C++学习总结

1. 模板相关

a. 可变模板参数

i. 模板参数包，函数参数包：

```
template<typename T,typename... Args> // typename... 模板参数包类型 Args 模板参数包，可以是0或者多种类型的组合
void printMsg(T t, Args... args) {} // Args...参数包展开，args函数参数包
sizeof...(args) // 获取参数个数
```

ii. 模板内用法

```
#include <iostream>

template <typename T>
void myPrint(T&& t) {
  std::cout << t << std::endl;
}

template <typename T, typename... Args>
void myPrint(T&& t, Args&&... args) {
  std::cout << t << ',';
  myPrint(std::forward<Args>(args)...);
}

int main() {
  myPrint(1, 3, "afasfdf", true);
}

```

iii. 高级用法：

```
// 测试耗时通用框架
#include <chrono>
#include <iostream>
```

```cpp
#include <type_traits>
#include <vector>

// --std=c++17
template <typename F, typename... Args>
std::invoke_result_t<F, Args...> runTimeNoReturn(F&& f, Args&&... args) {
  using ResType = std::invoke_result_t<F, Args...>;
  auto start = std::chrono::high_resolution_clock::now();
  f(std::forward<Args>(args)...);
  auto end = std::chrono::high_resolution_clock::now();
  std::cout << "cast: "
            << std::chrono::duration_cast<std::chrono::milliseconds>(end -
                                                                     start)
                   .count()
            << std::endl;
  return;
}

template <typename F, typename... Args>
std::invoke_result_t<F, Args...> runTimeWithReturn(F&& f, Args&&... args) {
  using ResType = std::invoke_result_t<F, Args...>;
  auto start = std::chrono::high_resolution_clock::now();
  ResType result = f(std::forward<Args>(args)...);
  auto end = std::chrono::high_resolution_clock::now();
  std::cout << "cast: "
            << std::chrono::duration_cast<std::chrono::milliseconds>(end -
                                                                     start)
                   .count()
            << std::endl;
  return result;
}

template <typename F, typename... Args>
std::invoke_result_t<F, Args...> runTime(F&& f, Args&&... args) {
  using ResType = std::invoke_result_t<F, Args...>;
  if constexpr (std::is_void_v<ResType>) {
    return runTimeNoReturn(std::forward<F>(f), std::forward<Args>(args)...);
  } else {
    return runTimeWithReturn(std::forward<F>(f), std::forward<Args>(args)...);
```

```
  }
}

int main() {
  std::vector<int> a;
  runTime([&]() -> void {
    for (int i = 0; i < 100000; ++i) {
      a.push_back(i);
    }
  });

  std::vector<int> b;
  int c = runTime([&]() -> int {
    for (int i = 0; i < 100; ++i) {
      b.push_back(i);
      c += i;
    };
    return c;
  });
  std::cout << c << std::endl;
  return 0;
}
```

2. 继承

a. 类型转换

i. 类型转换主要应用于需要转换对象类型的情况，比c语言原生括号的形式要更加安全，明了，且更符合c++规范。且类型转换可用于向下类型转换，是个动态的。而父类指针指向子类是静态的，编译的时候就已经确定了。

ii. static_cast：强制类型转换，和父类指针指向子类对象只是在用法上有差别，本质上目前还没发现差别。

```
// static_cast
// 基本类型之间转换
// 引用类型转换，上下都可以，但是需要程序员确定是安全转换
// 空指针转换，任何类型指针都可以转换成void*
```

iii. dynamic_cast：必须有虚函数，是对引用或者指针的转换，不能直接转换变量。转换引用失败抛出异

常，转换指针失败则为空指针。

b. 向上转型

i. 只有在构造的时候，才会把派生类升级为基类（调用基类的所有函数，包括基类的虚函数实现），其他情况父类指针引用指向派生类，普通函数调用的是父类的，虚函数调用的是派生类的。

ii. 派生类对象可以给基类赋值，基类不可以给派生类赋值。赋值的本质是数据的填充，如果不定义赋值（初始化列表或者赋值函数），默认就用多的成员变量的子类给基类的各个变量赋值。

```
#include <iostream>

class A {
 public:
  A(int a) : a_(a) { std::cout << "Init A" << std::endl; }
  A(const A& rhf) : a_(rhf.a_) {
    // 这里表现为调用子类的display
    std::cout << "rhf display" << std::endl;
    rhf.display();
    std::cout << "Copy Init A" << std::endl;
  }
  A& operator=(const A& rhf) {
    // 这里表现为调用子类的display
    std::cout << "rhf display" << std::endl;
    rhf.display();
    a_ = rhf.a_;
    std::cout << "Copy" << std::endl;
    return *this;
  }

  virtual void display() const { std::cout << "a:" << a_ << std::endl; }
  int a_;
};

class B : public A {
 public:
  B(int a, int b) : A(a), b_(b) { std::cout << "Init B" << std::endl; }

  void display() const override {
    std::cout << "a:" << a_ << " b:" << b_ << std::endl;
  }
  int b_;
```

```
};

int main() {
  B b(1, 2);
  A a = b;

  return 0;
}
```

iii. 派生类指针赋值给基类指针（引用），在调用虚函数的时候会有差别。

iv. 同样，在派生类指针赋值给基类指针的时候，编译器会在赋值之前进行处理，对象指针必须是对象的起始地址。

v. 类在调用成员函数（包括虚函数和非虚函数）的时候，默认都传this指针，当基类指针调用子类的虚函数的时候，会把this指针转换成子类传入。

vi. 函数的声明中成员变量和成员函数分开看，成员变量是成员变量，占用空间，普通函数（非虚函数）不占用空间，所有类的示例共享同一个类的函数实现，通过this指针（类成员函数默认传this指针）实现这种联系。

```
#include <iostream>

class A {
 public:
  A(int a) : a_(a) { std::cout << "Init A" << std::endl; }

  void self_display() const { std::cout << "a:" << a_ << std::endl; }
  virtual void display() const { std::cout << "a:" << a_ << std::endl; }
  int a_;
};

class B : public A {
 public:
  B(int a, int b) : A(a), b_(b) { std::cout << "Init B" << std::endl; }

  void self_display() const {
    std::cout << "a:" << a_ << " b:" << b_ << std::endl;
  }
  virtual void display() const override {
    std::cout << "a:" << a_ << " b:" << b_ << std::endl;
  }
```

```
  int b_;
};

class C {
 public:
  C(int c) : c_(c) { std::cout << "Init C" << std::endl; }

  void self_display() const { std::cout << "c:" << c_ << std::endl; }
  virtual void display() const { std::cout << "c:" << c_ << std::endl; }
  int c_;
};

class D : public B, public C {
 public:
  D(int a, int b, int c, int d) : B(a, b), C(c), d_(d) {
    std::cout << "Init D" << std::endl;
  }

  void self_display() const {
    std::cout << "a:" << a_ << " b:" << b_ << " c:" << c_ << " d:" << d_
              << std::endl;
  }
  virtual void display() const override {
    std::cout << static_cast<const void*>(this) << std::endl;
    std::cout << "a:" << a_ << " b:" << b_ << " c:" << c_ << " d:" << d_
              << std::endl;
  }
  int d_;
};

int main() {
  D* d = new D(1, 2, 3, 4);

  A* a = d;
  B* b = d;
  C* c = d;

  a->display();
  b->display();
  c->display();
```

```
  d->display();

  a->self_display();
  b->self_display();
  c->self_display();
  d->self_display();

  std::cout << static_cast<void*>(a) << std::endl;
  std::cout << static_cast<void*>(b) << std::endl;
  std::cout << static_cast<void*>(c) << std::endl;
  std::cout << static_cast<void*>(d) << std::endl;

  delete d;
  return 0;
}
```

c. 多继承

i. 多继承是一个类继承多个类，多继承会产生菱形继承的问题。

```
#include <iostream>

// 不包含虚继承的菱形继承
class A {
 public:
  virtual ~A() {}
  int32_t a_;
};

class B : public A {
 public:
  virtual ~B() {}
  int32_t b_;
};

class C : public A {
 public:
  virtual ~C() {}
  int32_t c_;
```

```
};

class D : public B, public C {
 public:
  // void seta(int32_t a) { a_ = a; } // 编译出错，不知道a_是哪个类里的。
  void setBa(int32_t a) { B::a_ = a; }
  void setCa(int32_t a) { C::a_ = a; }

  void setb(int32_t b) { b_ = b; }
  void setc(int32_t c) { c_ = c; }
  void setd(int32_t d) { d_ = d; }
  int32_t d_;
};

void test01() {
  D d;
  d.setBa(1);
  d.setCa(2);
  d.setb(3);
  d.setc(4);
  d.setd(5);

  B& b = static_cast<B&>(d);
  std::cout << b.a_ << std::endl;

  D& dd = static_cast<D&>(b);
  std::cout << dd.B::a_ << std::endl;
  std::cout << dd.C::a_ << std::endl;

  // 在父类没有定义虚函数的前提下，编译不支持从上往下动态装换
  D& ddd = dynamic_cast<D&>(b);
  // C& c = static_cast<C&>(b); // 在没有父子关系的类中，不支持static转换
  C& c = dynamic_cast<C&>(b);
  std::cout << c.a_ << std::endl;
}

int main() {
  test01();
  return 0;
}
```

d. 多继承 && 成员变量构造顺序

```
#include <iostream>

// 多继承父类的构造顺序，和成员变量的构造顺序是根据类的继承顺序，类内成员函数的定义的顺序决定的
// 而不是初始化列表的顺序
// 对于虚继承永远都是最终的派生类直接调用虚虚基类的构造函数，还是首先调用
class X {
 public:
  X(int32_t x) : x_(x) { std::cout << "Init X" << std::endl; }
  int32_t x_;
};

class A : virtual public X {
 public:
  A(int32_t a) : X(11), a_(a) { std::cout << "Init A" << std::endl; }
  int32_t a_;
};

class B : virtual public X {
 public:
  B(int32_t b) : X(22), b_(b) { std::cout << "Init B" << std::endl; }
  int32_t b_;
};

class C {
 public:
  C(int32_t c) : c_(c) { std::cout << "Init C" << std::endl; }
  int32_t c_;
};

class D {
 public:
  D(int32_t d) : d_(d) { std::cout << "Init D" << std::endl; }
  int32_t d_;
};

```

```
class E : public A, public B {
 public:
  E() : B(1), A(2), X(33), e_(3), d_(4), c_(5) {
    std::cout << "Init E" << std::endl;
  }
  C c_;
  D d_;
  int32_t e_;
};

int main() {
  E e;
  return 0;
}
```

e. 虚继承

i. 虚继承是在中间类上实现虚继承，具体的子类不用虚继承多个父类。就好像让某个类做出声明，承诺愿意共享它的基类。

ii. 虚派生只影响从指定了虚基类的派生类中进一步派生出来的类，它不会影响派生类本身。

iii. 当子类覆盖基类的成员变量时，仍存在二义性的情况。二义性是指在不指定成员变量所属域的前提下，存在同名变量，存在于多继承中，多个同级父类用相同名称的成员变量的情况，子类覆盖父类成员变量不算。（详见代码）

iv. 最终派生类的构造函数必须要调用虚基类的构造函数。对最终的派生类来说，虚基类是间接基类，而不是直接基类。（详见代码）

```
#include <iostream>

class A {
 public:
  A(int32_t init_a) : a(init_a), x(0), y(0) {
    std::cout << "Init A" << std::endl;
  }

  int32_t a;
  int32_t x;
  int32_t y;
};

