std::cout << std::is_same<int&&, decltype(my_move(b))>::value << std::endl;
  std::cout << std::is_same<int&&, decltype(my_move<int&>(b))>::value
            << std::endl;

  std::cout << std::is_same<int&&, decltype(my_move(int(1)))>::value
            << std::endl;
  std::cout << std::is_same<int&&, decltype(my_move<int>(int(1)))>::value
            << std::endl;
  std::cout << std::is_same<int&&, decltype(my_move<int&&>(int(1)))>::value
            << std::endl;
}

// forward
// 自己实现forward
// 和move的区别是move的结果都是右值引用，
// 而forward的结果原本是左值就是左值，右值就是右值
// forward在使用的时候需要指明forward之前变量类型
// forward并没有用到模板的自动类型推导
template <typename T>
constexpr T&& my_forward(typename my_remove_reference<T>::type& t) noexcept {
  return static_cast<T&&>(t);
}

template <typename T>
constexpr T&& my_forward(typename my_remove_reference<T>::type&& t) noexcept {
  return static_cast<T&&>(t);
}

void test03() {
  std::cout << "test03" << std::endl;
  int a = 1;
  std::cout << std::is_same<int&&, decltype(my_forward<int>(a))>::value
            << std::endl;
  std::cout << std::is_same<int&, decltype(my_forward<int&>(a))>::value
            << std::endl;

  int& b = a;
```

```
  std::cout << std::is_same<int&&, decltype(my_forward<int>(b))>::value
            << std::endl;
  std::cout << std::is_same<int&, decltype(my_forward<int&>(b))>::value
            << std::endl;
  // 编译报错
  // std::cout << std::is_same<int&, decltype(my_forward(b))>::value <<
  // std::endl;

  std::cout << std::is_same<int&&, decltype(my_forward<int>(int(1)))>::value
            << std::endl;
  std::cout << std::is_same<int&, decltype(my_forward<int&>(int(1)))>::value
            << std::endl;
  std::cout << std::is_same<int&&, decltype(my_forward<int&&>(int(1)))>::value
            << std::endl;
}

// 之前所有测试指定模板类型是为了加深理解，正常情况下是不用指明类型的
// 让模板进行自动类型推导
int main(int argc, char** argv) {
  test01();
  test02();
  test03();
  return 1;
}
```

2. 原子操作-Atomic

a. 底层原理

i. 参考文献：https://zhuanlan.zhihu.com/p/48460953?utm_source=wechat_session&utm_medium=social&utm_oi=61423010971648

ii. 原子性体现在：对于变量的修改是原子性的，任何其他core都不会观察到对变量的中间状态。从底层来看，一个core对内存的操作分三步（RMW），当多个core执行相同代码的时候会产生冲突。

```
a += 1;

