

C++ 编程指南

作者:向仔州

目录

const 常量修饰 C 语言和 C++语言的区别.....	5
bool 运算, C 语言没有, C++有.....	6
三目运算符, C 语言和 C++的区别.....	7
引用, 就类似 C 语言指.....	7
const 引用.....	8
结构体里面定义引用.....	9
子函数返回引用.....	10
函数返回值变量是引用, 但是 return 返回的是指针, 其实返回的就是地址和值	11
指针引用的使用 &*	12
C++定义函数增加了函数参数默认值定义功能	14
C++和 JAVA 语言一样, 有函数重载	15
函数重载与函数指针	17
C++调用 C 语言	18
C++动态分配内存用 new, 也就是 C 语言的 malloc	21
C++命名空间, 就是将 C 语言全局变量存放的区域进行划分	24
C++ struct 结构体的升级	27
C++用 class 来代替 struct	28
C++创建对象时, 对象在内存的分布	30
C++构造函数, 让你创建对象时自动初始化对象	32
C++用数组方式初始化构造函数.....	33
两个对象相互赋值	38
用列表来解决类无法初始化类里面成员变量的问题	41
const 在类中的初始化也要用到列表	45
析构函数帮你自动释放 new 创建的对象占用的内存	47
用 const 修饰类的对象	50

计算一个程序里面对象创建的次数，但是不允许用全局变量来计算	52
因为不允许用全局变量来计算对象创建次数，所以我们要用静态成员变量	53
static 修饰静态成员函数	56
二阶构造模式(设计模式的一种)，记住并不是什么新语法新关键字。	59
二阶构造函数实际应用	61
友元这个东西很扯，知道就行了	63
类里面函数重载	66
this 指针的用法	67
C++除了函数可以重载，+ - * /操作符也可以重载	69
STL 模板库与 << 操作符重载	75
STL 模板库使用	77
字符串转换成数字	78
数字转换成字符串	79
string 类定义的字符串长度计算	80
String 获取一整串字符串里面的某一段字符串	80
字符串将右移的字符放在前面	80
C++的 string 对象支持单个字符访问，单个字符访问方式就是 C 语言数组下标	81
C++的 string 对象，判断对象里面有几个是数字	81
数组符号[] 用作操作符重载	82
C 语言和 C++ 在函数内部使用 static 的区别	86
函数括号 () 作为操作符重载	87
智能指针就是帮助你自动释放内存的	88
类的继承关系，和面向对象	89
类里面 Public, Private, Protected 三个权限的区别	93
C++子类继承父类有三种权限方式，public, private, protected	95
C++子类继承父类，构造函数的调用顺序和析构函数的调用顺序	98
子类中定义了和父类一样的同名变量，同名函数，是否会和父类冲突	102
C++的多态使用 virtual 关键字	107
Struct 和 class 内存分配问题	111
C++抽象类使用	112
不怎么用的多重继承	114
C++变量强制类型转换和 C 语言不一样	119

static_cast 用于基本数据类型的转换，比如 char, int, long, double。不能用于指针类型转换，比如*char, *int, *long, *double。	119
指针和指针类型进行强制转换用 const_cast	120
dynamic_cast，就是动态类型转换。如子类和父类之间的类型转换。	122
reinterpret_cast	125
函数模板解决函数重载这种体力活问题	126
类模板使用	132
多参数模板类实现	136
模板类实现运算符重载	136
模板类继承	137
仿函数(函数对象)使用	138
用类模板做一个栈	140
容器实现	143
Vector 向量使用	143
迭代器	144
类模板方式的智能指针	145
下面我们用 QT 的智能指针来操作试试	147
C++11: override 和 final 关键字使用	149
for 循环的四种用法	150
C++11: 字符串原始字面量	151
C++11: auto 关键字，自动推导变量类型	152
C++11 auto 适用的场景	153
map 使用，STL 库 map	153
C++11: decltype 关关键字，自动推导变量类型，和 auto 有点类型	154
C++11: Lambda 表达式	155
C++, try catch 和 throw 使用	157
C++设计模式与语法实战	159
UML 类图介绍	159
UML 类图继承关系和依赖关系	160
UML 类图关联关系	161
UML 类图聚合和组合关系	162
模板方法，设计模式的基础	163

责任链模式	165
为什么需要工厂方法	170
C++11 版本 nullptr 空指针使用	171
nullptr 的应用场景	171
简单工厂方法使用	172
工厂模式	175
单例模式	177
桥接模式	179
桥接模式实际案例 1	182
桥接模式实际案例 2	183
桥接模式实际案例 3	186
适配器模式	188
代理模式	190
代理模式实际案例 1， 客户端通过代理对象间接访问远程服务器或者远程对象	191
代理模式实际案例 2， 虚拟代理	192
代理模式实际案例 3， 保护代理	192
观察者模式	193
观察者模式实际案例 1， 信号与槽基本设计思路	197
QT 中信号与槽， 就是用观察者实现的， 下面用模板来实现信号与槽函数	198
发布订阅模式	201
命令模式	205
中介者模式	208

const 常量修饰 C 语言和 C++ 语言的区别

```
1 #include<stdio.h>
2
3 int main()
4 {
5     const int c = 0;
6     int *p = (int *)&c;
7
8     printf("c = %d\n", c);
9     *p = 5;
10    printf("c = %d\n", c);
11
12    return 0;
13 }
```

在 gcc 编译器下，const 定义的常量可以取地址去修改它

```
root@ubuntu:/home/xiang/C++# gcc -o const const.c
root@ubuntu:/home/xiang/C++# ./const
c = 0
c = 5
root@ubuntu:/home/xiang/C++#
```

```
1 #include<stdio.h>
2
3 int main()
4 {
5     const int c = 0;
6     int *p = (int *)&c;
7
8     printf("c = %d\n", c);
9     *p = 5;
10    printf("c = %d\n", c);
11
12    return 0;
13 }
```

在 g++ 编译器下，const 定义的常量就是常量，谁都不能修改

```
g++: error: constcpp: No such file or directory
root@ubuntu:/home/xiang/C++# g++ -o constcpp constcpp.cpp
root@ubuntu:/home/xiang/C++# ls
const.cpp
root@ubuntu:/home/xiang/C++# ./constcpp
c = 0
c = 0
root@ubuntu:/home/xiang/C++#
```

这就是 C++ 对常量进行了优化，C++ 语言就是把 C 语言里面的一些语法 bug 进行了优化，让语法不再出现 bug，所以 C++ 就是 C 语言的升级版

```
1 #include<stdio.h>
2
3 int main()
4 {
5     const int A = 1;
6     const int B = 2;
7
8     char array[A+B] = {0};
9
10    return 0;
11 }
```

