C++学习总结

1. 继承

a. 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。

```cpp
#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;
```

```
18    }
19    virtual void display() const override {
20      std::cout << "a:" << a_ << " b:" << b_ << std::endl;
21    }
22    int b_;
23  };
24
25  class C {
26   public:
27    C(int c) : c_(c) { std::cout << "Init C" << std::endl; }
28
29    void self_display() const { std::cout << "c:" << c_ << std::endl; }
30    virtual void display() const { std::cout << "c:" << c_ << std::endl; }
31    int c_;
32  };
33
34  class D : public B, public C {
35   public:
36    D(int a, int b, int c, int d) : B(a, b), C(c), d_(d) {
37      std::cout << "Init D" << std::endl;
38    }
39
40    void self_display() const {
41      std::cout << "a:" << a_ << " b:" << b_ << " c:" << c_ << " d:" << d_
42                << std::endl;
43    }
44    virtual void display() const override {
45      std::cout << static_cast<const void*>(this) << std::endl;
46      std::cout << "a:" << a_ << " b:" << b_ << " c:" << c_ << " d:" << d_
47                << std::endl;
48    }
49    int d_;
50  };
51
52  int main() {
53    D* d = new D(1, 2, 3, 4);
54
55    A* a = d;
56    B* b = d;
57    C* c = d;
```

```
58
59    a->display();
60    b->display();
61    c->display();
62    d->display();
63
64    a->self_display();
65    b->self_display();
66    c->self_display();
67    d->self_display();
68
69    std::cout << static_cast<void*>(a) << std::endl;
70    std::cout << static_cast<void*>(b) << std::endl;
71    std::cout << static_cast<void*>(c) << std::endl;
72    std::cout << static_cast<void*>(d) << std::endl;
73
74    delete d;
75    return 0;
76  }
```

b. 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。

```
1  #include <iostream>
2
3  // 不包含虚继承的菱形继承
4  class A {
5   public:
6    virtual ~A() {}
7    int32_t a_;
8  };
9
10 class B : public A {
11  public:
12   virtual ~B() {}
13   int32_t b_;
14 };
15
```

```
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;
}

```

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

C. 多继承 && 成员变量构造顺序

```cpp
#include <iostream>

// 多继承父类的构造顺序，和成员变量的构造顺序是右定义的顺序决定的
// 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;
}
```

d. 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。（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;
```

```cpp
};

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;

    // 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;
}
```

2. 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
// 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;
};
```

```cpp

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;

  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
```

```cpp
// 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;
}
```

```
101
102 // 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
103 // 让模板进行自动类型推导
104 int main(int argc, char** argv) {
105   test01();
106   test02();
107   test03();
108   return 1;
109 }
```

3. 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。

```
1 a += 1;
2
3 lw r1, a
4 addi r1, r1, 1
5 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。

```
#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() {
```

```
15      int expected=1; // L15
16      // memory_order_relaxed is okay because this is an RMW,
17      // and RMWs (with any ordering) following a release form a release sequence
18      while (!flag.compare_exchange_strong(expected, 2, std::memory_order_acq_rel)) { // L18
19          expected = 1;
20      }
21  }
22
23  void thread_3() {
24      while (flag.load(std::memory_order_acquire) < 2); // L24
25      // if we read the value 2 from the atomic flag, we see 42 in the vector
26      assert(data.at(0) == 42); // L26
27  }
28
29  int main() {
30      std::thread a(thread_1);
31      std::thread b(thread_2);
32      std::thread c(thread_3);
33      a.join();
34      b.join();
35      c.join();
36
37      return 0;
38  }
```

4. STL-顺序容器

a. priority_queue

i. 原理：底层使用堆的思想，以vector为容器。

5. 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。

6. 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;
}

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>
#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;
```

```cpp

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;
}
```

7. 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

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

8. STL-Algorithms

a. for_each

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

#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

```
#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>
```

```cpp
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。

```cpp
#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

  std::uniform_int_distribution<int> dist(min, max);
  std::generate(start, end, [&]() { return dist(mte); });
}

template <class T>
void print(T t) {
  std::for_each(t.begin(), t.end(),
                [](const auto& i) { std::cout << i << " "; });
  std::cout << std::endl;
}

// 优先使用函数对象定义对比规则，此规则按升序排序
class MyComp {
 public:
  bool operator()(const int32_t& i, const int32_t& j) { return (i < j); }
};

int main() {
  // init
  std::vector<int32_t> a(10, 0);
  fill_with_random_int_values(a.begin(), a.end(), 0, 1000);
  print(a);