lw r1, a
addi r1, r1, 1
sw a, r1
```

iii. 原子操作可以保证原子性，但多个原子操作不能保证顺序性，在没有设置循序之前

（memory_order_relaxed）

iv. 两种实现方式：BusLock（CPU发出一个原子操作，锁住Bus，防止其他CPU内存操作）；CachelineLock（多核Cache（缓存，不同于寄存器）一通过MESI实现cache的一致性）

b. CPU微观原理

i. https://www.cnblogs.com/yanlong300/p/8986041.html

ii. CPU会有多级缓存体系，这将提升CPU的性能。但存在以下两个问题：

1. 上下级cache之间：CPU core将值写入L1 cache，但内存中的值没有得到更新，一般出现在单进程控制其他非CPU的agent，例如：CPU和GPU之间的交互。
2. 同级cache之间：一个4核处理器，每个core都有自己的L1缓存，那么一份数据可能在4个cache中都有相应拷贝。

iii. 在多核中，每个core都有自己的cache。围绕这个cache，有一个store buffer用以缓冲 本core 的store操作。此外，还有一个invalidate queue，用以排队 其他core 的invalidate command。

iv. 从代码层次看，单线程内相互依赖（有依赖关系的代码行）的数据执行是按照global顺序的，但没有依赖关系的顺序是不保证的。多线程就更别说了，更乱。

v. 内存序底层基础：写屏障是一条告诉处理器在执行这之后的指令之前，应用所有已经在存储缓存（store buffer）中的保存的指令（保证了release被其他core发现）；读屏障是一条告诉处理器在执行任何的加载前，先应用所有已经在失效队列中的失效操作的指令（保证之后读到的值都是其他核更新之后的值）。

c. 内存序

i. https://www.ccppcoding.com/archives/221

ii. Memory Order是用来用来约束同一个线程内的内存访问排序方式的，虽然同一个线程内的代码顺序重排不会影响本线程的执行结果，但是在多线程环境下，重排造成的数据访问顺序变化会影响其它线程的访问结果。（编译器优化，CPU优化，CPU Cache）

iii. 原子操作保证的是RMW的原子性（对一个atomic的修改是原子的，多线程下是互相可见的），但多原子操作之间的逻辑无法保证。

iv. 单线程指令不被重排指令：预编译指令asm volatile("":::"memory")

v. memory_order_seq_cst：顺序一致性模型，程序的执行顺序（单线程）和代码顺序严格一致。

vi. memory_order_relaxed：当前数据的访问是原子的，不会被其他线程的操作打断。例：多个++操作的结果是正确的（可以想象成RMW是原子的，至于底层怎么实现是底层的事）（CPU 编译器重排规则：while语句优先于后面的语句执行）。

vii. memory_order_release：T1对A写加memory_order_release，T1在A之前的任何读写操作都不能放在之后。T2对A读的时候加memory_order_acquire，T1在A之前任何读写操作都对T2可见；T2对A读的时候加memory_order_consume，T1在A之前所有依赖的读写操作对T2可见。

viii. memory_order_acquire：T1对A读加memory_order_acquire，在此指令之后的所有读写指令都不能重排到此指令之前。结合memory_order_release使用。

ix. memory_order_consume：T1对A读加memory_order_consume，在此指令之后的所有依赖此原子变量读写操作不能重排到该指令之前；T1在A之前所有依赖的对A操作对T2可见。结合

memory_order_release使用。(2016年6月起，所有编译起memory_order_consume和memory_order_acquire功能完全一致)

```cpp
#include <thread>
#include <atomic>
#include <cassert>
#include <string>

std::atomic<std::string*> ptr;
int data;

void producer() {
  std::string* p  = new std::string("Hello"); // L10
  data = 42; // L11
  ptr.store(p, std::memory_order_release); // L12
}

void consumer() {
  std::string* p2;
  while (!(p2 = ptr.load(std::memory_order_consume))); // L17
  assert(*p2 == "Hello"); // L18
  assert(data == 42); // L19
}

int main() {
  std::thread t1(producer);
  std::thread t2(consumer);
  t1.join();
  t2.join();

  return 0;
}
```

x. memory_order_acq_rel：当前线程T1中此操作之前或者之后的内存读写都不能被重新排序；对其它线程T2的影响是，如果T2线程使用了memory_order_release约束符的写操作，那么T2线程中写操作之前的所有操作均对T1线程可见；如果T2线程使用了memory_order_acquire约束符的读操作，则T1线程的写操作对T2线程可见。

```cpp
#include <thread>
#include <atomic>
#include <cassert>
#include <vector>

std::vector<int> data;
std::atomic<int> flag = {0};

void thread_1() {
    data.push_back(42); // L10
    flag.store(1, std::memory_order_release); // L11
}

void thread_2() {
    int expected=1; // L15
    // memory_order_relaxed is okay because this is an RMW,
    // and RMWs (with any ordering) following a release form a release sequence
    while (!flag.compare_exchange_strong(expected, 2, std::memory_order_acq_rel)) { // L18
        expected = 1;
    }
}

void thread_3() {
    while (flag.load(std::memory_order_acquire) < 2); // L24
    // if we read the value 2 from the atomic flag, we see 42 in the vector
    assert(data.at(0) == 42); // L26
}

int main() {
    std::thread a(thread_1);
    std::thread b(thread_2);
    std::thread c(thread_3);
    a.join();
    b.join();
    c.join();