在 C 语言里面常量是处于不明确地带的，所以数组使用编译报错

```
root@ubuntu:/home/xiang/C++# gcc -o const const.c
const.c: In function `main':
const.c:8:2: error: variable-sized object may not be initialized
char array[A+B] = {0};
^
const.c:8:21: warning: excess elements in array initializer
char array[A+B] = {0};
^
const.c:8:21: note: (near initialization for 'array')
```

```
2 #include<stdio.h>
3
4 int main()
5 {
6     const int A = 1;
7     const int B = 2;
8
9     char array[A+B] = {0};
10
11    return 0;
12 }
```

在 C++ 语言里面常量是真正意义上的常量
目标很明确，所以编译通过

```
root@ubuntu:/home/xiang/C++# g++ -o constcpp constcpp.cpp
root@ubuntu:/home/xiang/C++# ./constcpp
```

bool 运算, C 语言没有, C++有

因为 C 语言用来表示 bool 运算都是用整形，比如 int a=1 为 true, a=0 为假

但是 C 语言这样做很不严谨，所以 C++ 增加了 bool 的功能。

```
#include<stdio.h>
int main()
{
    bool b = 0;

    printf("b = %d\n", b);
    b++;
    printf("b = %d\n", b);
    b = b-3;
    printf("b = %d\n", b);
    return 0;
}
```

但是 C++ 的 bool 功能有运算功能，而不是像 JAVA 那样就只有 true 或者 false

但是 C++ 的 bool 值是大于 1 返回 1

这里是 -1，但是也返回 1

```
root@ubuntu:/home/xiang/C++# ./constcpp
b = 0
b = 1
b = 1
```

C++ 的 bool 只要得到的不是 0，不管是负数还是正数，只要不是 0 都返回 1

```
#include<stdio.h>
int main()
{
    bool b = false; // false 就是表示 0，在 C++ 中已经把 false 集成在里面了
    int a = b;
    printf("a = %d      b = %d\n", a, b); // a 可以接受 b 的值

    b = 3; // 虽然这里是 3，按照 bool 标准，会把这个 3 转换成 1 给 a
    a = b;
    printf("a = %d      b = %d\n", a, b);

    b = -5; // 虽然这里是 -5，按照 bool 标准，会把这个 -5 转换成 1 给 a
    a = b;
    printf("a = %d      b = %d\n", a, b);

    a = 10; // 整形 a 虽然等于 10，但是赋值给 bool 变量 b 的时候，b 会自动转换成 1
    b = a;
    printf("a = %d      b = %d\n", a, b);

    a = 0;
    b = a;
    printf("a = %d      b = %d\n", a, b);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/bool# g++ -o bool bool.cpp
root@ubuntu:/home/xiang/C++/bool# ./bool
a = 0      b = 0
a = 1      b = 1
a = 1      b = 1
a = 10     b = 1
a = 0      b = 0
root@ubuntu:/home/xiang/C++/bool#
```

这就是 bool 的运用方法

三目运算符，C 语言和 C++ 的区别

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int a = 1;
6     int b = 2;
7     int c = (a < b?a:b);
8     printf("c = %d\n",c);
9 }
```

这是 C 语言，如果 $a < b$ 就返回 a 的值给 c

```
root@ubuntu:/home/xiang/C++/sanmu# ./sanmu
c = 1
```

输出结果没有问题

```
root@ubuntu:/home/xiang/C++/sanmu# g++ -o sanmucpp sanmucpp.cpp
root@ubuntu:/home/xiang/C++/sanmu# ./sanmucpp
c = 1
```

在 C++ 语言里面也没有问题

```
#include<stdio.h>
int main()
{
    int a = 1;
    int b = 2;
    (a < b?a:b) = 10;
    printf("c = %d\n",a);
}
```

在 C 语言中三目运算符作为左值是不行的，因为编译器认为三目运算符的结果是个数值，而不是变量

```
root@ubuntu:/home/xiang/C++/sanmu# gcc -o sanmu sanmu.c
sanmu.c: In function 'main':
sanmu.c:7:14: error: lvalue required as left operand of assignment
(a < b?a:b) = 10;
^
root@ubuntu:/home/xiang/C++/sanmu#
```

```
#include<stdio.h>
int main()
{
    int a = 1;
    int b = 2;
    (a < b?a:b) = 10;
    printf("a = %d\n",a);
}
```

在 C++ 中三目运算符可以当左值，C++ 中先计算三目运算符 a 和 b ，发现 $a < b$ 成立返回 a 变量。然后自动将 10 赋值给返回的 a 变量

```
root@ubuntu:/home/xiang/C++/sanmu# g++ -o sanmucpp sanmucpp.cpp
root@ubuntu:/home/xiang/C++/sanmu# ./sanmucpp
a = 10
root@ubuntu:/home/xiang/C++/sanmu#
```

所以 C++ 对三目运算符做了升级，C++ 三目运算符返回的是变量本身 C 语言三目运算符返回的是变量值

引用，就类似 C 语言指针

```
#include<stdio.h>
int main()
{
    int a = 10;
    int &b = a;

    printf("a = %d\n",a);
    printf("b = %d\n",b);
    printf("a addr = %p\n",&a);
    printf("b addr = %p\n",&b);
}
```

在定义变量的前面加&符号就是引用，引用要求定义的变量类型两边必须一样，在定义的时候就必须让引用变量 b 得到 a 变量的地址($\text{int } \&b=a$)，然后操作 b 就等于去操作 a

C 语言指针变量要占用一个空间，但是 C++ 引用变量就不占用空间，所以引用是对指针的强化

```
root@ubuntu:/home/xiang/C++/yiyong# ./yy
a = 10
b = 10
a addr = 0x7fff60b1a22c
b addr = 0x7fff60b1a22c
```

那么引用这个东西到底有什么用，感觉看起没什么用啊！！，引用主要用在函数形参上

```
1 #include<stdio.h>
2
3
4 void func1(int *a , int *b)
5 {
6     *a = *a + *b;
7 }
8
9
10 int main()
11 {
12     int a = 10;
13     int b = 20;
14
15     func1(&a,&b);
16     printf("a = %d\n",a);
17 }
```

C 语言就是要用*号指针做函数形参

而且在传入 a, b 的时候还要
加&符号，感觉看起不直观

```
root@ubuntu:/home/xiang/C++/yiyong# gcc -o yy yyc.c
root@ubuntu:/home/xiang/C++/yiyong# ./yy
a = 30
```

```
#include<stdio.h>

void func1(int &a , int &b)
{
    a = a + b;
}

int main()
{
    int a = 10;
    int b = 20;
    func1(a,b);
    printf("a = %d\n",a);
}
```

C++在形参里面用，引用方法，取传入
进来变量的地址

C++将变量传入函数，但是不需要像上面 C 语言
那样用&符号，所以看起来直观

```
root@ubuntu:/home/xiang/C++/yiyong# g++ -o yy yy.cpp
root@ubuntu:/home/xiang/C++/yiyong# ./yy
a = 30
```

所以 C++用引用来代替 C 语言指针做函数形参，只是一种改进。

const 引用