```

```
class B : virtual public A {
 public:
  // 在使用B进行对象构造的时候，调用A的构造函数，a被赋值为111
  B(int32_t init_b) : A(111), b(init_b), x(0), y(0) {
    std::cout << "Init B" << std::endl;
  }
  int32_t b;
  int32_t x;
  int32_t y;
};

class C : virtual public A {
 public:
  // 在使用B进行对象构造的时候，调用A的构造函数，a被赋值为222
  C(int32_t init_c) : A(222), c(init_c), x(0) {
    std::cout << "Init C" << std::endl;
  }
  int32_t c;
  int32_t x;
};

class D : public B, public C {
 public:
  // 在使用D进行对象构造的时候，调用A的构造函数，a被赋值为333,与B、C构造冲突，查看结果
  D(int32_t init_d) : A(333), B(444), C(555), d(init_d) {
    std::cout << "Init D" << std::endl;
  }
  int32_t d;

  friend std::ostream& operator<<(std::ostream& out, const D& d) {
    out << d.a << ' ' << d.b << ' ' << d.c << ' ' << d.d << ' ' << d.B::x << ' '
        << d.C::x << ' ' << d.y;
    return out;
  }
};

void test01() {
  // 虚继承如果在不存在覆盖的情况下，不存在二义性
  // 在构造的时候，最终派生类直接调用虚基类的构造，跳过中间派生类的构造
  D d(4);
```

```
  d.a = 1;
  d.b = 2;
  d.c = 3;

  // 虚继承在被一个子类覆盖，也不存在二义性
  d.y = 5;
  std::cout << d.A::y << ' ' << d.B::y << std::endl;

  // 虚继承被多个子类覆盖，存在二义性
  d.B::x = 6;
  d.C::x = 7;
  std::cout << d << std::endl;

  // 测试虚基类构造调用顺序
  B b(11);
  std::cout << b.a << std::endl;
  D e(999);
  std::cout << e << std::endl;
}

int main() {
  test01();
  return 0;
}

```

3. 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 &。
#include <iostream>
#include <type_traits>

template <typename T>
void function(T&& parem) {
  if (std::is_lvalue_reference<T>::value) {
    std::cout << "T is left value reference" << std::endl;
  } else if (std::is_rvalue_reference<T>::value) {
    std::cout << "T is right value reference" << std::endl;
  } else {
    std::cout << "T is not reference" << std::endl;
  }

  if (std::is_lvalue_reference<T&&>::value) {
    std::cout << "T&& is left value reference" << std::endl;
  } else if (std::is_rvalue_reference<T&&>::value) {
    std::cout << "T&& is right value reference" << std::endl;
  } else {
    std::cout << "T&& is not reference" << std::endl;
  }
}

int main() {
  function(1);
  int a = 1;
  function(a);
  function(std::move(a));
  return 0;
}

```

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

```cpp
#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
// 核心思想在于模板的特化
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>
```

```cpp
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;

  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使用在万能引用内部，需要指明要转发的类型，因为是万能转发，
// 所以在指明变量类型的时候不能指明为非引用类型，即使指明为引用类型，也会被折叠为右值引用
// 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);
}
```

```
77
78  void test03() {
79    std::cout << "test03" << std::endl;
80    int a = 1;
81    // 一版不这么用，指明变量类型为左值引用或者右值引用
82    // std::cout << std::is_same<int&&, decltype(my_forward<int>(a))>::value <<
83    // std::endl;
84    std::cout << std::is_same<int&, decltype(my_forward<int&>(a))>::value
85              << std::endl;
86
87    int& b = a;
88    // 一版不这么用，指明变量类型为左值引用或者右值引用
89    // std::cout << std::is_same<int&&, decltype(my_forward<int>(b))>::value <<
90    // std::endl;
91    std::cout << std::is_same<int&, decltype(my_forward<int&>(b))>::value
92              << std::endl;
93    // 编译报错
94    // std::cout << std::is_same<int&, decltype(my_forward(b))>::value <<
95    // std::endl;
96
97    std::cout << std::is_same<int&&, decltype(my_forward<int>(int(1)))>::value
98              << std::endl;
99    std::cout << std::is_same<int&, decltype(my_forward<int&>(int(1)))>::value
100             << std::endl;
101   std::cout << std::is_same<int&&, decltype(my_forward<int&&>(int(1)))>::value
102             << std::endl;
103 }
104
105 // 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
106 // 让模板进行自动类型推导
107 int main(int argc, char** argv) {
108   test01();
109   test02();
110   test03();
111   return 1;
112 }
```

4. 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. 单核CPU上编译器也会对代码前后顺序进行优化，但不会影响代码逻辑。

iii. 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，不存在RMW问题。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。

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

uint64_t a = 0;
std::atomic_uint64_t count(0);

void func() {
  for (int i = 0; i < 1000000; i++) {
    a += 1;
    count.fetch_add(1, std::memory_order_relaxed);
  }
}

void test01() {
  std::thread t1(func);
  std::thread t2(func);
  t1.join();
  t2.join();
  std::cout << a << ' ' << count.load(std::memory_order_relaxed) << std::endl;
}

int main() {
  test01();
  return 0;
}

```

```
a += 1;
```

```
2
3 lw r1, a
4 addi r1, r1, 1
5 sw a, r1
```

iv. 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（因为在单核上跑的顺序 不一定是代码顺序），在没有设置循序之前（memory_order_relaxed）

v. 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）（l1缓存 离core更近）

vi. 原子操作要在函数内部要保证根据需求设置内存序，函数外就是编译器对多个函数代码前后执行顺序的优化。（函数顺序优化：两个无关独立函数；没有共享的成员函数；没有标记依赖关系的内联函数。）最好的办法是把相关的原子操作放在一个函数中。

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. MESI：每个cache line有四个状态，用2bit表示。

v.

vi. 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。

vii. 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，发现是通过MESI协议发现的，而不是通过内存）；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）。

viii. 屏障的意义在于规范不同变量之间的最新值被发现，和原子操作还不是一回事。

ix. 原子操作在于多线程对统一变量的修改是原子的。原子操作和屏障结合可以保证多原子变量按顺序执行。

x. 普通局部变量用不到是因为单线程内CPU和编译器进行的优化是不会影响逻辑的，因此没必要考虑内存序！！！

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)

x. 屏障之间的代码越少越好，屏障之间的代码可以理解为冲突区，冲突越小越好。

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

// 测试原子操作和非原子操作对结果的影响
// 原子操作保证了线程安全，非原子操作则可能导致数据不一致
uint64_t a = 0;
std::atomic_uint64_t count(0);

void func() {
  for (int i = 0; i < 1000000; i++) {
    a += 1;
    count.fetch_add(1, std::memory_order_relaxed);
  }
}

void test01() {
```

```
  std::thread t1(func);
  std::thread t2(func);
  t1.join();
  t2.join();
  std::cout << a << ' ' << count.load(std::memory_order_relaxed) << std::endl;
}

// 测试memory_order_acquire和memory_order_release配合使用
std::atomic<std::string*> ptr(nullptr);
std::atomic_uint64_t count2(0);
int data = 0;

void producer() {
  for (int i = 0; i < 1000000; ++i) {
    std::string* p = new std::string("producer");
    count2.fetch_add(1, std::memory_order_relaxed);
    ++data;
    ptr.store(p, std::memory_order_release);

    while (ptr.load(std::memory_order_acquire)) {}
  }
}

void consumer() {
  for (int i = 0; i < 1000000; ++i) {
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire))) {}
    count2.fetch_add(1, std::memory_order_relaxed);
    ++data;

    delete p2;
    ptr.store(nullptr, std::memory_order_release);
  }
}

void test02() {
  std::thread t1(producer);
  std::thread t2(consumer);
  t1.join();
  t2.join();
```

```
  std::cout << count2.load(std::memory_order_relaxed) << ' ' << data << ' '
            << static_cast<void*>(ptr.load(std::memory_order_relaxed))
            << std::endl;
}

int main() {
  test01();
  test02();
  return 0;
}

```

xi. 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。

xii. compare_exchange_weak,compare_exchange_storg;strong可以保证正确返回，weak不能保证正常返回，但效率高，可以使用while重试。

```
bool compare_exchange_weak (T& expected, T desired, std::memory_order success,
std::memory_order failure) noexcept;
// expected期待现在的值是多少
// 如果匹配则替换成desired，如果不匹配则把expected替换成现在值。
// 如果成功则使用第一个内存序，如果失败则使用第二个内存序。
// 失败的内存序不能超过成功
// success: std::memory_order_acq_rel failure: std::memory_order_acquire
// success: std::memory_order_acq_release failure: std::memory_order_relaxed
```

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

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

```

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

void thread_2() {
  int expected = 1;
  while (!flag.compare_exchange_weak(expected, 2, std::memory_order_release,
                                     std::memory_order_relaxed)) {
    expected = 1;
  }
}

void thread_3() {
  while (flag.load(std::memory_order_acquire) < 2) {
    // 让cpu等待一段小小的时间
    asm volatile("rep; nop" ::: "memory");
  }
  assert(data.at(0) == 42);
}

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;
}

```

d. 自旋锁

i. 任何时刻只有一个线程获得锁，如果在获取锁的时候，锁被其他线程占有，则循环等待并探测，直到获取锁才会退出循环。

ii. 优化方法：上锁之后延迟一段时间；尝试一定次数后线程挂起；

iii. X86设计了Pause指令，也就是调用Pause 指令的代码会抢着CPU 不释放，但是CPU 会打个盹，比如10个时钟周期，相对一次上下文切换是大几千个时钟周期。（sleep yield会释放CPU，sleep让出

的CPU时间是固定的， yield让出的时间片是不固定的，以CPU调度时间片为单位，且不同操作系统不同CPU调度策略表现不)

```
class spin_mutex {
  std::atomic_flag flag = ATOMIC_FLAG_INIT;
public:
  spin_mutex() = default;
  spin_mutex(const spin_mutex&) = delete;
  spin_mutex& operator= (const spin_mutex&) = delete;
  void lock() {
    while(flag.test_and_set(std::memory_order_acquire))
      ;
  }
  void unlock() {
    flag.clear(std::memory_order_release);
  }
};