  // example 1
  // greater or less 说明的是大顶堆还是小顶堆，默认less小顶堆
  std::sort(a.begin(), a.end(), std::greater<int32_t>());
  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
50    return 0;
51  }
52
```

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

```
1  #include <algorithm>
2  #include <iostream>
3  #include <numeric>
4  #include <vector>
5
6  void print1(const int& val) { std::cout << val << '\t'; }
7
8  // 仿函数
9  class print2 {
10  public:
11   void operator()(const int& val) { std::cout << val << '\t'; }
12 };
13
14 // for_each
15 void test01() {
16   std::vector<int> a = {1, 2, 3, 4};
17
18   std::for_each(a.begin(), a.end(), print1);
```

```cpp
  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
```

```cpp
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;

  std::vector<int> d = {10, 2, 3};
  std::swap(a, d);
  std::for_each(a.begin(), a.end(), print2());
  std::cout << std::endl;
}

// accumulate && fill
void test08() {
  std::vector<int> a = {1, 2, 3, 4, 5, 6};
  int total = std::accumulate(a.begin(), a.end(), 0);
  std::cout << total << std::endl;

  std::vector<int> b(10, 0);
  std::fill(b.begin(), b.end(), 10);
  std::for_each(b.begin(), b.end(), print2());
  std::cout << std::endl;
}

// set function
void test09() {
  // 两个集合必须有序！！！！
  std::vector<int> a = {1, 3, 3, 5};
  std::vector<int> b = {1, 2, 3, 9};

  std::vector<int> c(std::min(a.size(), b.size()), 0);
  auto iter =
      std::set_intersection(a.begin(), a.end(), b.begin(), b.end(), c.begin());
```

```
  std::for_each(c.begin(), iter, print2());
  std::cout << std::endl;

  std::vector<int> d(a.size() + b.size(), 0);
  iter = std::set_union(a.begin(), a.end(), b.begin(), b.end(), d.begin());
  std::for_each(d.begin(), iter, print2());
  std::cout << std::endl;

  std::vector<int> e(a.size(), 0);
  iter = std::set_difference(a.begin(), a.end(), b.begin(), b.end(), e.begin());
  std::for_each(e.begin(), iter, print2());
  std::cout << std::endl;
}

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

9. 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. 进程间通信

i.

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。

e. 容器中存的对象必须是指定类的对象，不能通过父类指向子类的方式存储子类对象，因为容器中的类是真实按照父类去操作的，存子类会存在内存泄露。但容器中可以父类指针指向子类对象。

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

class Compoment {
 public:
  virtual ~Compoment(){};

  virtual void start() = 0;
  virtual void stop() = 0;
};

class Cpu : public Compoment {
 public:
  void start() override { std::cout << "Cpu start" << std::endl; }
  void stop() override { std::cout << "Cpu stop" << std::endl; }
};

class Gpu : public Compoment {
 public:
  void start() override { std::cout << "Gpu start" << std::endl; }
  void stop() override { std::cout << "Gpu stop" << std::endl; }
};

class Display : public Compoment {
 public:
  void start() override { std::cout << "Display start" << std::endl; }
  void stop() override { std::cout << "Display stop" << std::endl; }
};

```

```
class Computer {
 public:
  Computer() {
    compoments_.emplace_back(new Cpu());
    compoments_.emplace_back(new Gpu());
    compoments_.emplace_back(new Display());
  }

  void start() {
    for (auto& compoment : compoments_) {
      compoment->start();
    }
  }

  void stop() {
    for (auto iter = compoments_.rbegin(); iter != compoments_.rend(); ++iter) {
      (*iter)->stop();
    }
  }

 private:
  std::list<std::unique_ptr<Compoment>> compoments_;
};

int main() {
  Computer computer;
  computer.start();
  computer.stop();
  return 0;
}
```

2. 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>
```

```cpp
#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();
```

```
42
43    computer = std::make_unique<ComputerB>();
44    computer->product();
45  }
46
```

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。

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。

```
1  #include <iostream>
2  #include <memory>
3  #include <string>
4
5  class Girl {
6   public:
7    Girl(const std::string& name = "Girl") : name_(name) {}
8    std::string GetName() { return name_; }
9
10  private:
11   std::string name_;
12 };
13
14 // 需要将代理和真实对象之间有一个公共类
15 // 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
16 class Profession {
17  public:
18   virtual ~Profession() {}
19   virtual void profess() = 0;
20 };
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;
```

```
  }

 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;
}