```
int main()
{
    int a = 10;
    const int &b = a;
    int c;
    c = b;
    printf("c = %d\n",c);
}
```

C++的 const 修饰引用变量，这个引
用变量就变成了只读属性。

我将 b 赋值给 c 是没有问题的

```
root@ubuntu:/home/xiang/C++/yiyong# ./yy
c = 10
```

```

0 int main()
1 {
2     int a = 10;
3     const int &b = a;
4     b = 50; // 这里修改了常量的值
5     printf("b = %d\n", b);
6 }

```

但是我去修改 const 定义了的引用的值就会报错，因为该引用变量只有可读属性

```

root@ubuntu:/home/xiang/C++/yiyong# g++ -o yy yy.cpp
yy.cpp: In function 'int main()':
yy.cpp:14:4: error: assignment of read-only reference 'b'
      b = 50;
      ^

```

结构体里面定义引用

```

#include<stdio.h>

struct Tref
{
    char &r;
};

int main()
{
    char c = 'c';
    char &rc = c;
    Tref ref = { c };

    printf("rc 占用内存大小: %d\n", sizeof(rc)); // 这里计算的就是c变量占用的内存空间大小
    printf("Tref 占用内存大小: %d\n", sizeof(Tref)); // 在ubuntu系统里面c语言指针占用8个字节，引用是C++里面替代指针的角色，所以也是8个字节
    printf("ref.r 占用内存大小: %d\n", sizeof(ref.r)); // 这里结构体里面的r指向了c，所以这里计算的是c变量的大小
    return 0;
}

```

```

root@ubuntu:/home/xiang/C++/yiyong# ./yy
rc 占用内存大小: 1
Tref 占用内存大小: 8
ref.r 占用内存大小: 1

```

所以 C++ 里面的引用本质就是指针，只是为了程序员看起代码直观用引用代替了指针。

```

1 #include<stdio.h>
2
3 struct Tref
4 {
5     char *before;
6     char &ref;
7     char *after;
8 };
9
10 int main()
11 {
12     char a = 'a';
13     char b = 'b';
14     char c = 'c';
15
16     Tref r = {&a,b,&c};
17     printf("r 占用内存大小: %d\n", sizeof(r));
18     printf("r.before 占用内存大小: %d\n", sizeof(r.before));
19     printf("r.ref 占用内存大小: %d\n", sizeof(r.ref));
20     printf("r.after 占用内存大小: %d\n", sizeof(r.after));
21     printf("r.before 地址: %p\n", &r.before);
22     printf("r.ref: %p\n", &r.ref);
23     printf("r.after: %p\n", &r.after);
24
25 }

```

```

root@ubuntu:/home/xiang/C++/yiyong# ./yy
r 占用内存大小: 24
r.before 占用内存大小: 8
r.ref 占用内存大小: 1
r.after 占用内存大小: 8
r.before 地址: 0x7ffdb95cfbd0
r.ref: 0x7ffdb95cfbce
r.after: 0x7ffdb95cfbe0

```

子函数返回引用

```
5 int& demo()
6 {
7     int d = 50;
8     return d;
9 }
10 int& func()
11 {
12     static int s = 10;
13     return s;
14 }
15
16
17 int main()
18 {
19     int& rd = demo();
20     int& rs = func();
21
22     printf("rd = %d\n", rd);
23     printf("rs = %d\n", rs);
24     return 0;
25 }
```

子函数定义的变量在返回时子函数已经释放了内存，所以变量地址也不在了

```
root@ubuntu:/home/xiang/C++/yiyong# g++ -o yy yy.cpp
yy.cpp: In function ‘int& demo()’:
yy.cpp:6:6: warning: reference to local variable ‘d’ returned [-Wreturn-local-addr]
  int d = 50;
           ^
root@ubuntu:/home/xiang/C++/yiyong# ./yy
Segmentation fault (core dumped)
```

编译警告，运行段错误

```
4 int& demo()
5 {
6 }
7
8 int& func()
9 {
10     static int s = 10;
11     return s;
12 }
13
14
15 int main()
16 {
17     int& rd = demo();
18     int& rs = func();
19
20     printf("rd = %d\n", rd);
21     printf("rs = %d\n", rs);
22     return 0;
23 }
```

取消掉子函数变量返回引用

用 static 将子函数变量写死在内存里面，为什么用 static，我的 C_advanc 文档讲了的

这样返回变量地址的时候，子函数执行完了，变量地址都还在

定义一个新的引用在函数外部接收子函数返回的变量地址

这样就没有问题

```
root@ubuntu:/home/xiang/C++/yiyong# g++ -o yy yy.cpp
root@ubuntu:/home/xiang/C++/yiyong# ./yy
rd = -443987883
rs = 10
```

函数返回值变量是引用，但是 return 返回的是指针，其实返回的就是地址和值

```
#include<iostream>
#include<stdio.h>
using namespace std; // 如果这里写返回引用

int &func(int *p)
{
    *p = 10;
    return *p; // 那么必须返回整个变量的
} // 地址和地址上面的值

int main()
{
    int a = 0;
    int b = func(&a);
    cout << "b = " << b << endl;
    printf("a addr = %p\n", &a);
    printf("b addr = %p\n", &b);
    return 0;
}
```

root@ubuntu:/home/xzz/code/
b = 10
a addr = 0x7ffd74b6be0
b addr = 0x7ffd74b6be4
root@ubuntu:/home/xzz/code/

函数引用返回值，返回对象是什么样的？

```
#include<iostream>
#include<stdio.h>
using namespace std;

class test
{
    int a;
public:
    int get(void)
    {
        a = 10;
        return a;
    }
};

test &retest(test *p) // 函数引用返回值，如果是返回对象指针，那么就要将对象的地址和值都返回给引用
{
    return *p;
}

int main()
{
    test t1;
    cout << "t1.get = " << retest(&t1).get() << endl;
    printf("t1 addr = %p\n", &t1);
    printf("retest addr = %p\n", &retest(&t1));
    return 0;
}
```

root@ubuntu:/home/xzz/code/C++/

t1.get = 10
t1 addr = 0x7fff1d82d000
retest addr = 0x7fff1d82d000

引用返回的对象和传入的 t1 对象地址相同。

所以你发现返回的是整个对象，还可以操作对象里面的函数

指针引用的使用 &*

```
static int global = 100;

void func(int *p)
{
    p = &global;
}

int main()
{
    int num = 20;
    int *p1 = &num;
    cout << "p1 = " << *p1 << endl;

    func(p1); //将p1指向的num的地址传递进func函数
    cout << "p1 = " << *p1 << endl;
    return 0;
}
```

P 指向 num 的地址
打印出 num 的数值
没有错

内存模型

num	20
地址	数值

```
root@ubun
p1 = 20
p1 = 20
```

输出还是 20

子函数形参指针 func() 的 *p，在子函数里面是无法修改传入子函数 p1 指向的内存地址 num，只能用 *p 去修改 num 上面的值，但是不能换掉 num 的地址