// 延迟一段时间执行，不同CPU架构不同方法，公司x86_64
inline void cpu_relax() {
#if defined(__aarch64__)
  asm volatile("yield" ::: "memory");
#else
  asm volatile("rep; nop" ::: "memory"); // 这个会被翻译成pause
#endif
}
/**
  Run a delay loop while waiting for a shared resource to be released.
  @param delay originally, roughly microseconds on 100 MHz Intel Pentium
*/
static inline void ut_delay(unsigned delay)
{
  unsigned i= my_cpu_relax_multiplier / 4 * delay;
  HMT_low();
  while (i--)
    MY_RELAX_CPU();
  HMT_medium();
}

```

```
static inline void MY_RELAX_CPU(void)
{
#ifdef _WIN32
  /*
    In the Win32 API, the x86 PAUSE instruction is executed by calling
    the YieldProcessor macro defined in WinNT.h. It is a CPU architecture-
    independent way by using YieldProcessor.
  */
  YieldProcessor();
#elif defined HAVE_PAUSE_INSTRUCTION
  /*
    According to the gcc info page, asm volatile means that the
    instruction has important side-effects and must not be removed.
    Also asm volatile may trigger a memory barrier (spilling all registers
    to memory).
  */
#ifdef __SUNPRO_CC
  asm ("pause" );
#else
  __asm__ __volatile__ ("pause");
#endif
#elif defined(_ARCH_PWR8)
  __ppc_get_timebase();
#elif defined __GNUC__ && (defined __arm__ || defined __aarch64__)
  /* Mainly, prevent the compiler from optimizing away delay loops */
  __asm__ __volatile__ (""::::"memory");
#else
  int32 var, oldval = 0;
  my_atomic_cas32_strong_explicit(&var, &oldval, 1, MY_MEMORY_ORDER_RELAXED,
                                  MY_MEMORY_ORDER_RELAXED);
#endif
}
```

e. 真实使用用例

i. 代码示例如下

```
// 编译添加-latomic参数
#include <atomic>
```

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

// 1.atomic_flag 保证一定是免锁的
std::atomic_flag flag_lock = ATOMIC_FLAG_INIT;

void f(int n) {
  // 如果只有一个原子变量的操作，不涉及代码前后的逻辑，可以使用内存序memory_order_relaxed
  // 但涉及多个原子变量，或者单个原子变量还要兼顾代码前后逻辑，要使用 memory_order_acquire/memory_order_release
  // 以原子方式将 std::atomic_flag 的状态更改为设置 (true) 并返回之前保存的值。
  while (flag_lock.test_and_set(std::memory_order_acquire))
    ;
  std::cout << "Thread " << n << " say!" << std::endl;
  flag_lock.clear(std::memory_order_release);  // 内存序同store
}

// 2.atomic<T>
// 并不一定是免锁的，例如内存对齐是原子的，那么错误对齐则必须使用锁。
// 非免锁情况：旧编译器、CPU不支持、非原子类型（vector,数组）、大对象、所有超过8字节不支持
// 总之原子操作适用于char、int、long、double等原子类型上，自定义类型慎用！！！
struct A {
  int a[100];
};
struct B {
  int x, y;
};
void TestIsLockFree() {
  std::atomic<A> a;
  std::atomic<B> b;
  std::atomic<double> c;
  std::cout << a.is_lock_free() << std::endl;
  std::cout << b.is_lock_free() << std::endl;
  std::cout << c.is_lock_free() << std::endl;
}

// 3.一般原子操作
// 原子变量不可以直接相互赋值，需要转换成T在赋值。
void TestAtomic() {
```

```
  std::atomic<int> a{};
  std::cout << a.load(std::memory_order_relaxed) << std::endl;

  std::atomic<int> b{};
  // b = a; error
  b.store(a.load(std::memory_order_acquire), std::memory_order_release);
}

// 4.内存序
// store memory_order_relaxed release seq_cst
// load memory_order_acquire consume acquire seq_cst
std::atomic<int> foo(0);
std::atomic<int> bar(0);
void SetFoo(int x) { foo.store(x, std::memory_order_relaxed); }
void CopyFooToBar() {
  while (foo.load(std::memory_order_acquire) == 0) std::this_thread::yield();
  bar.store(foo.load(std::memory_order_relaxed), std::memory_order_release);
}
void PrintBar() {
  while (bar.load(std::memory_order_relaxed) == 0) std::this_thread::yield();
  std::cout << "bar: " << bar.load(std::memory_order_relaxed) << std::endl;
}
void TestMemoryOrder() {
  std::thread t1(PrintBar);
  std::thread t2(CopyFooToBar);
  std::thread t3(SetFoo, 10);
  t1.join();
  t2.join();
  t3.join();
}

// 5.compare_exchange_weak compare_exchange_strong
// CAS，原子比较expect和当前值，如果相同则用value代替，返回success
// 如果不同则当前值替代expect，返回false
// success内存序支持所有值，failure不支持若比较失败，则加载操作所用的内存同步顺序。不能为
// std::memory_order_release 或 std::memory_order_acq_rel
// weak支持比较值相同但返回false的情况，适用于循环。
struct ListNode {
  int value;
  ListNode* next;
```

```
};
std::atomic<ListNode*> head(nullptr);
void append(int value) {
  ListNode* old_head = head.load(std::memory_order_relaxed);
  ListNode* new_node = new ListNode{value, old_head};

  while (!head.compare_exchange_weak(old_head, new_node,
                                     std::memory_order_release,
                                     std::memory_order_acquire)) {
    new_node->next = old_head;
  }
}
void TestAppend() {
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) threads.emplace_back(append, i);
  for (auto& t : threads) t.join();

  ListNode* temp = head.load(std::memory_order_relaxed);
  while (temp) {
    std::cout << temp->value << "->";
    temp = temp->next;
  }
  std::cout << std::endl;

  while (temp = head.load(std::memory_order_relaxed)) {
    head.store(temp->next, std::memory_order_relaxed);
    delete temp;
  }
}

// 6.比较和复制是逐位的，类似memcmp\memcpy，不适用构造函数、赋值运算符和比较运算符

int main() {
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; i++) {
    threads.emplace_back(f, i);
  }
  for (auto& t : threads) {
    t.join();
```

```
    }

    TestIsLockFree();
    TestAtomic();
    TestMemoryOrder();
    TestAppend();
    return 0;
}
```

5. STL-顺序容器

a. priority_queue

i. 原理：底层使用堆的思想，以vector为容器。

6. 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。

7. 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。
3. 常应用于在使用constexpr表达式或者costexpr函数的地方使用字面常量类。

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。

```
#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;
}

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：

```
#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

```
#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;
  }
}
```

```
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 }
```

c. enable_shared_from_this<>

i. 用以在类内部函数中获取当前类对象的shard_ptr指针，可以在类内部的函数中将对象传给其他啊业务逻辑，多用于异步调用将自身传递给异步函数。

ii. 不能shared_from_this的时候（局部变量调用）会抛出异常！！

iii. 这是一种侵入式设计！！！！

iv. 不能在构造函数中使用share_from_this()！！！！

```
1  template<typename _Tp>
2  class enable_shared_from_this
3  {
4  public:
5      shared_ptr<_Tp> shared_from_this()
6      { return shared_ptr<_Tp>(this->_M_weak_this); }
7
8      shared_ptr<const _Tp> shared_from_this() const
9      { return shared_ptr<const _Tp>(this->_M_weak_this); }
10
11     weak_ptr<_Tp> weak_from_this() noexcept
12     { return this->_M_weak_this; }
13
14     weak_ptr<const _Tp> weak_from_this() const noexcept
15     { return this->_M_weak_this; }
16
17 protected:
18     template<typename _Tp1>
19     void _M_weak_assign(_Tp1* __p, const __shared_count<>& __n) const noexcept
```

```
    { _M_weak_this._M_assign(__p, __n); }

private:
    mutable weak_ptr<_Tp>  _M_weak_this;
};

template<typename _Yp, typename = _SafeConv<_Yp>>
explicit __shared_ptr(_Yp* __p) : _M_ptr(__p), _M_refcount(__p, typename
is_array<_Tp>::type())
{
    _M_enable_shared_from_this_with(__p);
}

template<typename _Yp, typename _Yp2 = typename remove_cv<_Yp>::type>
typename enable_if<__has_esft_base<_Yp2>::value>::type
_M_enable_shared_from_this_with(_Yp* __p) noexcept {
    // 重点在这，判断__P类型是不是enable_shared_from_this的子类，如果是，则给
enable_shared_from_this中的weak_ptr赋值
    if (auto __base = __enable_shared_from_this_base(_M_refcount, __p))
        __base->_M_weak_assign(const_cast<_Yp2*>(__p), _M_refcount);
    }
}
```

d. std::make_shared

i. 优势在于一次性分配内存，，在分配计数器空间的空时，多分配一块空间给数据用，让数据和计数器在连续的内存区域，对 CPU 缓存也友好。

8. c++17特性

i. byte类型

```
#include <bitset>
#include <cstddef>
#include <iostream>

template <typename T>
inline std::bitset<sizeof(T) * 8> to_bitset(const std::byte& b) {
  return std::bitset<sizeof(T) * 8>(std::to_integer<T>(b));
}

```

```
int main() {
  std::byte b1{0x3F};
  std::byte b2{0b1111'0000};
  std::cout << std::to_integer<int32_t>(b1) << std::endl;
  std::cout << std::to_integer<int32_t>(b2) << std::endl;

  b1 <<= 1;
  b2 >>= 4;
  std::cout << std::to_integer<int32_t>(b1) << std::endl;
  std::cout << std::to_integer<int32_t>(b2) << std::endl;

  std::byte b3 = b1 ^ b2;
  std::cout << std::to_integer<int32_t>(b3) << std::endl;
  std::cout << to_bitset<int8_t>(b1) << std::endl;
  std::cout << to_bitset<int8_t>(b2) << std::endl;
  std::cout << to_bitset<int8_t>(b3) << std::endl;
}

```

ii. optional

```
#include <iomanip>
#include <iostream>
#include <optional>
#include <string>
#include <vector>

int main() {
  auto op1 = std::make_optional<std::vector<char>>({'a', 'b', 'c'});
  std::cout << "op1: ";
  for (char c : op1.value()) {
    std::cout << c << ",";
  }
  auto op2 = std::make_optional<std::vector<int>>(5, 2);
  std::cout << "\nop2: ";
  for (int i : *op2) {
    std::cout << i << ",";
  }
```

```
  std::string str{"hello world"};
  auto op3 = std::make_optional<std::string>(std::move(str));
  std::cout << "\nop3: " << quoted(op3.value_or("empty value")) << '\n';
  std::cout << "str: " << std::quoted(str) << '\n';

  // make_optional的返回值不是std::nullopt，make_optional会调用默认构造函数
  auto op4 = std::make_optional<int>();
  std::cout << (op4 == std::nullopt ? -999 : *op4) << std::endl;
  std::optional<int> op5;
  std::cout << (op5 == std::nullopt ? -999 : *op5) << std::endl;
}

```

iii.