```

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。

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

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);
    }
  }
```

```
51
52    void notify() override {
53      for (auto& p_view : view_list_) {
54        p_view->update();
55      }
56    }
57
58   private:
59    // 存储观察者
60    std::list<std::shared_ptr<View>> view_list_;
61  };
62
63  int main() {
64    auto v1 = std::make_shared<TableView>("TableView1");
65    auto v2 = std::make_shared<TableView>("TableView2");
66    auto v3 = std::make_shared<TableView>("TableView3");
67    auto v4 = std::make_shared<TableView>("TableView4");
68
69    IntDataModel int_data_model;
70    int_data_model.addView(v1);
71    int_data_model.addView(v2);
72    int_data_model.addView(v3);
73    int_data_model.addView(v4);
74
75    int_data_model.notify();
76
77    int_data_model.removeView(v1);
78    int_data_model.removeView(v2);
79    int_data_model.removeView(v3);
80    int_data_model.removeView(v4);
81
82    return 0;
83  }
84
```

5. 单例模式

a. 单例模式-懒汉模式

```cpp
#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() {
```

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

```
29
30 class Strategy01 : public Strategy {
31  public:
32   std::string doAlgorithm(const std::string& data) const override {
33     return data;
34   }
35 };
36
37 class Strategy02 : public Strategy {
38  public:
39   std::string doAlgorithm(const std::string& data) const override {
40     std::string ret(data);
41     std::sort(ret.begin(), ret.end());
42     return ret;
43   }
44 };
45
46 int main() {
47   Context context(std::make_unique<Strategy01>());
48   context.doBusiness();
49
50   context.setStrategy(std::make_unique<Strategy02>());
51   context.doBusiness();
52
53   return 0;
54 }
55
```

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。

```
1 #include <iostream>
2
3 // 被适配的类
4 class Deque {
5  public:
6   void pushFront() { std::cout << "push front" << std::endl; }
7   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;
}

```

8. 组合模式

a. 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。

b. 例如， 你有两类对象： 产品和 盒子 。 一个盒子中可以包含多个 产品或者几个较小的 盒子 。 这些小 盒子 中同样可以包含一些 产品或更小的 盒子 ， 以此类推。

c.

9. 桥接模式

a. 可将一个大类或一系列紧密相关的类拆分为抽象和实现两个独立的层次结构， 从而能在开发时分别使用。

b. 假如你有一个几何 形状Shape类， 从它能扩展出两个子类： 圆形Circle和 方形Square 。 你希望对这样的类层次结构进行扩展以使其包含颜色， 所以你打算创建名为 红色Red和 蓝色Blue的形状子类。 但是， 由于你已有两个子类， 所以总共需要创建四个类才能覆盖所有组合， 例如 蓝色圆形BlueCircle和 红色方形RedSquare 。在层次结构中新增形状和颜色将导致代码复杂程度指数增长。 例如添加三角形状， 你需要新增两个子类， 也就是每种颜色一个； 此后新增一种新颜色需要新增三个子类， 即每种形状一个。 如此以往， 情况会越来越糟糕。

c. 本质就是多个不同维度的类组合，当有子类扩展的时候，采用傻瓜模式进行组合，代码冗余高。

d. 拆分或重组一个具有多重功能的庞杂类 （例如能与多个数据库服务器进行交互的类）， 可以使用桥接模式。

e. 在几个独立维度上扩展一个类， 可使用该模式。

f. 需要在运行时切换不同实现方法， 可使用桥接模式。

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

```cpp
#include <string>

// 具体实现类的抽象父类，要实现具体实现类
// 每个实现类要实现自己的父类规定的纯虚函数
class Implementation {
 public:
  // 素有虚析构函数都要有实现
  virtual ~Implementation() {}
  virtual std::string OperationImplementation() const = 0;
};

class ConcreteImplementationA : public Implementation {
 public:
  std::string OperationImplementation() const override {
    return "ConcreteImplementationA operate";
  }
};

class ConcreteImplementationB : public Implementation {
 public:
  std::string OperationImplementation() const override {
    return "ConcreteImplementationB operate";
  }
};