```
static int global = 100;

void func(int *p)
{
    *p = 200;
}

int main()
{
    int num = 20;
    int *p1 = &num;
    cout << "p1 = " << *p1 << endl;

    func(p1);
    cout << "p1 = " << *p1 << endl;
    return 0;
}
```

所以在子函数里面只能修改 p1 指针指向变量 num 的值，不能修改 p1 指针指向的地址

```
root@ubunt
p1 = 20
p1 = 200
```

如果用** 指针的指针就能解决在子函数里修改外部传进来指针指向的地址

```

5 static int global = 100;
6
7 void func(int **p)
8 {
9     *p = &global;
10 }
11
12 int main()
13 {
14     int num = 20;
15     int *p1 = &num;
16     cout << "p1 = " << *p1 << endl;
17
18     func(&p1);
19     cout << "p1 = " << *p1 << endl;
20     return 0;
21 }

```

用二级指针的方法来接收传入进来的指针本身地址

我们传入的是 p1 指针本身的地址，但是子函数形参*p 的 p 地址还是和主函数 p1 不一样，但是子函数 func() 的 *p 是可以操作 p1 指向的内存空间的 num 的地址，子函数**p 可以操作 p1 指向内存空间 num 上的值

```

root@ubuntu:~/code/C++/yin>
p1 = 20
p1 = 100
root@ubuntu:~/code/C++/yin>

```

所以在子函数 func 里面只有这样才可以修改外部指针 p1 指向的地址值。

下面我们用指针引用实现 func(int **p)，子函数修改外部指针指向的地址，

```

static int global = 100;
void func(int *&p)
{
    p = &global;
}

int main()
{
    int num = 20;
    int *p1 = &num;
    cout << "p1 = " << *p1 << endl;

    func(p1);
    cout << "p1 = " << *p1 << endl;
    return 0;
}

```

指针引用和**p 是一样的，可以修改外部指针指向的地址

```

root@ubuntu:~/code/C++/yin>
p1 = 20
p1 = 100
root@ubuntu:~/code/C++/yin>

```

指针引用还有个重要的好处

```

8 void func(int *&p)
9 {
10     p = new int;
11     *p = 50;
12     printf("func p = %p\n", p);
13 }
14
15 int main()
16 {
17     int num = 20;
18     int *p1 = &num;
19     cout << "p1 = " << *p1 << endl;
20     printf("p1 addr = %p\n", p1);
21
22     func(p1);
23     cout << "p1 = " << *p1 << endl;
24     printf("func before p1 addr = %p\n", p1);
25     return 0;
26 }

```

指针引用形参的函数可以在函数里面动态分配一个内存

然后外部指针传入这个 func 函数，就会得到这个 p 变量的地址，然后就可以在外部操作 p 变量了，这种 func 函数就很适合像 malloc 一样做内存分配函数

```

root@ubuntu:~/code/C++/yin>
p1 = 20
p1 addr = 0x7ffc5935019c
func p = 0x1efd030
p1 = 50
func before p1 addr = 0x1efd030
root@ubuntu:~/code/C++/yin>

```

C++定义函数增加了函数参数默认值定义功能

```
#include<stdio.h>

int mul(int x=0);

int main()
{
    int c = mul();
    printf("c = %d\n",c);
    return 0;
}

int mul(int x)
{
    return x*x;
}
```

C语言是不能在定义函数的时候传入参数的

```
root@ubuntu:/home/xiang/C++/func# gcc -o func func.c
func.c:3:14: error: expected ';' ',' or ')' before '=' token
int mul(int x=0);
                  ^
func.c: In function 'main':
func.c:7:10: warning: implicit declaration of function 'mul' [-Wimplicit-function-declaration]
int c = mul();
```

```
3 int mul(int x=0);
4
5 int main()
6 {
7     int c = mul();
8     printf("c = %d\n",c);
9     return 0;
10 }

11 int mul(int x)
12 {
13     return x*x;
14 }
```

C++对函数功能做了提升，可以在定义函数的时候传入参数

在调用函数时，如果不给函数形参传入参数，函数就用默认定义的形参数

```
root@ubuntu:/home/xiang/C++/func# g++ -o funcpp funcpp.cpp
root@ubuntu:/home/xiang/C++/func# ls
func.c  funcpp  funcpp.cpp
root@ubuntu:/home/xiang/C++/func# ./funcpp
c = 0
root@ubuntu:/home/xiang/C++/func#
```

```
2
3 int mul(int x=5);
4
5 int main()
6 {
7     int c = mul();
8     printf("c = %d\n",c);
9     return 0;
10 }

11 int mul(int x=5)
12 {
13     return x*x;
14 }
```

如果我在函数声明定义形参数值

在函数定义我也定义一样的形参数值

```
root@ubuntu:/home/xiang/C++/func# g++ -o funcpp funcpp.cpp
funcpp.cpp: In function 'int mul(int)':
funcpp.cpp:12:16: error: default argument given for parameter 1 of 'int mul(int)' [-fpermissive]
int mul(int x=5)
                  ^
funcpp.cpp:3:5: note: previous specification in 'int mul(int)' here
int mul(int x=5);
```

C++编译会报错，C++严格规定如果你要在函数参数定义数值，就只能在声明函数这里定义。

C++和JAVA语言一样，有函数重载

```
1 #include<stdio.h>
2
3 void func(int x)
4 {
5     printf("func (int x) = %d\n",x);
6 }
7
8 void func(int x,int y)
9 {
10    int c = x + y;
11    printf("func (int x , y) = %d\n",c);
12 }
13
14 void func(char *s)
15 {
16     printf("func (char *s) = %s\n",s);
17 }
18
19 int main()
20 {
21     func(10);
22     func(1,2);
23     func("string....");
24     return 0;
25 }
```

根据传入子函数形参的数量不同，调用不同形参的同名子函数

```
root@ubuntu:/home/xiang/C++/funcz# g++ -o funczcpp funczcpp.cpp
funczcpp.cpp: In function ‘int main()’:
funczcpp.cpp:23:19: warning: deprecated conversion from string constant to ‘char*’ [-Wwrite-strings]
  func("string....");
^
root@ubuntu:/home/xiang/C++/funcz# ls
funczcpp  funczcpp.cpp
root@ubuntu:/home/xiang/C++/funcz# ./funczcpp
func (int x) = 10
func (int x , y) = 3
func (char *s) = string....
root@ubuntu:/home/xiang/C++/funcz#
```

输出结果正常

```
void func(int x)
{
    printf("func (int x) = %d\n",x);
}

void func(long l)
{
    printf("func (int l) = %ld\n",l);
}

void func(char *s)
{
    printf("func (char *s) = %s\n",s);
}

int main()
{
    func(10);
    func(5000);
    func("string....");
    return 0;
}
```

传入函数形参的变量个数相同，变量类型不同，也可以重载

```
root@ubuntu:/home/xiang/C++/funcz# ./funczcpp
func (int x) = 10
func (int x) = 5000
func (char *s) = string....
```