9. 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]) {/* ... */};
```

10. STL-Algorithms

a. for_each

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first)
        f(*first);
    return f;
}
```

```cpp

#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> v{3, -4, 2, -8, 15, 267};

  auto print = [](const int& n) { std::cout << n << ' '; };

  std::cout << "before:\t";
  std::for_each(v.cbegin(), v.cend(), print);
  std::cout << '\n';

  // increment elements in-place
  std::for_each(v.begin(), v.end(), [](int& n) { n++; });

  std::cout << "after:\t";
  std::for_each(v.cbegin(), v.cend(), print);
  std::cout << '\n';

  // 还能这么用，厉害！！
  struct Sum {
    void operator()(int n) { sum += n; }
    int sum{0};
  };
  Sum s = std::for_each(v.cbegin(), v.cend(), Sum());
  std::cout << "sum:\t" << s.sum << '\n';
}
```

b. count

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>

int main() {
```

```
  constexpr std::array<int, 10> v{1, 2, 3, 4, 4, 3, 7, 8, 9, 10};
  std::cout << "v: ";
  std::copy(v.cbegin(), v.cend(), std::ostream_iterator<int>(std::cout, " "));
  std::cout << '\n';

  // determine how many integers match a target value.
  for (const int target : {3, 4, 5}) {
    const int num_items = std::count(v.cbegin(), v.cend(), target);
    std::cout << "number: " << target << ", count: " << num_items << '\n';
  }

  // use a lambda expression to count elements divisible by 4.
  int count_div4 =
      std::count_if(v.begin(), v.end(), [](int i) { return i % 4 == 0; });
  std::cout << "numbers divisible by four: " << count_div4 << '\n';
}
```

c. find && binary_search

i. bool binary_search(iterator beg,iterator end,value);返回值是bool，不是迭代器

```
#include <iostream>
#include <vector>

// std库里的find就是顺序查找，因此适用于顺序容器。
// 对于map、set等容器最好使用自身带的find，效率高。
template <class Iter, class T>
Iter my_find(Iter begin, Iter end, const T& t) {
  for (; begin != end; ++begin) {
    if (*begin == t) return begin;
  }
  return end;
}

template <class Iter, class UnaryPredicate>
Iter my_find_if(Iter begin, Iter end, UnaryPredicate p) {
  for (; begin != end; ++begin) {
    if (p(*begin)) return begin;
  }
```

```
  return end;
}

int main() {
  std::vector<int> v{1, 2, 3, 4};
  int n1 = 3;
  int n2 = 5;
  auto is_even = [](int i) { return i % 2 == 0; };

  auto result1 = my_find(begin(v), end(v), n1);
  auto result2 = my_find(begin(v), end(v), n2);
  auto result3 = my_find_if(begin(v), end(v), is_even);

  (result1 != std::end(v)) ? std::cout << "v contains " << n1 << '\n'
                           : std::cout << "v does not contain " << n1 << '\n';

  (result2 != std::end(v)) ? std::cout << "v contains " << n2 << '\n'
                           : std::cout << "v does not contain " << n2 << '\n';

  (result3 != std::end(v))
      ? std::cout << "v contains an even number: " << *result3 << '\n'
      : std::cout << "v does not contain even numbers\n";
}
```

d. sort

i. 数据量大时采用快速排序 Quick Sort，分段递归排序。一旦分段后的数据量小于某个阈值，为避免 Quick Sort的递归调用带来过大的额外开销，就改用插入排序 Insertion Sort。如果递归层次过深，还会改用堆排序 Heap Sort。

```
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>

template <class Iter>
void fill_with_random_int_values(Iter start, Iter end, int min, int max) {
  static std::random_device rd;   // you only need to initialize it once
  static std::mt19937 mte(rd());  // this is a relative big object to create
```

```
10
11    std::uniform_int_distribution<int> dist(min, max);
12    std::generate(start, end, [&]() { return dist(mte); });
13  }
14
15  template <class T>
16  void print(T t) {
17    std::for_each(t.begin(), t.end(),
18                  [](const auto& i) { std::cout << i << " "; });
19    std::cout << std::endl;
20  }
21
22  // 优先使用函数对象定义对比规则，此规则按升序排序
23  class MyComp {
24   public:
25    bool operator()(const int32_t& i, const int32_t& j) { return (i < j); }
26  };
27
28  int main() {
29    // init
30    std::vector<int32_t> a(10, 0);
31    fill_with_random_int_values(a.begin(), a.end(), 0, 1000);
32    print(a);
33
34    // example 1
35    // greater or less 说明的是大顶堆还是小顶堆，默认less小顶堆
36    std::sort(a.begin(), a.end(), std::greater<int32_t>());
37    print(a);
38
39    // example 2
40    fill_with_random_int_values(a.begin(), a.end(), 0, 1000);
41    std::sort(a.begin(), a.end(), MyComp());
42    print(a);
43
44    // example 3
45    fill_with_random_int_values(a.begin(), a.end(), 0, 1000);
46    print(a);
47    std::partial_sort(a.begin(), a.begin() + 4, a.end(), std::greater<int32_t>());
48    print(a);
49
```

```
  return 0;
}

```

ii. stable_sort排序不改变相同值的位置，但sort可能会改变相同值的位置。

iii.

e. 汇总代码

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>

void print1(const int& val) { std::cout << val << '\t'; }

// 仿函数
class print2 {
 public:
  void operator()(const int& val) { std::cout << val << '\t'; }
};

// for_each
void test01() {
  std::vector<int> a = {1, 2, 3, 4};

  std::for_each(a.begin(), a.end(), print1);
  std::cout << std::endl;
  std::for_each(a.begin(), a.end(), print2());
  std::cout << std::endl;
}

// transform
class Transform {
 public:
  int operator()(const int& val) { return val + 10; }
};

void test02() {
```

```
  std::vector<int> a = {1, 2, 3, 4};
  std::vector<int> b;
  b.resize(a.size());

  std::transform(a.begin(), a.end(), b.begin(), Transform());
  std::for_each(b.begin(), b.end(), print2());
  std::cout << std::endl;
}

// find && find_if
class GreaterFive {
 public:
  bool operator()(const int& val) { return val > 5; }
};

void test03() {
  std::vector<int> a = {1, 2, 3, 4, 5, 6};
  auto iter = std::find_if(a.begin(), a.end(), GreaterFive());
  if (iter == a.end()) {
    std::cout << "not find greater five" << std::endl;
  } else {
    std::cout << *iter << std::endl;
  }
}

// count 和 count_if 使用方式和 find find_if相似，不在赘述

// binary search
void test04() {
  std::vector<int> a = {1, 2, 3, 4, 5, 6};
  auto ret = std::binary_search(a.begin(), a.end(), 4);
  if (ret) {
    std::cout << "binary search find 4" << std::endl;
  } else {
    std::cout << "binary search not find 4" << std::endl;
  }
}

// merge后会自动进行排序
void test05() {
```

```
  std::vector<int> a = {1, 2, 3, 5, 7, 9};
  std::vector<int> b = {4, 6, 8, 10};

  std::vector<int> c;
  c.resize(a.size() + b.size());
  std::merge(a.begin(), a.end(), b.begin(), b.end(), c.begin());
  std::for_each(c.begin(), c.end(), print2());
  std::cout << std::endl;

  std::reverse(c.begin(), c.end());
  std::for_each(c.begin(), c.end(), print2());
  std::cout << std::endl;
}

// copy
void test06() {
  std::vector<int> a = {1, 2, 3, 4, 5, 6};
  std::vector<int> b;
  b.resize(a.size());
  std::copy(a.begin(), a.end(), b.begin());
  std::for_each(b.begin(), b.end(), print2());
  std::cout << std::endl;
}

// replace && replace_if && swap
void test07() {
  std::vector<int> a = {1, 2, 3, 4, 5, 6};
  std::replace(a.begin(), a.end(), 5, 50);
  std::for_each(a.begin(), a.end(), print2());
  std::cout << std::endl;

  std::replace_if(a.begin(), a.end(), GreaterFive(), 100);
  std::for_each(a.begin(), a.end(), print2());
  std::cout << std::endl;

  int b = 1;
  int c = 2;
  std::swap(b, c);
  std::cout << b << '\t' << c << std::endl;
```

```
110
111   std::vector<int> d = {10, 2, 3};
112   std::swap(a, d);
113   std::for_each(a.begin(), a.end(), print2());
114   std::cout << std::endl;
115 }
116
117 // accumulate && fill
118 void test08() {
119   std::vector<int> a = {1, 2, 3, 4, 5, 6};
120   int total = std::accumulate(a.begin(), a.end(), 0);
121   std::cout << total << std::endl;
122
123   std::vector<int> b(10, 0);
124   std::fill(b.begin(), b.end(), 10);
125   std::for_each(b.begin(), b.end(), print2());
126   std::cout << std::endl;
127 }
128
129 // set function
130 void test09() {
131   // 两个集合必须有序！！！！
132   std::vector<int> a = {1, 3, 3, 5};
133   std::vector<int> b = {1, 2, 3, 9};
134
135   std::vector<int> c(std::min(a.size(), b.size()), 0);
136   auto iter =
137       std::set_intersection(a.begin(), a.end(), b.begin(), b.end(), c.begin());
138   std::for_each(c.begin(), iter, print2());
139   std::cout << std::endl;
140
141   std::vector<int> d(a.size() + b.size(), 0);
142   iter = std::set_union(a.begin(), a.end(), b.begin(), b.end(), d.begin());
143   std::for_each(d.begin(), iter, print2());
144   std::cout << std::endl;
145
146   std::vector<int> e(a.size(), 0);
147   iter = std::set_difference(a.begin(), a.end(), b.begin(), b.end(), e.begin());
148   std::for_each(e.begin(), iter, print2());
149   std::cout << std::endl;
```

```
}

int main() {
  test01();
  test02();
  test03();
  test04();
  test05();
  test06();
  test07();
  test08();
  test09();
  return 0;
}
```

11. boos库

a. 使用注意

i. 在使用bazel编译的时候，明明有lib库，但是找不到，可以在linkopts里加入"-L/usr/local/lib"

```
cc_binary (
    name = "coroutine",
    srcs = [
        "coroutine.cc",
    ],
    linkopts = [
        "-L/usr/local/lib",
        "-lboost_coroutine",
        "-lboost_context",
    ],
)

```

b. filesystem

i. STL在c++17后已经集成了boost的filesystem功能，关于文件系统的操作可以使用filesystem库函数，对于文件的操作还是要使用

```cpp
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main() {
  fs::path a("/home/heshi/data/test/main.cc");
  std::cout << fs::file_size(a) << std::endl;

  fs::path root("/");
  fs::space_info root_space_info = fs::space(root);
  std::cout << root_space_info.capacity << std::endl;
  std::cout << root_space_info.free << std::endl;
  std::cout << root_space_info.available << std::endl;
  return 0;
}