    return 0;
```

```
38 }
```

3. STL-顺序容器

a. priority_queue

i. 原理：底层使用堆的思想，以vector为容器。

4. STL-关联容器

a. 二叉搜索树

i. 增：按树的结构去查找，找到相应的节点，插入叶子节点，叶子节点即为新值。

ii. 删：如果是叶子节点，直接删除；如果有一个子节点，则将父节点指向子节点；如果有两个节点，则将右子树的最小值替换当前值，递归删除最小值节点。

iii. 查：按左小右大的规则去查询。

b. AVL树

i. 任何节点的左右子树的高度差最多是1。

ii. 左左，左右，右左，右右分这四种插入情况，对于左左，右右这种方式，采用单旋；左右，右左采用双旋（第一次旋是变成左左、右右这种方式）；

c. RB-tree（红黑树）

i. 每个节点不是红就是黑。

ii. 根节点为黑。

iii. 如果节点为红，则子节点必须为黑。-》新增节点父节点必为黑。

iv. 任何节点至NULL（树尾端）的任何路径，黑节点个数必须相同。-》新增节点首先为红。

v. 每个节点有四个元素：颜色、父节点、左子节点、右子节点。

d. Set/Multiset

i. 底层采用红黑树的结构，key需要可以进行比较，或提供比较函数。

ii. 面对关联式容器，最好使用容器所提供的find函数查找，比STL算法find要快。

e. Map/Multimap

i. 底层采用红黑树的结构。

f. Hashtable

i. 冲突解决方案：线性探测（n+i）、二次探测(n+i**2)、拉链法。

ii. 空间大小一般为质数。

5. 关键字详解

a. constexpr

i. 常量表达式：由一个或者多个常量组成的表达式，常量表达式一旦确定该，值将无法修改。

ii. constexpr：使得指定的常量表达式获得在程序编译阶段就能计算出结果的能力，而不需要等到运行阶段（只是有能力，但能不能计算出来还是编译器说了算）。

iii. 修饰普通变量：constexpr int num = 1 + 2 + 3;

iv. 修饰函数返回值：

1. 整个函数的函数体中，除了可以包含 using 指令、typedef 语句以及 static_assert 断言外，只能包含一条 return 返回语句（不能包含赋值）。
2. 该函数必须有返回值，即函数的返回值类型不能是 void。
3. 函数在使用之前，必须有对应的定义语句。
4. return 返回的表达式必须是常量表达式。

v. 修饰类的构造函数：

1. 不能把自定义类型的struct class定义为constexptr，可以把构造函数定义为constexptr。
2. 函数体只能为空，采用初始化列表对成员变量进行赋值。

vi. 修饰模板函数：

1. 如果constexptr修饰的函数模板实例化结果不满足常量表达式的要求，constexptr会被忽略，等同步普通函数。

vii. C++版本区别

1. c++14可以声明变量，也可以使用goto try等流程控制语句。

```cpp
#include <iostream>
using namespace std;
// C++98/03
template<int N> struct Factorial
{
    const static int value = N * Factorial<N - 1>::value;
};
template<> struct Factorial<0>
{
    const static int value = 1;
};
// C++11
constexpr int factorial(int n)
{
    return n == 0 ? 1 : n * factorial(n - 1);
}
// C++14
constexpr int factorial2(int n)
{
    int result = 1;
    for (int i = 1; i <= n; ++i)
        result *= i;
    return result;
}
```

```cpp

int main()
{
    static_assert(Factorial<3>::value == 6, "error");
    static_assert(factorial(3) == 6, "error");
    static_assert(factorial2(3) == 6, "error");
    int n = 3;
    cout << factorial(n) << factorial2(n) << endl; //66
}
```

b. call_once

i. 指定函数只会被调用一次，需要结合once_flag（非局部变量）使用。

ii. 在call_once函数内部不要往外抛出异常，否则程序无法正常运行，这和网上说的不太一样，自己测试出来的。

iii. 使用案例：