```

void func(char *s,int x)
{
    printf("%s %d\n",s,x);
}

void func(int x,char *s)
{
    printf("%d %s\n",x,s);
}

int main()
{
    func("string1...",10);
    func(20,"string2..");
    return 0;
}

```

两个同名函数，形参变量数量相同，
形参变量类型相同，只是形参变量顺序不一样，这样的函数(可以重载)

```

root@ubuntu:/home/xiang/C++/funcz# ./funczcpp
string1... 10
20 string2..

```

```

void func(float f,int x)
{
    printf("float = %f int = %d\n",f,x);
}

void func(int x,float f)
{
    printf("int = %d float = %f\n",x,f);
}

int main()
{
    func(1.25,10);
    func(20,3.14);
    return 0;
}

```

```

root@ubuntu:/home/xiang/C++/funcz# g++ -o funczcpp funczcpp.cpp
root@ubuntu:/home/xiang/C++/funcz# ./funczcpp
float = 1.250000 int = 10
int = 20 float = 3.140000
root@ubuntu:/home/xiang/C++/funcz#

```

注意函数变量数量参数相同，顺序不同的函数重载只适用于字符串和整形，或者浮点型和整形。如果你是两个类型很接近的变量，函数是不能重载的。

```

4 void func(long l,int x)
5 {
6     printf("long = %ld int = %d\n",l,x);
7 }
8
9 void func(int x,long l)
10 {
11     printf("int = %d long = %ld\n",x,l);
12 }
13
14 int main()
15 {
16     func(5000,10);
17     func(20,5000);
18     return 0;
19 }

```

这种就不行，虽然函数参数数量相同，顺序不同，但是长整型和整形都是十进制，所以函数无法重载

```

root@ubuntu:/home/xiang/C++/funcz# g++ -o funczcpp funczcpp.cpp
funczcpp.cpp: In function 'int main()':
funczcpp.cpp:16:14: error: call of overloaded 'func(int, int)' is ambiguous
      func(5000,10);
                  ^
funczcpp.cpp:4:6: note: candidate: void func(long int, int)
void func(long l,int x)
                  ^
funczcpp.cpp:9:6: note: candidate: void func(int, long int)
void func(int x,long l)
                  ^
funczcpp.cpp:17:14: error: call of overloaded 'func(int, int)' is ambiguous
      func(20,5000);
                  ^

```

函数重载与函数指针

```
3 void func(int x)
4 {
5     printf("int = %d\n",x);
6 }
7
8
9 void func(int x,long l)
10 {
11     printf("int = %d long = %ld\n",x,l);
12 }
13
14 void func(char *s)
15 {
16     printf("%s\n",s);
17 }
18
19 int main()
20 {
21     void (*pFunc)(int a);
22     pFunc = func;
23     pFunc(1);
24
25     return 0;
26 }
```

C++和C语言一样
定义函数指针，虽然
子函数都是重名的，
但是根据定义的函
数指针可以决定我
只能指向什么类型的
函数

根据函数指针的定
义我们知道该指针
指向的func绝对是
func(int x)这个函
数地址

```
root@ubuntu:/home/xiang/C++/funcz# g++ -o funczcpp funczcpp.cpp
root@ubuntu:/home/xiang/C++/funcz# ./funczcpp
int = 1
```

```
4 void func(int x)
5 {
6     printf("int = %d\n",x);
7 }
8
9 void func(int x,long l)
10 {
11     printf("int = %d long = %ld\n",x,l);
12 }
13
14 void func(char *s)
15 {
16     printf("%s\n",s);
17 }
18
19 int main()
20 {
21     void (*pFunc)(int a);
22     pFunc = func;
23     pFunc(1,2); // 错误
24
25     return 0;
26 }
```

函数指针在定义的时候就
已经规定了只能指向一个
形参的函数，这里我写两个
参数给函数就是错误的

```
root@ubuntu:/home/xiang/C++/funcz# g++ -o funczcpp funczcpp.cpp
funczcpp.cpp: In function 'int main()':
funczcpp.cpp:23:11: error: too many arguments to function
    pFunc(1,2);
          ^
```

所以函数重载，每个函数名字是一样的，但是根据每个函数的参数数量类型不同，每个同名函数的地址是不一样的。

C++调用 C 语言

在目录下建立 C 文件和 h 文件

```
func.c
#include<stdio.h>
#include "func.h"

int main()
{
    void (*pFunc)(int a);
    pFunc = func;
    pFunc(1,2);

    return 0;
}
```

```
func.h
#ifndef __FUNC_H
#define __FUNC_H

int add(int x,int y);
int sub(int x,int y);

#endif
```

```
root@ubuntu:/home/xiang/C++/funcz# gcc -c func.c -o func.o
root@ubuntu:/home/xiang/C++/funcz# ls
func.c  func.h  func.o  funcz.cpp  funczcpp.cpp
root@ubuntu:/home/xiang/C++/funcz#
```

因为 C++文件调用 C 文件是靠 C 文件的 .o 文件链接的，所以要把 C 语言代码编译成.o 文件

```
funczcpp.cpp
#include<stdio.h>
#include "func.h"

int main()
{
    int c = add(10,20);
    int s = sub(50,40);
    printf("c = %d s = %d\n",c,s);

    return 0;
}
```

C++文件调用 C 文件里面的
函数

```
root@ubuntu:/home/xiang/C++/funcz# g++ -o funczcpp funczcpp.cpp func.o
/tmp/ccZr1Kb8.o: In function `main':
funczcpp.cpp:(.text+0x13): undefined reference to `add(int, int)'
funczcpp.cpp:(.text+0x25): undefined reference to `sub(int, int)'
collect2: error: ld returned 1 exit status
```

发现编译 c++ 文件出错，找不到 c 文件的函数

```
root@ubuntu:/home/xiang/C++/funcz# nm func.o
0000000000000000 T add
000000000000000014 T sub
root@ubuntu:/home/xiang/C++/funcz#
```

我们用 nm 命令查看 func.o 发现 add 和 sub 函数确实是编译进去了的。那么怎么 C++ 调用不到呢？

```
1 #include<stdio.h>
2
3 extern "C"
4 {
5 #include "func.h"
6 }
7
8 int main()
9 {
10
11     int c = add(10,20);
12     int s = sub(50,40);
13     printf("c = %d s = %d\n",c,s);
14
15     return 0;
16 }
```

```
root@ubuntu:/home/xiang/C++/funcz# g++ -o funczcpp funcz.cpp func.o
root@ubuntu:/home/xiang/C++/funcz# ./funczcpp
c = 30 s = 10
```

我们在 C++ 代码里面用
extern 来指定括号里面的东西是用 C 语言编写的
这里我指定 func.h 头文件里面的函数是用 c 语言编写的

编译通过，执行成功

如果我将这个 funcz.cpp 用 gcc 来编译可以吗