```

c. 智能指针

i. STL集成的unique_ptr和shared_ptr已经集成了boost库中大多数智能指针的需求，可以使用STL的。

d. Container

i. stable_vector

1. 稳定的容器，只要元素不被删除，迭代器和引用就能用。

```cpp
#include <iostream>

#include "boost/container/stable_vector.hpp"

std::ostream& operator<<(std::ostream& out,
                         const boost::container::stable_vector<int>& nums) {
  for (const auto& n : nums) {
    out << n << " ";
  }
  return out;
}

```

```cpp
int main() {
  boost::container::stable_vector<int> v = {1, 2, 3, 4, 5};
  std::cout << v.size() << " " << v.capacity() << std::endl;
  auto iter1 = v.begin() + 2;
  auto iter2 = v.begin() + 3;
  std::cout << *iter1 << " " << *iter2 << std::endl;
  v.insert(iter1, 7);
  std::cout << v << std::endl;
  std::cout << *iter1 << " " << *iter2 << std::endl;
  std::cout << v.size() << " " << v.capacity() << std::endl;
  iter1 = v.erase(iter1);
  std::cout << v.size() << " " << v.capacity() << std::endl;
  std::cout << *iter1 << " " << *iter2 << std::endl;
  return 0;
}

```

ii. flat_(multi)map/set

1. 以向量的方式存储数据，数据在存储的时候是有序的，所有可以通过二分法进行查找，但插入和删除效率低，涉及空间的申请和容器内已有元素的拷贝。

```cpp
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <set>

#include "boost/container/flat_set.hpp"

template <typename F, typename... Args>
void funcCast(F&& f, Args&&... args) {
  auto start = std::chrono::high_resolution_clock::now();
  f(std::forward<Args>(args)...);
  auto end = std::chrono::high_resolution_clock::now();
  std::cout << "cast: "
            << std::chrono::duration_cast<std::chrono::microseconds>(end -
                                                                     start)
                   .count()
```

```
            << std::endl;
}

int main() {
  boost::container::flat_set<int> s;
  funcCast(
      [&](int range) {
        srand(static_cast<int>(time(0)));
        for (int i = 0; i < range; i++) {
          s.emplace(static_cast<int>(rand() % range));
        }
      },
      1000000);
  funcCast(
      [&](int dest) {
        auto iter = s.find(dest);
        if (iter != s.end()) {
          std::cout << *iter << std::endl;
        } else {
          std::cout << "not find" << std::endl;
        }
      },
      12345);

  std::set<int> s2;
  funcCast(
      [&](int range) {
        srand(static_cast<int>(time(0)));
        for (int i = 0; i < range; i++) {
          s2.emplace(static_cast<int>(rand() % range));
        }
      },
      1000000);
  funcCast(
      [&](int dest) {
        auto iter = s2.find(dest);
        if (iter != s2.end()) {
          std::cout << *iter << std::endl;
        } else {
```

```
            std::cout << "not find" << std::endl;
        }
      },
      12345);
  return 0;
}

```

12. 线程独占资源（线程上下文）

a. 栈帧

i. 函数返回值；调用其他函数的参数；该函数使用的局部变量；函数使用的寄存器信息。

b. 程序计数器

i. CPU执行指令的信息保存在程序计数器中，通过寄存器可以知道下次要执行那一条命令。

c. 寄存器

13. 重载算数运算符

```
#include <iostream>
#include <vector>

class MyClass {
 public:
  MyClass() : a_(0){};
  MyClass(const MyClass& rhf) : data_(rhf.data_), a_(rhf.a_) {}
  MyClass(MyClass&& rhf) : data_(std::move(rhf.data_)), a_(std::move(rhf.a_)) {}
  ~MyClass() = default;
  MyClass& operator=(const MyClass& rhf) {
    data_ = rhf.data_;
    a_ = rhf.a_;
    return *this;
  }
  MyClass& operator=(MyClass&& rhf) {
    data_ = std::move(rhf.data_);
    a_ = std::move(rhf.a_);
    return *this;
  }

```

```
21  void emplace_back(const int& value) { data_.emplace_back(value); }
22
23  MyClass& operator++() {
24    ++a_;
25    return *this;
26  }
27  MyClass operator++(int) {
28    MyClass tmp(*this);
29    ++a_;
30    return tmp;
31  }
32  MyClass& operator--() {
33    --a_;
34    return *this;
35  }
36  MyClass operator--(int) {
37    MyClass tmp(*this);
38    --a_;
39    return tmp;
40  }
41  MyClass& operator+=(const MyClass& rhf) {
42    a_ += rhf.a_;
43    return *this;
44  }
45  MyClass& operator-=(const MyClass& rhf) {
46    a_ -= rhf.a_;
47    return *this;
48  }
49
50  // 重载指针运算符号，返回的是指针
51  MyClass* operator->() { return this; }
52  // 重载解引用运算符，返回的是引用
53  MyClass& operator*() { return *this; }
54
55  // 一般重载双目运算符为friend
56  friend std::ostream& operator<<(std::ostream& out, const MyClass& r) {
57    for (const auto& d : r.data_) {
58      std::cout << d << "->";
59    }
60    std::cout << "||" << r.a_;
```

```
    return out;
  }
  // 重载算，要限定不被当做左值用就加const
  friend const MyClass operator+(const MyClass& a, const MyClass& b) {
    MyClass tmp;
    tmp.a_ = a.a_ + b.a_;
    return tmp;
  }
  friend const MyClass operator-(const MyClass& a, const MyClass& b) {
    MyClass tmp;
    tmp.a_ = a.a_ - b.a_;
    return tmp;
  }

 private:
  std::vector<int> data_;
  int a_;
};

int main() {
  MyClass a;
  std::cout << a++ << ++a << a << std::endl;
  std::cout << --a << a-- << a << std::endl;
  a->emplace_back(9);
  std::cout << a << std::endl;
  std::cout << *a << std::endl;

  MyClass b;
  ++b;
  a += b + b;
  std::cout << a << std::endl;
  a -= b;
  std::cout << a << std::endl;
  std::cout << a - b << std::endl;

  return 1;
}
```

14. 其他

a. c++临时变量在表达式结束的时候被释放（析构）。

设计模式

1. 创建模式

a. 抽象工厂模式

i. 为一系列产品（逻辑上或业务上相关或有交互的的一些具体类）创建实体工厂，进而实例化每一个类对象。

```
// bastract_factory.cc
#include <iostream>
#include <memory>

class AbstractProductA {
 public:
  virtual ~AbstractProductA(){};
  virtual void UseFunctionA() = 0;
};

class ConcreteProductA1 : public AbstractProductA {
 public:
  ConcreteProductA1() { std::cout << "ConcreteProductA1 create" << std::endl; }
  void UseFunctionA() override {
    std::cout << "ConcreteProductA1 UseFunctionA" << std::endl;
  }
};

class ConcreteProductA2 : public AbstractProductA {
 public:
  ConcreteProductA2() { std::cout << "ConcreteProductA2 create" << std::endl; }
  void UseFunctionA() override {
    std::cout << "ConcreteProductA2 UseFunctionA" << std::endl;
  }
};

class AbstractProductB {
 public:
```

```
29   virtual ~AbstractProductB(){};
30   virtual void UseFunctionB() = 0;
31 };
32
33 class ConcreteProductB1 : public AbstractProductB {
34  public:
35   ConcreteProductB1() { std::cout << "ConcreteProductB1 create" << std::endl; }
36   void UseFunctionB() override {
37     std::cout << "ConcreteProductB1 UseFunctionB" << std::endl;
38   }
39 };
40
41 class ConcreteProductB2 : public AbstractProductB {
42  public:
43   ConcreteProductB2() { std::cout << "ConcreteProductB2 create" << std::endl; }
44   void UseFunctionB() override {
45     std::cout << "ConcreteProductB2 UseFunctionB" << std::endl;
46   }
47 };
48
49 class AbstractFactory {
50  public:
51   virtual ~AbstractFactory(){};
52   virtual std::unique_ptr<AbstractProductA> CreateProductA() const = 0;
53   virtual std::unique_ptr<AbstractProductB> CreateProductB() const = 0;
54 };
55
56 class Type1Factory : public AbstractFactory {
57  public:
58   std::unique_ptr<AbstractProductA> CreateProductA() const {
59     return std::make_unique<ConcreteProductA1>();
60   }
61   std::unique_ptr<AbstractProductB> CreateProductB() const {
62     return std::make_unique<ConcreteProductB1>();
63   }
64 };
65
66 class Type2Factory : public AbstractFactory {
67  public:
68   std::unique_ptr<AbstractProductA> CreateProductA() const {
```

```
    return std::make_unique<ConcreteProductA2>();
  }
  std::unique_ptr<AbstractProductB> CreateProductB() const {
    return std::make_unique<ConcreteProductB2>();
  }
};

void ClientCode(const AbstractFactory& factory) {
  std::unique_ptr<AbstractProductA> product_a = factory.CreateProductA();
  std::unique_ptr<AbstractProductB> product_b = factory.CreateProductB();
  product_a->UseFunctionA();
  product_b->UseFunctionB();
}

int main() {
  std::cout << "test bastract factory" << std::endl;
  std::unique_ptr<AbstractFactory> type1_factory =
      std::make_unique<Type1Factory>();
  ClientCode(*type1_factory);

  std::unique_ptr<AbstractFactory> type2_factory =
      std::make_unique<Type2Factory>();
  ClientCode(*type2_factory);
  return 1;
}
```

b. 生成器模式

i. 将对象构造代码从产品类中抽取出来， 并将其放在一个名为生成器的独立对象中。每次创建对象时，需要通过生成器对象执行一系列步骤。 重点在于你无需调用所有步骤， 而只需调用创建特定对象配置所需的那些步骤即可。