```cpp
#include <iostream>
#include <thread>
#include <mutex>

std::once_flag flag;

inline void do_once(bool do_throw)
{
  std::call_once(flag, [](){ std::cout << "once" << std::endl; });
}

int main()
{
    std::thread t1(do_once, true);
    std::thread t2(do_once, true);

    t1.join();
    t2.join();
}
```

iv. 单例模式应用

```cpp
#include <atomic>
```

```cpp
#include <iostream>
#include <mutex>
#include <thread>

class SingletonAtomic {
 public:
  static SingletonAtomic* instance() {
    SingletonAtomic* ptr = inst_ptr_.load(std::memory_order_acquire);
    if (ptr == nullptr) {
      std::lock_guard<std::mutex> lock(mutex_);
      ptr = inst_ptr_.load(std::memory_order_relaxed);
      if (ptr == nullptr) {
        ptr = new (std::nothrow) SingletonAtomic();
        // 如果对inst_ptr_的操作不是原子操作，代码优化可能先赋值，后执行初始化
        // 如果在赋值之后切出线程，则内存为初始化，造成Bug
        inst_ptr_.store(ptr, std::memory_order_release);
      }
    }
    return ptr;
  }

 private:
  SingletonAtomic() {}
  SingletonAtomic(const SingletonAtomic&) {}
  SingletonAtomic& operator=(const SingletonAtomic&);

 private:
  static std::atomic<SingletonAtomic*> inst_ptr_;
  static std::mutex mutex_;
};
std::atomic<SingletonAtomic*> SingletonAtomic::inst_ptr_;
std::mutex SingletonAtomic::mutex_;

class SingletonCallOnce {
 public:
  static SingletonCallOnce* instance() {
    static std::atomic<SingletonCallOnce*> instance{nullptr};
    if (!instance.load(std::memory_order_acquire)) {
      static std::once_flag flag;
      std::call_once(flag, []() {
```

```
42          auto ptr = new (std::nothrow) SingletonCallOnce();
43          instance.store(ptr, std::memory_order_release);
44        });
45      }
46      return instance.load(std::memory_order_relaxed);
47    }
48
49   private:
50    SingletonCallOnce() {}
51    SingletonCallOnce(const SingletonCallOnce&) {}
52    SingletonCallOnce& operator=(const SingletonCallOnce&);
53  };
54
55  std::mutex cout_mutetx;
56
57  void Func() {
58    auto ptr1 = SingletonAtomic::instance();
59    {
60      std::lock_guard<std::mutex> lock(cout_mutetx);
61      std::cout << "ptr1: " << static_cast<void*>(ptr1) << std::endl;
62    }
63
64    auto ptr2 = SingletonCallOnce::instance();
65    {
66      std::lock_guard<std::mutex> lock(cout_mutetx);
67      std::cout << "ptr2: " << static_cast<void*>(ptr2) << std::endl;
68    }
69  }
70
71  int main() {
72    std::thread t1(Func);
73    std::thread t2(Func);
74    std::thread t3(Func);
75
76    t1.join();
77    t2.join();
78    t3.join();
79    return 0;
80  }
```

6. c++20特性（回家补充，公司电脑g++版本低）

a. 协程

i. 需要g++10版本的支持

b. Lambda

i. c++20之前隐式捕获this，c++20开始需要显式捕获this。

ii. 可以在lambda表达式中使用模板

```
[]template<T>(T x) {/* ... */};
[]template<T>(T* p) {/* ... */};
[]template<T, int N>(T (&a)[N]) {/* ... */};
```

设计模式

1. 外观模式

a. 为子系统中的一组接口定义一个一致的界面；外观模式提供一个高层的接口，这个接口使得这一子系统更加容易被使用。

b. 设计初期阶段，应有意识的将不同层分离，层与层之间建立外观模式。

c. 开发阶段，子系统越来越复杂，使用外观模式提供一个简单的调用接口。

d. 一个系统可能已经非常难易维护和扩展，但又包含了非常重要的功能，可以为其开发一个外观类，使得新系统可以方便的与其交互。