```
/C++/funcz
  funcz.c
/c++/funcz
```

现将 C++ 文件改成 c 文件

```
#include<stdio.h>

extern "C"
{
#include "func.h"
}

int main()
{
    int c = add(10,20);
    int s = sub(50,40);
    printf("c = %d s = %d\n",c,s);

    return 0;
}
```

C 语言编译器不支持这种
extern "C" 的写法

```
root@ubuntu:/home/xiang/C++/funcz# gcc -o funcz funcz.c func.o
funcz.c:3:8: error: expected identifier or '(' before string constant
  extern "C"
      ^
funcz.c: In function 'main':
```

所以编译报错

我们用 __cplusplus 宏来设置定义的代码是否被编译

```
1 #include<stdio.h>
2
3 #ifdef __cplusplus
4 extern "C"
5 {
6 #endif
7
8 #include "func.h"
9
10 #ifdef __cplusplus
11 }
12#endif
13
14 int main()
15 {
16
17     int c = add(10,20);
18     int s = sub(50,40);
19     printf("c = %d s = %d\n",c,s);
20
21     return 0;
22 }
```

这个`__cplusplus`是 C++ 编译器增加的功能，在用 `gcc` 编译的时候 `__cplusplus` 不成立，所以不执行`#ifdef` 里面的代码，用 g++ 编译器编译的时候执行 `ifdef` 里面的代码

```
root@ubuntu:/home/xiang/C++/funcz# gcc -o funcz funcz.c func.o
root@ubuntu:/home/xiang/C++/funcz# ls
func.c func.h func.o funcz funcz.c funcz.cpp funczcpp.cpp
root@ubuntu:/home/xiang/C++/funcz# ./funcz
c = 30 s = 10
root@ubuntu:/home/xiang/C++/funcz# vim funcz.cpp
```

这样我们把 `funcz.c` 改成 `funcz.cpp`

```
#ifdef __cplusplus
extern "C"
{
#endif

#include "func.h"

#ifdef __cplusplus
}
#endif

int main()
{
    int c = add(10,20);
    int s = sub(50,40);
    printf("c = %d s = %d\n",c,s);

    return 0;
}
```

你看同样的 C 代码用 C++ 编译器也可以编译运行了

```
root@ubuntu:/home/xiang/C++/funcz# g++ -o funcz funcz.cpp func.o
root@ubuntu:/home/xiang/C++/funcz# ./funcz
c = 30 s = 10
```

C++动态分配内存用 new，也就是 C 语言的 malloc

```
#include<stdio.h>

int main()
{
    int *p = new int;//在内存里面分配一块整形的区域，然后用p获取这块区域的地址。
    *p = 5;//向p指向的内存地址上面写数据
    *p = *p+10;

    printf("p addr = %p\n",p);
    printf("*p = %d\n",*p);
    delete p;
}

return 0;
```

```
newcpp newcpp.cpp
root@ubuntu:/home/xiang/C++/new# ./newcpp
p addr = 0x174ac20
*p = 15
```

输出正常

0x174ac20	int
地址	数字

释放内存

0x174ac20	int
地址	数字

内存模型

```
#include<stdio.h>

int main()
{
    int *p = new int[5];//在内存里面分配一块连续的数组区域

    for(int i=0;i<5;i++)
    {
        p[i] = i;
        printf("p addr = %p\n",p+i);
        printf("p[i] = %d\n",p[i]);
    }

    delete[] p;//这里一定要记住，new分配的数组，这里就一定要delete[]数组

    return 0;
}
```

new int[5] 就是向内存空间分配了 5 个 int 大小的空间
delete[] p 就是释放你申请的 5 个 int 大小的空间

0x254cc20	int
0x254cc24	int
0x254cc28	int
0x254cc2c	int
0x254cc30	int
地址	数字

内存模型

```
root@ubuntu:/home/xiang/C++/new# g++ -o newcpp2 newcpp2.cpp
root@ubuntu:/home/xiang/C++/new# ./newcpp2
p addr = 0x254cc20
p[i] = 0
p addr = 0x254cc24
p[i] = 1
p addr = 0x254cc28
p[i] = 2
p addr = 0x254cc2c
p[i] = 3
p addr = 0x254cc30
p[i] = 4
```

输出正常

下面有种有问题的写法

```
int main()
{
    int *p = new int[5];//在内存里面分配一块连续的数组区域

    for(int i=0;i<5;i++)
    {
        p[i] = i;
        printf("p addr = %p\n",p+i);
        printf("p[i] = %d\n",p[i]);
    }
    delete p; // 在释放数组内存空间的时候,我没有用 delete[] 括号来表示
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/new# g++ -o newcpp2 newcpp2.cpp
root@ubuntu:/home/xiang/C++/new# ./newcpp2
p addr = 0x254cc20
p[i] = 0
p addr = 0x254cc24
p[i] = 1
p addr = 0x254cc28
p[i] = 2
p addr = 0x254cc2c
p[i] = 3
p addr = 0x254cc30
p[i] = 4
```

输出运行也是正常的，但是.....

这种 `delete p` 释放 `new int` 是没有问题的

但是 `delete p` 释放 `new int[5]` 数组就有问题

因为在释放数组 `delete p` 只释放了数组首元素地址

剩下的 4 个数组地址没有释放，所以会出现内存泄漏

所以 C 语言 `malloc` 是按字节分配空间

C++ `new` 是按照变量类型分配空间

0x254cc20	int
0x254cc24	int
0x254cc28	int
0x254cc2c	int
0x254cc30	int
地址	数字

内存模型

C 语言 malloc 分配内存的同时无法初始化值，但是 C++ new 内存的同时可以初始化值

```
#include<stdio.h>
#include<malloc.h>

int main()
{
    int *pi = malloc(sizeof(int)); //分配内存，无法直接初始化
    float *pf = malloc(sizeof(float)); //分配内存，无法直接初始化
    char *pc = malloc(sizeof(char)); //分配内存，无法直接初始化

    *pi = 1;
    *pf = 2.5;
    *pc = 'c';

    printf("*pi = %d\n", *pi);
    printf("*pf = %f\n", *pf);
    printf("*pc = %c\n", *pc);

    free(pi);
    free(pf);
    free(pc);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/new# ./new3
*pi = 1
*pf = 2.500000
*pc = c
```

```
#include<stdio.h>

int main()
{
    int *pi = new int(1);
    float *pf = new float(2.0);
    char* pc = new char('c');

    printf("*pi = %d\n", *pi);
    printf("*pf = %f\n", *pf);
    printf("*pc = %c\n", *pc);