// 桥接模式中的抽象类，也可以理解为表现类，或者不同维度的类
// 由于抽象类中没有定义纯虚函数，则可以用抽象类定义变量
// 也可以使用继承的方式实现自己的抽象类
class Abstraction {
 public:
  Abstraction(std::unique_ptr<Implementation>&& implementation)
      : implementation_(std::move(implementation)) {}
  virtual ~Abstraction(){};

  virtual std::string Operation() const {
    return "Abstraction Operation call: " +
           implementation_->OperationImplementation();
  }

  // 父类中可以被子类直接用的成员变量要声明成protected
```

```
 protected:
  std::unique_ptr<Implementation> implementation_;
};

class ExtendedAbstraction : public Abstraction {
 public:
  ExtendedAbstraction(std::unique_ptr<Implementation>&& implementation)
      : Abstraction(std::move(implementation)) {}

  std::string Operation() const override {
    return "ExtendedAbstraction Operation call: " +
           implementation_->OperationImplementation();
  }
};

void Client(const Abstraction& abstraction) {
  std::cout << abstraction.Operation() << std::endl;
}

int main() {
  // 客户端负责将抽象类和具体类相连接
  // 创建Abstraction并绑定具体Implementation
  std::unique_ptr<Implementation> imp_a =
      std::make_unique<ConcreteImplementationA>();
  Abstraction abs1(std::move(imp_a));
  Client(abs1);

  // 客户端负责将抽象类和具体类相连接
  // 创建ExtendedAbstraction并绑定具体Implementation
  std::unique_ptr<Implementation> imp_b =
      std::make_unique<ConcreteImplementationB>();
  ExtendedAbstraction abs2(std::move(imp_b));
  Client(abs2);

  return 0;
}
```

10. 生成器模式

a. 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。

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

// 具体的产品，产品里可以包含多个part，模拟房子中的
// 元素，例如窗户，门窗，，，，
class Product {
 public:
  void list() const {
    for (const auto& part : parts_) {
      std::cout << part << " ##";
    }
    std::cout << std::endl;
  }

  void insert(const std::string& part) { parts_.emplace_back(part); }

 private:
  std::list<std::string> parts_;
};

// 抽象Builder类，定义所有Builer的通用方法
class Builder {
 public:
  Builder() : product_(new Product()) {}
  virtual ~Builder(){};

  virtual void productPartA() const = 0;
  virtual void productPartB() const = 0;
  virtual void productPartC() const = 0;

  virtual std::unique_ptr<Product> getProduct() {
    // move之后一定要用左值去接，要不起不到移动资源的目的
    auto temp = std::move(product_);
    product_.reset(new Product());
    return std::move(temp);
  };
```

```
38
39  protected:
40   std::unique_ptr<Product> product_;
41 };
42
43 // 定义具体Builder类去生产
44 class Builder1 : public Builder {
45  public:
46   void productPartA() const override { product_->insert("PartA_From1"); }
47   void productPartB() const override { product_->insert("PartB_From1"); }
48   void productPartC() const override { product_->insert("PartC_From1"); }
49 };
50
51 class Builder2 : public Builder {
52  public:
53   void productPartA() const override { product_->insert("PartA_From2"); }
54   void productPartB() const override { product_->insert("PartB_From2"); }
55   void productPartC() const override { product_->insert("PartC_From2"); }
56 };
57
58 // 定义经理类，控制Builder的生产顺序
59 // 仅仅是指挥生产循序
60 class Director {
61  public:
62   Director() : builder_(nullptr) {}
63
64   void setBuilder(const std::shared_ptr<Builder>& builder) {
65     builder_ = builder;
66   }
67
68   void BuildMiniProduct() { builder_->productPartA(); }
69
70   void BuildFullProduct() {
71     builder_->productPartA();
72     builder_->productPartB();
73     builder_->productPartC();
74   }
75
76  private:
77   std::shared_ptr<Builder> builder_;
```

```
};

// 客户端的标准操作参考
void ClientCode() {
  Director director;
  std::shared_ptr<Builder> builder1 = std::make_shared<Builder1>();
  director.setBuilder(builder1);
  director.BuildMiniProduct();
  auto product = builder1->getProduct();
  product->list();

  director.BuildFullProduct();
  product = builder1->getProduct();
  product->list();

  std::shared_ptr<Builder> builder2 = std::make_shared<Builder2>();
  builder2->productPartA();
  builder2->productPartB();
  builder2->productPartC();
  product = builder2->getProduct();
  product->list();
}

int main() {
  ClientCode();
  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

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

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

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

h.