```
#include <iostream>

// 外观模式，为一组组件提供统一的外观定义（纯虚函数）
// 使得这组接口有着相同的表现，用的时候还是声明具体的对象

class Control {
 public:
  virtual void start() = 0;
  virtual void shutdown() = 0;
};

class Host: public Control {
 public:
  virtual void start() {
```

```
15      std::cout << "Host start" << std::endl;
16    }
17
18    virtual void shutdown() {
19      std::cout << "Host shutdown" << std::endl;
20    }
21  };
22
23  class Display: public Control {
24   public:
25    virtual void start() {
26      std::cout << "Display start" << std::endl;
27    }
28
29    virtual void shutdown() {
30      std::cout << "Display shutdown" << std::endl;
31    }
32  };
33
34  class Peripheral: public Control {
35   public:
36    virtual void start() {
37      std::cout << "Peripheral start" << std::endl;
38    }
39
40    virtual void shutdown() {
41      std::cout << "Peripheral shutdown" << std::endl;
42    }
43  };
44
45  class Computer {
46   public:
47    void start() {
48      host_.start();
49      display_.start();
50      peripheral_.start();
51    }
52
53    void shutdown() {
54      peripheral_.shutdown();
```

```
    display_.shutdown();
    host_.shutdown();
  }

 private:
  Host host_;
  Display display_;
  Peripheral peripheral_;
};

int main() {
  Computer computer;

  computer.start();

  computer.shutdown();

  return 0;
}
```

2. 模板模式

a. 定义一个操作中的算法的骨架，而将一些步骤延迟到子类中（虚函数）。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

```
#include <iostream>

class Computer {
 public:
  void product() {
    installCpu();
    installRam();
    installGpu();
  }

 private:
  virtual void installCpu() = 0;
  virtual void installRam() = 0;
  virtual void installGpu() = 0;
};
```

```
16
17 class ComputerA: public Computer {
18  private:
19   virtual void installCpu() override {
20     std::cout << "ComputerA us cpu1" << std::endl;
21   }
22   virtual void installRam() override {
23     std::cout << "ComputerA us ram1" << std::endl;
24   }
25   virtual void installGpu() override {
26     std::cout << "ComputerA us gpu1" << std::endl;
27   }
28 };
29
30 class ComputerB: public Computer {
31  private:
32   virtual void installCpu() override {
33     std::cout << "ComputerB us cpu2" << std::endl;
34   }
35   virtual void installRam() override {
36     std::cout << "ComputerB us ram2" << std::endl;
37   }
38   virtual void installGpu() override {
39     std::cout << "ComputerB us gpu2" << std::endl;
40   }
41 };
42
43 class ComputerC: public Computer {
44  private:
45   virtual void installCpu() override {
46     std::cout << "ComputerC us cpu3" << std::endl;
47   }
48   virtual void installRam() override {
49     std::cout << "ComputerC us ram3" << std::endl;
50   }
51   virtual void installGpu() override {
52     std::cout << "ComputerC us gpu3" << std::endl;
53   }
54 };
55
```

```
int main() {
  Computer* computer = new ComputerA();
  computer->product();
  delete computer;

  computer = new ComputerB();
  computer->product();
  delete computer;

  computer = new ComputerC();
  computer->product();
  delete computer;
  computer = nullptr;
}
```

3. 代理模式

a. 为其它对象提供一种代理以控制这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介作用。

```
#include <iostream>
#include <memory>
#include <string>

class Girl {
 public:
  Girl(const std::string& name = "Girl") : name_(name) {}
  std::string GetName() { return name_; }

 private:
  std::string name_;
};

// 需要将代理和真实对象之间有一个公共类
// 代理和真实类实现相同的方法，使得代理和真实类表现一样
class Profession {
 public:
  virtual ~Profession() {}
  virtual void profess() = 0;
};
```

```
21
22 class YongMan : public Profession {
23  public:
24   YongMan(const Girl& girl) : girl_(girl) {}
25   virtual void profess() {
26     std::cout << "Yong man name: " << girl_.GetName() << std::endl;
27   }
28
29  private:
30   Girl girl_;
31 };
32
33 class ManProxy : public Profession {
34  public:
35   ManProxy(const Girl& girl) : p_man_(new YongMan(girl)) {}
36   virtual void profess() { p_man_->profess(); }
37
38  private:
39   std::unique_ptr<YongMan> p_man_;
40 };
41
42 int main() {
43   Girl girl("Hanmeimei");
44   ManProxy man_proxy(girl);
45   man_proxy.profess();
46
47   return 0;
48 }
49
```

4. 观察者模式

a. 定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都要得到通知并自动更新。