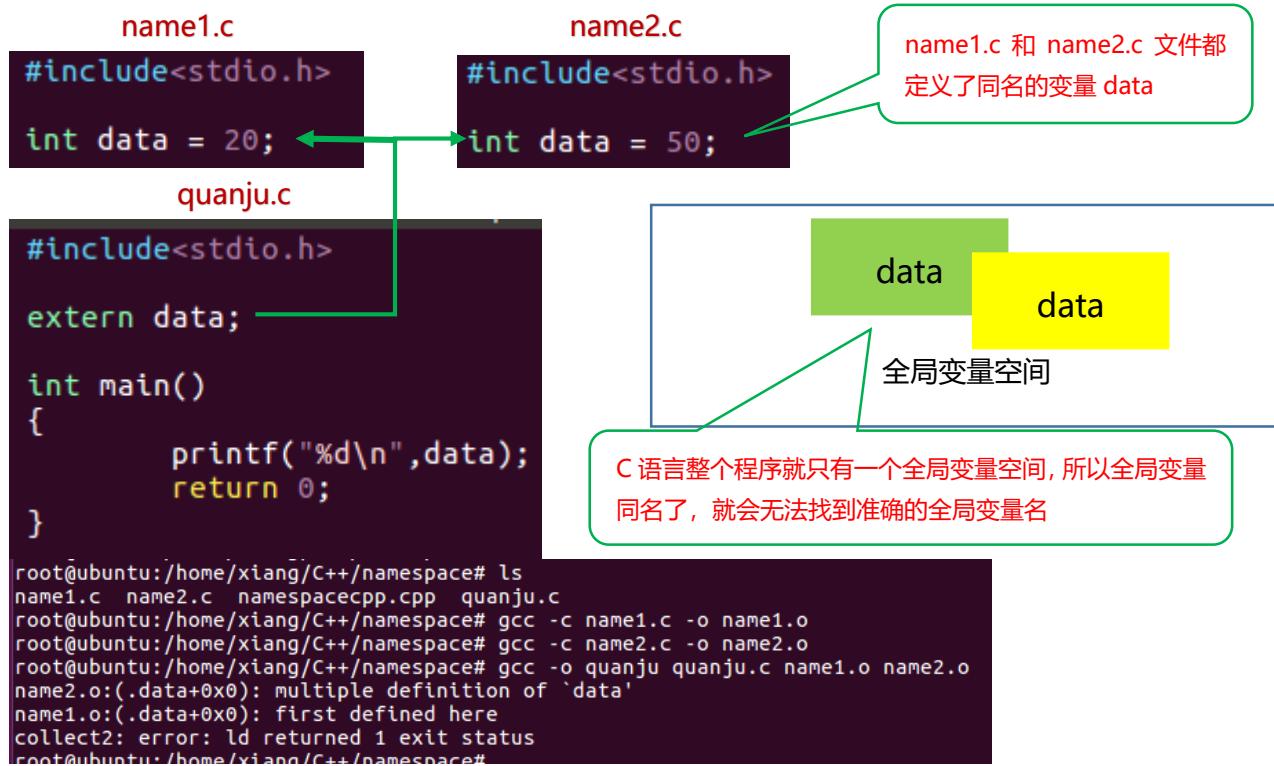
    delete pi;
    delete pf;
    delete pc;
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/new# g++ -o newcpp3 newcpp3.cpp
root@ubuntu:/home/xiang/C++/new# ./newcpp3
*pi = 1
*pf = 2.000000
*pc = c
```

C++直接把 C 语言动态分配内存和初始化两步变成了一步解决

C++命名空间，就是将 C 语言全局变量存放的区域进行划分

C 语言全局变量的问题



C 语言函数重名问题在头文件包含出问题比较多



一般小型项目不会，大型项目因为变量同名可能很多，一个 c 文件一个人去写，几个 c 文件几个人去写就

难免会出现函数或者变量同名的问题，所以大型项目很多都是 C++，JAVA 来做。C++有个 namespace 命名空间功能来划分全局变量区域，以免发生全局变量重名

```
1 #include<stdio.h>
2
3
4 namespace first
5 {
6     int i = 10;
7 }
8
9 namespace second
10 {
11     int i = 20;
12     namespace internal
13     {
14         struct p
15         {
16             int x;
17             int y;
18         };
19     }
20 }
21
22 int main()
23 {
24     using namespace first;
25     using second::internal::p;
26
27     printf("first 里面的 i = %d\n", i);
28     printf("second 里面的 i = %d\n", second::i);
29
30     p sp;
31     sp.x = 300;
32     sp.y = 500;
33
34     printf("x = %d\n", sp.x);
35     printf("y = %d\n", sp.y);
36
37     return 0;
38 }
```

这种写法声明之后，代码里面就可以直接操作变量名，编译器会自动帮你找到对应的全局变量

这是一个命名空间

这是另外一个命名空间

使用命名空间前，直接定义 using namespace (命名空间)

first 命名空间
区域

second 命名空间区域

internal 命名空间区域

全局变量区域

这就是 C++ 和 C 语言不一样的地方，C++ 将全局变量用命名空间划分开来，就不会冲突。

```
1 #include<stdio.h>
2
3
4 namespace first
5 {
6     int i = 10;
7 }
8
9 namespace second
10 {
11     int i = 20;
12     namespace internal
13     {
14         struct p
15         {
16             int x;
17             int y;
18         };
19     }
20 }
21
22 int main()
23 {
24     using namespace first;
25     using second::internal::p;
26
27     printf("first 里面的 i = %d\n", i);
28     printf("second 里面的 i = %d\n", second::i);
29
30     p sp;
31     sp.X = 300;
32     sp.y = 500;
33
34     printf("x = %d\n", sp.x);
35     printf("y = %d\n", sp.y);
36
37     return 0;
38 }
39
```

root@ubuntu:/home/xiang/C++/namespace# g++ -o namespacecpp namespacecpp.cpp
root@ubuntu:/home/xiang/C++/namespace# ./namespacecpp
first 里面的 i = 10
second 里面的 i = 20
x = 300
y = 500

如果只用 using 关键字
那么就表示要指定命名空间的
具体区域

输出结果正常

其实我觉得命名空间没什么意思，因为你还是要靠:: 符号指定空间名来寻找对应的全局变量。我以为是直接写个变量名就知道找对应位置的全局变量也!! 但是还是有一定用处。

一般小型 C++ 项目命名空间定义全局变量用得少，大型 C++ 项目用得多

C++ struct 结构体的升级

C++中结构体里面可以定义变量权限和函数，和 JAVA 一样

```
1 #include<stdio.h>
2
3 struct A
4 {
5     private: int arg;
6     private: char name;
7     public: int height;
8     public: char *arrayname;
9     void print()
10    {
11        printf("arg = %d name = %c\n",arg,name);
12    }
13 };
14
15 int main()
16 {
17     A a;
18     a.height = 100;
19     a.arrayname = "aaaabbbb";
20
21     printf("A - height = %d\n",a.height);
22     printf("%s\n",a.arrayname);
23
24     return 0;
25 }
26
```

private 表示该权限下的变量只能在 struct 结构体内部调用

Public 表示该权限下的变量可以被 struct 结构体外部调用

结构体里面可以定义函数

要调用上面的结构体内容, 要给结构体创建一个变量和 C 语言一样

用点 . 去调用结构体里面的内容, 和 C 语言一样

```
root@ubuntu:/home/xiang/C++/class# ./struct
A - height = 100
aaaabbbb
```

输出结果正常