ii. 引用场景：类的构造参数多，并不是每次创建对象都会用到所有参数；分步骤构造复杂对象。

iii. 主管类可定义创建步骤的执行顺序， 而生成器则提供这些步骤的实现。

iv. 就跟工厂流水线一样，一步一步的来构造复杂对象。

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

```
5
6  // Product1承载所有模块，因为是不确定变量，所以用vector表示并存储
7  class Product1 {
8   public:
9    void insert(const std::string& part) { parts_.emplace_back(part); }
10
11   friend std::ostream& operator<<(std::ostream& out, const Product1& p) {
12     for (const auto& part : p.parts_) {
13       out << part << " | ";
14     }
15     return out;
16   }
17
18  private:
19   std::vector<std::string> parts_;
20 };
21
22 // 所有Builder的接口，外观模式
23 class Builder {
24  public:
25   virtual ~Builder(){};
26   virtual void ProducePartA() const = 0;
27   virtual void ProducePartB() const = 0;
28   virtual void ProducePartC() const = 0;
29 };
30
31 // 真实的Builder，跟流水线一样，负责插入各模块
32 class Builder1 : public Builder {
33  public:
34   Builder1() : product_(std::make_unique<Product1>()) {}
35   std::unique_ptr<Product1> GetProduct() {
36     auto tmp = std::make_unique<Product1>();
37     product_.swap(tmp);
38     return tmp;
39   }
40
41   void ProducePartA() const override { product_->insert("PartA1"); }
42
43   void ProducePartB() const override { product_->insert("PartB1"); }
```

```
44
45    void ProducePartC() const override { product_->insert("PartC1"); }
46
47   private:
48    std::unique_ptr<Product1> product_;
49  };
50
51  // Director可以指定插入模块流程，但真实插入模块还是Builder承载
52  class Director {
53   public:
54    Director() : builder_(nullptr) {}
55
56    void SetBuilder(const std::shared_ptr<Builder>&& builder) {
57      builder_ = builder;
58    }
59
60    void BuildMiniProduct() {
61      if (!builder_) return;
62
63      builder_->ProducePartA();
64    }
65
66    void BuildFullProduct() {
67      if (!builder_) return;
68
69      builder_->ProducePartA();
70      builder_->ProducePartB();
71      builder_->ProducePartC();
72    }
73
74   private:
75    std::shared_ptr<Builder> builder_;
76  };
77
78  int main() {
79    // use director
80    auto director = std::make_unique<Director>();
81    auto builder1 = std::make_shared<Builder1>();
82    director->SetBuilder(builder1);
83    director->BuildMiniProduct();
```

```
  auto product_mini = builder1->GetProduct();
  std::cout << *product_mini << std::endl;

  director->BuildFullProduct();
  auto product_full = builder1->GetProduct();
  std::cout << *product_full << std::endl;

  builder1->ProducePartB();
  builder1->ProducePartC();
  auto product_auto = builder1->GetProduct();
  std::cout << *product_auto << std::endl;
  return 0;
}

```

2. 结构模式

a. 适配器模式

i. 适配器类需要继承或依赖已有的类，实现想要的目标接口。

ii. 说白了是用已有接口实现自己的功能，大多数代码就是这个逻辑。

```
#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();
```

```
58
59    return 0;
60  }
61
```

b. 桥接模式

i. 桥接模式通过将继承改为组合的方式，解决多维度交叉的问题。

ii. 当需要拆分或者重组代码结构的时候（可以与多个书库进行交互的类）。

iii. 从多个独立维度上扩展类。

iv. 在运行时切换不同的实现方法。

```
1  #include <iostream>
2  #include <memory>
3  #include <string>
4
5  class Implementation {
6   public:
7    virtual ~Implementation() = default;
8    virtual std::string OperationImplementation() const = 0;
9  };
10
11 class ImplementationA : public Implementation {
12  public:
13   std::string OperationImplementation() const override {
14     return "Operation ImplementationA";
15   }
16 };
17
18 class ImplementationB : public Implementation {
19  public:
20   std::string OperationImplementation() const override {
21     return "Operation ImplementationB";
22   }
23 };
24
25 class Abstraction {
26  public:
27   Abstraction(std::unique_ptr<Implementation>&& i) : impl_(std::move(i)) {}
```

```
  virtual ~Abstraction() = default;

  virtual std::string Operation() const = 0;

 protected:
  std::unique_ptr<Implementation> impl_;
};

class AbstractionA : public Abstraction {
 public:
  AbstractionA(std::unique_ptr<Implementation>&& i)
      : Abstraction(std::move(i)) {}

  std::string Operation() const override {
    return "AbstractionA call, return : " + impl_->OperationImplementation();
  }
};

class AbstractionB : public Abstraction {
 public:
  AbstractionB(std::unique_ptr<Implementation>&& i)
      : Abstraction(std::move(i)) {}

  std::string Operation() const override {
    return "AbstractionB call, return : " + impl_->OperationImplementation();
  }
};

int main() {
  AbstractionA a(std::make_unique<ImplementationA>());
  std::cout << a.Operation() << std::endl;
  AbstractionA b(std::make_unique<ImplementationB>());
  std::cout << b.Operation() << std::endl;
  AbstractionB c(std::make_unique<ImplementationA>());
  std::cout << c.Operation() << std::endl;
  AbstractionB d(std::make_unique<ImplementationB>());
  std::cout << d.Operation() << std::endl;

  return 0;
}
```

```
68
```

c. 组合模式

i. 将对象组合成树状结构， 并且能像使用独立对象一样使用它们。

ii. 组合模式中所有元素公用一个接口，使用时不必在意细节。

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

class Component {
 public:
  virtual ~Component() = default;
  virtual void Add(const std::shared_ptr<Component>& component){};
  virtual void Remove(const std::shared_ptr<Component>& component){};

  virtual bool IsComposite() const { return false; }
  virtual std::string Operator() const = 0;
};

class Leaf : public Component {
 public:
  explicit Leaf(const std::string& name) : name_(name) {}
  std::string Operator() const override { return std::string("Leaf_") + name_; }

 private:
  using Component::Add;
  using Component::Remove;

 private:
  std::string name_;
};

class Composite : public Component {
 public:
  explicit Composite(const std::string& name) : name_(name) {}

```

```
  void Add(const std::shared_ptr<Component>& component) override {
    components_.emplace_back(component);
  }
  void Remove(const std::shared_ptr<Component>& component) override {
    components_.remove(component);
  }

  bool IsComposite() const override { return true; }
  std::string Operator() const override {
    std::string ret;
    for (const auto& component : components_) {
      ret += component->Operator() + std::string(",");
    }
    if (!ret.empty()) ret.erase(ret.end() - 1);
    return "[Composite " + name_ + ": " + ret + "]";
  }

 private:
  std::string name_;
  std::list<std::shared_ptr<Component>> components_;
};

int main() {
  auto leaf1 = std::make_shared<Leaf>("1");
  auto leaf2 = std::make_shared<Leaf>("2");
  auto leaf3 = std::make_shared<Leaf>("3");

  auto composite1 = std::make_shared<Composite>("1");
  auto composite2 = std::make_shared<Composite>("2");

  composite1->Add(leaf1);
  composite1->Add(leaf2);
  composite2->Add(leaf3);

  auto composite3 = std::make_shared<Composite>("3");
  composite3->Add(composite1);
  composite3->Add(composite2);
  std::cout << composite3->Operator() << std::endl;

  composite1->Remove(leaf1);
```

```
  std::cout << composite3->Operator() << std::endl;
}

```

d. 装饰模式

i. 就跟套娃一样，在不改变现有代码的前提下使用对象，且为对象新增功能。

ii. 如果通过继承对象（final）是不可能的，可以采用装饰模式。

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

std::string SERIALIZE_PREFIX("Serialize data: ");
std::string COMPRESS_PREFIX("Compress data: ");

class DataSource {
 public:
  virtual ~DataSource() = default;
  virtual void WriteData(const std::string& data) = 0;
  virtual std::string ReadData() const = 0;
};

class FileDataSource : public DataSource {
 public:
  FileDataSource(const std::string& path) : path_(path), data_(){};

  void WriteData(const std::string& data) override {
    data_ = data;
    std::cout << "Writer to file " << path_ << " data: [" << data_ << "]"
              << std::endl;
  }

  std::string ReadData() const override {
    std::cout << "Read from file " << path_ << " data: [" << data_ << "]"
              << std::endl;
    return data_;
  }

```

```
 private:
  std::string path_;
  std::string data_;
};

class DataSourceDecorator : public DataSource {
 public:
  DataSourceDecorator(const std::shared_ptr<DataSource>& data_source)
      : wrapper_(data_source) {}

  void WriteData(const std::string& data) override {
    wrapper_->WriteData(data);
  }

  std::string ReadData() const override { return wrapper_->ReadData(); }

 private:
  std::shared_ptr<DataSource> wrapper_;
};

class SerializeDecorator : public DataSourceDecorator {
 public:
  SerializeDecorator(const std::shared_ptr<DataSource>& data_source)
      : DataSourceDecorator(data_source) {}

  void WriteData(const std::string& data) override {
    std::string tmp = SERIALIZE_PREFIX + data;
    DataSourceDecorator::WriteData(tmp);
  }

  std::string ReadData() const override {
    std::string tmp = DataSourceDecorator::ReadData();
    return tmp.substr(SERIALIZE_PREFIX.length());
  }
};

class CompressDecorator : public DataSourceDecorator {
 public:
  CompressDecorator(const std::shared_ptr<DataSource>& data_source)
      : DataSourceDecorator(data_source) {}
```

```
71
72    void WriteData(const std::string& data) override {
73      std::string tmp = COMPRESS_PREFIX + data;
74      DataSourceDecorator::WriteData(tmp);
75    }
76
77    std::string ReadData() const override {
78      std::string tmp = DataSourceDecorator::ReadData();
79      return tmp.substr(COMPRESS_PREFIX.length());
80    }
81  };
82
83  int main() {
84    auto file = std::make_shared<FileDataSource>("/tmp/1.txt");
85    auto compress = std::make_shared<CompressDecorator>(file);
86    compress->WriteData("abcdefg");
87    std::cout << compress->ReadData() << std::endl;
88
89    // 最上层的应该写在外面封装
90    auto serialize = std::make_shared<SerializeDecorator>(compress);
91    serialize->WriteData("abcedfg");
92    std::cout << serialize->ReadData() << std::endl;
93
94    return 0;
95  }
96
```

e. 外观模式

i. 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。

ii. 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。

iii. 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。

iv. 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。

```
1  // facade_pattern.cc
2  #include <iostream>
3  #include <memory>
```

```cpp
#include <string>

class Subsystem1 {
 public:
  std::string Operator1() const { return "Subsystem1 Operator1"; }

  std::string OperatorN() const { return "Subsystem1 OperatorN"; }
};

class Subsystem2 {
 public:
  std::string Operator1() const { return "Subsystem2 Operator1"; }

  std::string OperatorN() const { return "Subsystem2 OperatorN"; }
};

class Facade {
 public:
  Facade(std::unique_ptr<Subsystem1>&& sub1, std::unique_ptr<Subsystem2>&& sub2)
      : subsystem1_(sub1 ? std::move(sub1) : std::make_unique<Subsystem1>()),
        subsystem2_(sub2 ? std::move(sub2) : std::make_unique<Subsystem2>()) {}

  std::string OperatorNew() {
    std::string ret("Facade operator: ");
    ret += subsystem1_->Operator1();
    ret += subsystem2_->OperatorN();
    ret += subsystem1_->OperatorN();
    ret += subsystem2_->Operator1();
    return ret;
  }

 private:
  std::unique_ptr<Subsystem1> subsystem1_;
  std::unique_ptr<Subsystem2> subsystem2_;
};

int main() {
  auto subsystem1 = std::make_unique<Subsystem1>();
  auto subsystem2 = std::make_unique<Subsystem2>();
```

```
  auto facade =
      std::make_unique<Facade>(std::move(subsystem1), std::move(subsystem2));
  std::cout << facade->OperatorNew() << std::endl;
  return 0;
}
```