```
1 #include <algorithm>
2 #include <iostream>
3 #include <list>
4 #include <memory>
5
```

```cpp
class View {
 public:
  virtual ~View() { std::cout << "~View()" << std::endl; }
  virtual void update() = 0;
  virtual std::string getName() = 0;
};

class DataModel {
 public:
  virtual ~DataModel() { std::cout << "~DataModel()" << std::endl; }
  virtual void addView(const std::shared_ptr<View>& view) = 0;
  virtual void removeView(const std::shared_ptr<View>& view) = 0;
  virtual void notify() = 0;
};

class TableView : public View {
 public:
  TableView(const std::string& name) : name_(name) {}
  void update() override { std::cout << name_ << " update" << std::endl; }
  std::string getName() override { return name_; }

 private:
  std::string name_;
};

class IntDataModel : public DataModel {
 public:
  // 增加or移除 view
  void addView(const std::shared_ptr<View>& view) override {
    auto iter = std::find(view_list_.begin(), view_list_.end(), view);
    if (iter == view_list_.end()) {
      view_list_.push_front(view);
    } else {
      std::cout << "View: " << view->getName() << " already exists"
                << std::endl;
    }
  }

  void removeView(const std::shared_ptr<View>& view) override {
    auto iter = std::find(view_list_.begin(), view_list_.end(), view);
```

```
    if (iter != view_list_.end()) {
      std::cout << "Remove view: " << (*iter)->getName() << std::endl;
      view_list_.erase(iter);
    }
  }

  void notify() override {
    for (auto& p_view : view_list_) {
      p_view->update();
    }
  }

 private:
  // 存储观察者
  std::list<std::shared_ptr<View>> view_list_;
};

int main() {
  auto v1 = std::make_shared<TableView>("TableView1");
  auto v2 = std::make_shared<TableView>("TableView2");
  auto v3 = std::make_shared<TableView>("TableView3");
  auto v4 = std::make_shared<TableView>("TableView4");

  IntDataModel int_data_model;
  int_data_model.addView(v1);
  int_data_model.addView(v2);
  int_data_model.addView(v3);
  int_data_model.addView(v4);

  int_data_model.notify();

  int_data_model.removeView(v1);
  int_data_model.removeView(v2);
  int_data_model.removeView(v3);
  int_data_model.removeView(v4);

  return 0;
}
```

5. 单例模式

a. 单例模式-懒汉模式

```cpp
#include <atomic>
#include <iostream>
#include <mutex>
#include <thread>

class SingletonAtomic {
 public:
  static SingletonAtomic* instance() {
    SingletonAtomic* ptr = inst_ptr_.load(std::memory_order_acquire);
    if (ptr == nullptr) {
      std::lock_guard<std::mutex> lock(mutex_);
      ptr = inst_ptr_.load(std::memory_order_relaxed);
      if (ptr == nullptr) {
        ptr = new (std::nothrow) SingletonAtomic();
        // 如果对inst_ptr_的操作不是原子操作，代码优化可能先赋值，后执行初始化
        // 如果在赋值之后切出线程，则内存为初始化，造成Bug
        inst_ptr_.store(ptr, std::memory_order_release);
      }
    }
    return ptr;
  }

 private:
  SingletonAtomic() {}
  SingletonAtomic(const SingletonAtomic&) {}
  SingletonAtomic& operator=(const SingletonAtomic&);

 private:
  static std::atomic<SingletonAtomic*> inst_ptr_;
  static std::mutex mutex_;
};
std::atomic<SingletonAtomic*> SingletonAtomic::inst_ptr_;
std::mutex SingletonAtomic::mutex_;

class SingletonCallOnce {
```

```
 public:
  static SingletonCallOnce* instance() {
    static std::atomic<SingletonCallOnce*> instance{nullptr};
    if (!instance.load(std::memory_order_acquire)) {
      static std::once_flag flag;
      std::call_once(flag, []() {
        auto ptr = new (std::nothrow) SingletonCallOnce();
        instance.store(ptr, std::memory_order_release);
      });
    }
    return instance.load(std::memory_order_relaxed);
  }

 private:
  SingletonCallOnce() {}
  SingletonCallOnce(const SingletonCallOnce&) {}
  SingletonCallOnce& operator=(const SingletonCallOnce&);
};

std::mutex cout_mutetx;

void Func() {
  auto ptr1 = SingletonAtomic::instance();
  {
    std::lock_guard<std::mutex> lock(cout_mutetx);
    std::cout << "ptr1: " << static_cast<void*>(ptr1) << std::endl;
  }

  auto ptr2 = SingletonCallOnce::instance();
  {
    std::lock_guard<std::mutex> lock(cout_mutetx);
    std::cout << "ptr2: " << static_cast<void*>(ptr2) << std::endl;
  }
}

int main() {
  std::thread t1(Func);
  std::thread t2(Func);
  std::thread t3(Func);

```

```
  t1.join();
  t2.join();
  t3.join();
  return 0;
}
```

6. 策略模式

a. 策略模式是指定义一系列的算法，把它们单独封装起来，并且使它们可以互相替换，使得算法可以独立于使用它的客户端而变化。本质是不同的策略为引起环境角色表现出不同的行为。

```
#include <functional>
#include <iostream>

void adcHurt() { std::cout << "Adc Hurt" << std::endl; }

void apcHurt() { std::cout << "Apc Hurt" << std::endl; }

class Soldier {
 public:
  typedef std::function<void()> Function;

  Soldier(Function fun) : func_(fun) {}
  void attack() { func_(); }

 private:
  Function func_;
};

class Mage {
 public:
  typedef std::function<void()> Function;

  Mage(Function fun) : func_(fun) {}
  void attack() { func_(); }

 private:
  Function func_;
};

```

```
int main() {
  Soldier soldier(adcHurt);
  Mage mage(apcHurt);

  soldier.attack();
  mage.attack();

  return 0;
}
```

7. 适配器模式

a. 适配器类需要继承或依赖已有的类，实现想要的目标接口。

```
#include <iostream>

// 被适配的类
class Deque {
 public:
  void pushFront() { std::cout << "push front" << std::endl; }
  void pushBack() { std::cout << "push back" << std::endl; }
  void popFront() { std::cout << "pop front" << std::endl; }
  void popBack() { std::cout << "pop back" << std::endl; }
};

// 成员函数的方式实现适配器模式
class Stack1 {
 public:
  void push() { deque_.pushFront(); }
  void pop() { deque_.popFront(); }

 private:
  Deque deque_;
};

class Queue1 {
 public:
  void push() { deque_.pushFront(); }
  void pop() { deque_.popBack(); }

```

```
 private:
  Deque deque_;
};

// 继承的方式实现适配器模式
class Stack2 : private Deque {
 public:
  void push() { pushFront(); }
  void pop() { popFront(); }
};

class Queue2 : private Deque {
 public:
  void push() { pushFront(); }
  void pop() { popBack(); }
};

int main() {
  Stack1 stack1;
  stack1.push();
  stack1.pop();
  Queue1 queue1;
  queue1.push();
  queue1.pop();

  Stack2 stack2;
  stack2.push();
  stack2.pop();
  Queue2 queue2;
  queue2.push();
  queue2.pop();

  return 0;
}
```

编码经验

1. Effective C++

a. 类默认生成6个函数。

b. 不自动生成的函数就拒绝，要么public + delete，要么private的方式。

c. 基类的析构函数一定要声明为虚函数。

d. 不要让异常逃离析构函数：在析构函数中处理异常；在单独的函数中调用异常函数，异常交由调用者处理。

e.

参考文献

c. https://github.com/Light-City/CPlusPlusThings

d. https://refactoringguru.cn/design-patterns/cpp

e. 常用设计模式：https://www.cnblogs.com/schips/p/12306851.html

f.