3. 行为模式

a. 命令模式

b. 迭代器模式

i. 当集合背后有更为复杂的数据结构时，需要对客户隐藏细节，可以使用迭代器模式。

ii. 减少程序中重复的遍历代码。

iii. 具体怎么用还得结合来看。可以用迭代器的思想完成对复杂数据的遍历。

```
#include <iostream>
#include <vector>

// 1. 定义iterator类，类由真实数据指针和容器指针组成
// 由于容器类也是类模板，且由相同的T组成，因此先定义为U
template <typename T, typename U>
class Iterator {
 public:
  typedef typename std::vector<T>::iterator iter_type;
  Iterator(U* data, bool reverse = false) : data_(data) {
    it_ = data_->data_.begin();
  }
  Iterator(const Iterator& rhf) : data_(rhf.data_), it_(rhf.it_) {}

  Iterator& operator++() {
    ++it_;
    return *this;
  }
  Iterator operator++(int) {
    Iterator tmp(*this);
    ++it_;
    return tmp;
  }
  T& operator*() { return *it_; }
```

```
25    T* operator->() { return &(*it_); }
26    bool operator==(const Iterator& rhf) {
27      return data_ == rhf.data_ && it_ == rhf.it_;
28    }
29
30    bool IsEnd() { return it_ == data_->data_.end(); }
31
32   private:
33    U* data_;
34    iter_type it_;
35  };
36
37  // 2. 定义容器类，存放真实数据，模拟自定义数据结构
38  template <typename T>
39  class Container {
40   public:
41    void Add(const T& A) { data_.emplace_back(A); }
42    Iterator<T, Container> begin() { return Iterator<T, Container>(this); }
43
44   private:
45    std::vector<T> data_;
46    friend class Iterator<T, Container>;
47  };
48
49  // 3. 自定义类型
50  class Data {
51   public:
52    explicit Data(int a = 0) : data_(a) {}
53    int SetData(int a) {
54      data_ = a;
55      return data_;
56    }
57    int GetData() { return data_; }
58
59   private:
60    int data_;
61  };
62
63  // client code
64  void Client01() {
```

```
  Container<int> container;
  for (int i = 0; i < 10; ++i) {
    container.Add(i);
  }
  for (auto iter = container.begin(); !iter.IsEnd(); iter++) {
    std::cout << *iter << std::endl;
  }
}

void Client02() {
  Container<Data> container;
  Data a1(1), a2(2), a3(3);
  container.Add(a1);
  container.Add(a2);
  container.Add(a3);
  for (auto iter = container.begin(); !iter.IsEnd(); ++iter) {
    std::cout << iter->GetData() << std::endl;
  }
  for (auto iter = container.begin(); !iter.IsEnd(); ++iter) {
    std::cout << iter->SetData(8) << std::endl;
  }
}

int main() {
  Client01();
  Client02();
  return 0;
}
```

c. 中介者模式

i. 当一些对象和其他对象紧密耦合以至于难于修改，可以用中介模式。

ii. 当组件过度依赖其他组件而无法复用的时候。

iii. 为了能在不同情境下复用一些基本行为，导致需要建立大量子类的时候。

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

```

```
class BaseComponent;
// 基础的中介者类，声明各模块下发命令的通用接口
// 告诉中介者是哪个模块，下发的什么命令
class Mediator {
 public:
  virtual void Notify(const std::shared_ptr<BaseComponent>& sender,
                      const std::string& event) const = 0;
};

// 基础模块的基类，定义通用的设置中介者的方法
class BaseComponent {
 public:
  BaseComponent() : mediator_(nullptr) {}
  void SetMediator(const std::shared_ptr<Mediator>& mediator) {
    mediator_ = mediator;
  }

 protected:
  std::shared_ptr<Mediator> mediator_;
};

// 各个基础模块的抽象定义
class Component1 : public BaseComponent,
                   public std::enable_shared_from_this<Component1> {
 public:
  void DoA() {
    std::cout << "Component1 DoA" << std::endl;
    if (mediator_) mediator_->Notify(shared_from_this(), "A");
  }
  void DoB() {
    std::cout << "Component1 DoB" << std::endl;
    if (mediator_) mediator_->Notify(shared_from_this(), "B");
  }
};

// 各个基础模块的抽象定义
class Component2 : public BaseComponent,
                   public std::enable_shared_from_this<Component2> {
 public:
  void DoC() {
```

```
    std::cout << "Component2 DoC" << std::endl;
    if (mediator_) mediator_->Notify(shared_from_this(), "C");
  }
  void DoD() {
    std::cout << "Component2 DoD" << std::endl;
    if (mediator_) mediator_->Notify(shared_from_this(), "D");
  }
};


// 中介者的具体类
class ConcreteMediaror : public Mediator,
                         public std::enable_shared_from_this<ConcreteMediaror> {
 public:
  void SetComponent(const std::shared_ptr<Component1>& component1,
                    const std::shared_ptr<Component2>& component2) {
    component1_ = component1;
    component1_->SetMediator(shared_from_this());
    component2_ = component2;
    component2_->SetMediator(shared_from_this());
  }

  void Notify(const std::shared_ptr<BaseComponent>& sender,
              const std::string& event) const override {
    if (event == "A") {
      std::cout << "Mediator receive A, do next:" << std::endl;
      component2_->DoC();
    } else if (event == "D") {
      std::cout << "Mediator receive D, do next:" << std::endl;
      component1_->DoB();
      component2_->DoC();
    }
  }

 private:
  std::shared_ptr<Component1> component1_;
  std::shared_ptr<Component2> component2_;
};

```

```
void Client() {
  auto component1 = std::make_shared<Component1>();
  auto component2 = std::make_shared<Component2>();
  component1->DoA();
  component2->DoD();

  auto mediator = std::make_shared<ConcreteMediaror>();
  mediator->SetComponent(component1, component2);
  component1->DoA();
  component2->DoD();
}

int main() {
  Client();
  return 0;
}
```

d. 观察者模式

i. 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。

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

class Observer {
 public:
  explicit Observer(const std::string& name) : name_(name) {}
  virtual ~Observer(){};
  virtual void Update(const std::string& message) = 0;

 protected:
  std::string name_;
};

class Subject {
 public:
```

```
  virtual ~Subject(){};
  virtual void Attach(const std::shared_ptr<Observer>& observer) = 0;
  virtual void Detach(const std::shared_ptr<Observer>& observer) = 0;
  virtual void Notify(const std::string& message) = 0;
};

class SubjectA : public Subject {
 public:
  ~SubjectA() { std::cout << "Exit SubjectA" << std::endl; }
  void Attach(const std::shared_ptr<Observer>& observer) override {
    for (auto iter = list_observer_.begin(); iter != list_observer_.end();
         ++iter) {
      if (*iter == observer) {
        std::cout << "Observer already exit in subject" << std::endl;
        return;
      }
    }
    list_observer_.emplace_back(observer);
    std::cout << "Observer attach success" << std::endl;
  }
  void Detach(const std::shared_ptr<Observer>& observer) override {
    for (auto iter = list_observer_.begin(); iter != list_observer_.end();
         ++iter) {
      if (*iter == observer) {
        list_observer_.erase(iter);
        std::cout << "Observer detach success" << std::endl;
        return;
      }
    }
    std::cout << "Observer not exit in subject" << std::endl;
  }
  void Notify(const std::string& message) override {
    std::cout << "Notify " << list_observer_.size() << " observers"
              << std::endl;
    for (auto iter = list_observer_.begin(); iter != list_observer_.end();
         ++iter) {
      (*iter)->Update(message);
    }
  }

```

```
 private:
  std::list<std::shared_ptr<Observer>> list_observer_;
};

class ObserverA : public Observer {
 public:
  ObserverA(const std::string& name) : Observer(name) {}
  void Update(const std::string& message) override {
    std::cout << "ObserverA, name " << name_ << " receive: " << message
              << std::endl;
  }
};

void Test() {
  auto observer1 = std::make_shared<ObserverA>("ob1");
  auto observer2 = std::make_shared<ObserverA>("ob2");
  auto subject = std::make_shared<SubjectA>();
  subject->Attach(observer1);
  subject->Attach(observer2);
  subject->Notify("First message");

  subject->Detach(observer2);
  subject->Notify("Second message");
}

int main() {
  Test();
  return 0;
}
```

e. 策略模式

i. 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，本质是类里保存不同策略，可以通过设置策略来表现不同行为。

ii. 使用对象中各种不同的算法变体， 并希望能在运行时切换算法时， 可使用策略模式。

iii. 有许多仅在执行某些行为时略有不同的相似类时， 可使用策略模式。

iv. 总体来说就是有一组接口相同的类，但类的具体算法（业务逻辑）有区别，但在上层业务不区分具体

实现（算法），可以使用策略模式。

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

class Strategy {
 public:
  virtual ~Strategy() = default;
  virtual std::string Read(const int32_t& index) const = 0;
};

class StrategyA : public Strategy {
 public:
  std::string Read(const int32_t& index) const override {
    return "StrategyA read " + std::to_string(index);
  }
};

class StrategyB : public Strategy {
 public:
  std::string Read(const int32_t& index) const override {
    return "StrategyB read " + std::to_string(index);
  }
};

class Context {
 public:
  Context() : strategy_(std::make_unique<StrategyA>()) {}
  explicit Context(std::unique_ptr<Strategy>&& strategy)
      : strategy_(std::move(strategy)) {}

  void SetStrategy(std::unique_ptr<Strategy>&& strategy) {
    strategy_ = std::move(strategy);
  }

  void BusinessProcess(const int32_t& index) {
    std::cout << "BusinessProcess" << std::endl;
    std::cout << strategy_->Read(index) << std::endl;
```

```
  }

 private:
  std::unique_ptr<Strategy> strategy_;
};

void Test() {
  Context context;
  context.BusinessProcess(1);

  auto strategy_b = std::make_unique<StrategyB>();
  context.SetStrategy(std::move(strategy_b));
  context.BusinessProcess(2);
}

int main() {
  Test();
  return 0;
}

```

f. 状态模式

i. 对象需要根据自身当前状态进行不同行为， 同时状态的数量非常多且与状态相关的代码会频繁变更的话， 可使用状态模式。

ii. 某个类需要根据成员变量的当前值改变自身行为， 从而需要使用大量的条件语句时， 可使用状态模式。

iii. 相似状态和基于条件的状态机转换中存在许多重复代码时， 可使用状态模式。

iv. 核心点在于一个类的业务逻辑和当前所处状态有着十分紧密的联系，逻辑的运行依赖状态的转换！！

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

class Context;

class State {
 public:
```

```
  explicit State(std::unique_ptr<std::string>&& doc) : doc_(std::move(doc)) {}
  virtual ~State() = default;

  virtual bool Write(const std::string& message) { return false; }
  virtual std::string GetDoc() const { return {}; }

  virtual std::unique_ptr<State> Submit() { return nullptr; }
  virtual std::unique_ptr<State> Reject() { return nullptr; }

 protected:
  std::unique_ptr<std::string> doc_;
};

class StateDraft : public State {
 public:
  explicit StateDraft(std::unique_ptr<std::string>&& doc)
      : State(std::move(doc)) {}
  bool Write(const std::string& message) override;
  std::unique_ptr<State> Submit() override;
};

class StateModeration : public State {
 public:
  explicit StateModeration(std::unique_ptr<std::string>&& doc)
      : State(std::move(doc)) {}
  std::unique_ptr<State> Submit() override;
  std::unique_ptr<State> Reject() override;
};

class StatePublished : public State {
 public:
  explicit StatePublished(std::unique_ptr<std::string>&& doc)
      : State(std::move(doc)) {}
  std::string GetDoc() const override;
  std::unique_ptr<State> Reject() override;
};

bool StateDraft::Write(const std::string& message) {
  *doc_ += message;
  return true;
```

```
}
std::unique_ptr<State> StateDraft::Submit() {
  return std::make_unique<StateModeration>(std::move(doc_));
}

std::unique_ptr<State> StateModeration::Submit() {
  return std::make_unique<StatePublished>(std::move(doc_));
}
std::unique_ptr<State> StateModeration::Reject() {
  return std::make_unique<StateDraft>(std::move(doc_));
}

std::string StatePublished::GetDoc() const { return *doc_; }
std::unique_ptr<State> StatePublished::Reject() {
  return std::make_unique<StateDraft>(std::move(doc_));
}

class Context {
 public:
  Context()
      : state_(std::make_unique<StateDraft>(std::make_unique<std::string>())) {}

  void Write(const std::string& message) {
    if (state_->Write(message)) {
      std::cout << "Write success" << std::endl;
    } else {
      std::cout << "Curr state not support Write" << std::endl;
    }
  }

  void Read() {
    auto doc = state_->GetDoc();
    if (!doc.empty()) {
      std::cout << doc << std::endl;
    } else {
      std::cout << "Curr state not support Read" << std::endl;
    }
  }

  void Submit() {
```

```
    auto state = state_->Submit();
    if (!state) {
      std::cout << "Curr state not support Submit" << std::endl;
      return;
    }
    state_ = std::move(state);
  }

  void Reject() {
    auto state = state_->Reject();
    if (!state) {
      std::cout << "Curr state not support Reject" << std::endl;
      return;
    }
    state_ = std::move(state);
  }

 private:
  std::unique_ptr<State> state_;
};

void Test() {
  auto context = std::make_unique<Context>();
  context->Write("abcde");
  context->Submit();
  context->Read();
  context->Reject();
  context->Submit();
  context->Submit();
  context->Write("edfg");
  context->Read();
}

int main() {
  Test();
  return 0;
}

```

4. 模板模式

a. 定义一个操作中的算法的骨架（就是相同的过程），而将一些步骤延迟到子类中（虚函数），在调用过程中真实调用的是派生类。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。

b. 本质是把父类当做模板，定义公共过程。把子类当做模板的适配，父类指针或者引用指向子类对象，通过虚函数的方式实现对子类方法的调用。

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

class Computer {
 public:
  virtual ~Computer() {}

  // 定义通用的方法，不需要虚函数
  void product() {
    productCpu();
    productGpu();
    productDisplay();
  }

 private:
  virtual void productCpu() = 0;
  virtual void productGpu() = 0;
  virtual void productDisplay() = 0;
};

class ComputerA : public Computer {
 private:
  void productCpu() override { std::cout << "ComputerA_CpuA" << std::endl; }
  void productGpu() override { std::cout << "ComputerA_GpuA" << std::endl; }
  void productDisplay() override {
    std::cout << "ComputerA_DisplayA" << std::endl;
  }
};

class ComputerB : public Computer {
 private:
  void productCpu() override { std::cout << "ComputerB_CpuB" << std::endl; }
```

```
  void productGpu() override { std::cout << "ComputerB_GpuB" << std::endl; }
  void productDisplay() override {
    std::cout << "ComputerB_DisplayB" << std::endl;
  }
};

int main() {
  std::unique_ptr<Computer> computer = std::make_unique<ComputerA>();
  computer->product();

  computer = std::make_unique<ComputerB>();
  computer->product();
}

```

5. 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。

b. 本质是有一个公共类，代理类和真实类有相同表现，只不过代理类中保存了真实类的对象，代理类的方法调用真实类的方法，只是做中转。

```
#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;
};

class YongMan : public Profession {
 public:
  YongMan(const Girl& girl) : girl_(girl) {}
  virtual void profess() {
    std::cout << "Yong man name: " << girl_.GetName() << std::endl;
  }

 private:
  Girl girl_;
};

class ManProxy : public Profession {
 public:
  ManProxy(const Girl& girl) : p_man_(new YongMan(girl)) {}
  virtual void profess() { p_man_->profess(); }

 private:
  std::unique_ptr<YongMan> p_man_;
};

int main() {
  Girl girl("Hanmeimei");
  ManProxy man_proxy(girl);
  man_proxy.profess();

  return 0;
}

```

6. 单例模式

a. 单例模式-懒汉模式

```
#include <atomic>
#include <iostream>
```

```
#include <mutex>
#include <string>
#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;
}
```

```
82
```

7. 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，本质是类里保存不同策略，可以通过设置策略来表现不同行为。

```
#include <algorithm>
#include <iostream>
#include <memory>

class Strategy {
 public:
  virtual ~Strategy() = default;
  virtual std::string doAlgorithm(const std::string& data) const = 0;
};

class Context {
 public:
  explicit Context(std::unique_ptr<Strategy>&& strategy)
      : strategy_(std::move(strategy)) {}

  void setStrategy(std::unique_ptr<Strategy>&& startegy) {
    strategy_ = std::move(startegy);
  }

  void doBusiness() {
    std::cout << "doBusiness" << std::endl;
    auto ret = strategy_->doAlgorithm("sadfada");
    std::cout << "business ret: " << ret << std::endl;
  }

 private:
  std::unique_ptr<Strategy> strategy_;
};

class Strategy01 : public Strategy {
 public:
```

```
  std::string doAlgorithm(const std::string& data) const override {
    return data;
  }
};

class Strategy02 : public Strategy {
 public:
  std::string doAlgorithm(const std::string& data) const override {
    std::string ret(data);
    std::sort(ret.begin(), ret.end());
    return ret;
  }
};

int main() {
  Context context(std::make_unique<Strategy01>());
  context.doBusiness();

  context.setStrategy(std::make_unique<Strategy02>());
  context.doBusiness();

  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。非基类则不需要声明virtual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. operator=处理自我赋值：采用copy and swap的方案

```
#include <algorithm>
#include <iostream>

```

```cpp
class A {
 public:
  A() : data_(new int(0)) {}
  A(const int& data) : data_(new int(data)) {}
  A(const A& rhs) : data_(new int(*rhs.data_)) {}
  A(A&& rhs) : data_(rhs.data_) { rhs.data_ = nullptr; }

  ~A() {
    if (data_) {
      delete data_;
      data_ = nullptr;
    }
  }

  A& operator=(const A& rhs) {
    A temp(rhs);
    swap(temp);
    return *this;
  }

  A& operator=(A&& rhs) {
    A temp(std::move(rhs));
    swap(temp);
    return *this;
  }

  void swap(A& rhs) { std::swap(data_, rhs.data_); }

  friend std::ostream& operator<<(std::ostream& out, const A& a) {
    if (a.data_) {
      out << "data: " << *(a.data_);
    } else {
      out << "data is nullptr";
    }
    return out;
  }

 private:
  int* data_;
```

```
};

int main() {
  A a(1);
  std::cout << a << std::endl;

  A b;
  std::cout << b << std::endl;

  A c(std::move(a));
  std::cout << c << std::endl;
  std::cout << a << std::endl;

  b = c;
  std::cout << b << std::endl;
  std::cout << c << std::endl;

  b = std::move(c);
  std::cout << b << std::endl;
  std::cout << c << std::endl;

  std::cout << "bye" << std::endl;
  return 0;
}
```

f. 尽量以const，enum，inline替换#define，只要能替换就替换，除非替换不了。

g. 能加const的地方就加const，const &的方式可以不产生副本，没有额外开销。

h. 可以使用非const函数调用const函数，非const函数会有两次强制类型转换。

```
class MyString {
 public:
  const char& operator[](std::size_t position) const {
    ... // 边界检查
    ... // 日志记录
    ... // 数据完整性校验
    return text[position];
  }

```

```
  char& operator[](std::size_t position) {
    return const_cast<char&>(static_cast<const MyString&>(*this)[position]);
  }
}
```

i. 确定对象在使用前被初始化，内置类型（手动初始化）；自定义class类（构造函数初始化）

j. 跨越编译单元初始化次序：如果代码涉及两个以上源码文件，且文件依赖另外一个文件的non-local static对象。->使用单例模式解决问题。

2. CPU密集型/IO密集型

a. CPU密集型：系统运行CPU读写IO可以在很短的时间内完成，几乎没有阻塞，而CPU一直有大量运算要处理，表现是CPU负载长期过高。这时如果线程过多会徒增线程切换的消耗，没卵用。通常在训练算法模型、搞训练集的时候出现。线程数设置为CPU核数。

b. IO密集型：CPU使用率较低，CPU一直在等待IO读写操作，导致线程空余时间较多。通常会开CPU核心数倍数的线程，以提高CPU利用率。

3. 杂项

a. 加锁成功不涉及用户态到内核态的切换，加锁失败则会有用户态到内核态的切换，将线程设置为睡眠。解锁时会用用户态到内核态的切换，唤醒睡眠的线程。被唤醒线程会从内核态到用户态，执行获取mutex之后的逻辑。

b. sleep_for yield都会涉及用户态到内核态的切换。yield在需要调整线程的执行顺序的时候才会使用。

c. asm volatile保证汇编代码不会被编译器重排。

d. volatile保证编译器优化对变量的访问，每次访问从内存中获取而不是寄存器；防止编译器对代码进行重排，控制单线程内代码的访问顺序。

```
volatile int v;

void func() {
  v = 1; // A
  int x = v; // B
} // 单线程内B可能会排到A的前面，应为编译器认为v的值是1不变。
```

e.

量化

1. MPI

a. 基本概念

i. MPI（Message Passing Interface）是一种用于进程间通信的标准接口。

K8S

1. 基本概念：

a. pod：所有容器都在Pod中运行，一个Pod可以承载一个或者多个容器；一个Pod可以包含零个或者多个磁盘组；一个pod默认的容器交pause；一个pod中所有docker共享同一个网络栈，这就意味着虽然服务部署在不同docker中但不能配置不同端口。

b. StatefulSet:

i. 稳定的持久化存储

ii. 稳定的网络标志

iii. 有序部署，有序扩展，pod是有序的。

iv. 有序收缩，有序删除。

c. 服务发现：

2. 搭建

a. 使用minikube创建集群

参考文献

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

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

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

e.