```
struct A
{
    private: int arg;
    private: char name;
    public: int height;
    public: char *arrayname;
    void print()
    {
        printf("arg = %d name = %c\n",arg,name);
    }
}

int main()
{
    A a;
    a.height = 100;
    a.arrayname = "aaaabbbb";
    printf("A - height = %d\n",a.height);
    printf("%s\n",a.arrayname);
    a.arg = 50;
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/class# g++ -o struct struct.cpp
struct.cpp: In function 'int main()':
struct.cpp:22:14: warning: deprecated conversion from string literal to
      type 'char*' [-Wwrite-strings]
      a.arrayname = "aaaabbbb";
                  ^
struct.cpp:6:7: error: 'int A::arg' is private
      ^
struct.cpp:27:4: error: within this context
      a.arg = 50;
      ^
```

为什么 height 变量都可以,但是 arg 变量就不行?
这是因为 arg 属性属于 private, 你注意看结构体 A 权限的定义

如果是这样我需要用 private 权限里面的变量怎么办？

你可以像 JAVA 一样在类里面定义输入输出函数来间接操作变量

```
struct A
{
    private:
        int arg;
        char name;
    public:
        int height;
        char *arrayname;
    void print()
    {
        arg = 50;
        name = 'c';
        printf("arg = %d name = %c\n",arg,name);
    }
};

int main()
{
    A a;
    a.height = 100;
    a.arrayname = "aaaabbbb";

    printf("A - height = %d\n",a.height);
    printf("%s\n",a.arrayname);

    a.print();
    return 0;
}
```

我在结构体里面调用
private 属性的变量是
可以的

我用结构体里面的 print 函数
来间接输出 private 属性的变
量，因为 print 属性是 public

```
root@ubuntu:/home/xiang/C++/class# ./struct
A - height = 100
aaaabbbb
arg = 50 name = c
```

输出结果正常

C++结构体可以像 JAVA 那样具有继承性，也就是子类继承父类，孙子类继承子类这种复杂的继承关系

C++用 class 来代替 struct

因为 struct 是 C 语言里面用的，所以在 C++里面虽然 struct 和 class 属性是一样的，但是为了将 C++和 C 语言分开来表示，C++里面还是用 class 居多。

```
#include<stdio.h>

struct A
{
    int i;
    void print()
    {
        i = 10;
        printf("i = %d\n",i);
    }
};

int main()
{
    A a;
    a.print();
    return 0;
}
```

struct 定义的
变量函数默认
是 public

```
#include<stdio.h>

class B
{
    int i;
    void print()
    {
        i = 10;
        printf("i = %d\n",i);
    }
};

int main()
{
    B b;
    b.print();
    return 0;
}
```

class 定
义的变量
函数默认
private

```
#include<stdio.h>

struct A
{
    int i;
    void print()
    {
        i = 10;
        printf("i = %d\n",i);
    }
};

int main()
{
    A a;
    a.print();
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/class# g++ -o classA classA.cpp
root@ubuntu:/home/xiang/C++/class# ./classA
i = 10
root@ubuntu:/home/xiang/C++/class#
```

```
#include<stdio.h>

class B
{
    int i;
    void print()
    {
        i = 10;
        printf("i = %d\n",i);
    }
};

int main()
{
    B b;
    b.print();
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/class# g++ -o classB classB.cpp
classB.cpp: In function ‘int main()’:
classB.cpp:6:7: error: ‘void B::print()’ is private
  void print()
      ^
classB.cpp:16:10: error: within this context
  b.print();
      ^
```

这个错误就是 class 里面
定义的函数和变量默认都
是 private 属性的

所以 C++ 里面 struct 和 class 的唯一区别就是类成员的默认属性。

C++创建对象时，对象在内存的分布

```
#include<stdio.h>

class test
{
    private:
        int i;
        int j;
    public:
        int getI()
        {
            return i;
        }
        int getJ()
        {
            return j;
        }
};

test gt; //这是将对象静态分配在全局区
int main()
{
    int gti = gt.getI();
    int gtj = gt.getJ();
    printf("gt i = %d\n",gti);
    printf("gt j = %d\n",gtj);

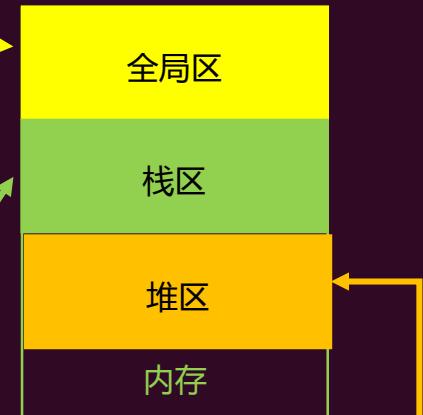
    test t1; //这是将对象静态分配在栈上
    int t1i = t1.getI();
    int t1j = t1.getJ();
    printf("t1 i = %d\n",t1i);
    printf("t1 j = %d\n",t1j);

    test *pt = new test; //C++ new就是C语言malloc将对象分配在堆上面
    int pti = pt->getI();
    int ptj = pt->getJ();
    printf("pt i = %d\n",pti);
    printf("pt j = %d\n",ptj);

    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/class# g++ -o init init.cpp
root@ubuntu:/home/xiang/C++/class# ./init
```

```
gt i = 0 } 在全局区变量默认初始值为 0
gt j = 0 }
t1 i = 4196320 } 在栈区的变量初始值是随机的
t1 j = 0 }
pt i = 0 } 在堆区的变量初始值是随机的
pt j = 0 }
```



我们可以用 C 语言的思维方式写一个变量初始化函数

```
class test
{
    private:
        int i;
        int j;
public:
    int getI()
    {
        return i;
    }
    int getJ()
    {
        return j;
    }
    void initialize()
    {
        i = 10;
        j = 20;
    }
};

test gt;
int main()
{
    gt.initialize(); //调用类里面的initialize函数来初始化变量值
    int gti = gt.getI();
    int gtj = gt.getJ();
    printf("gt i = %d\n",gti);
    printf("gt j = %d\n",gtj);

    test t1;
    t1.initialize(); //调用类里面的initialize函数来初始化变量值
    int t1i = t1.getI();
    int t1j = t1.getJ();
    printf("t1 i = %d\n",t1i);
    printf("t1 j = %d\n",t1j);

    test *pt = new test;
    pt->initialize(); //调用类里面的initialize函数来初始化变量值
    int pti = pt->getI();
    int ptj = pt->getJ();
    printf("pt i = %d\n",pti);
    printf("pt j = %d\n",ptj);

    delete pt; //记住动态分配的类一定要释放
    return 0;
}
```

输出结果正常，但是我们发现每次去创建一个不同的对象都要去调用一次 initialize 函数，感觉很累

我们将用构造函数解决这个很累的问题

C++构造函数，让你创建对象时自动初始化对象

```
class test
{
    private:
        int i;
        int j;
    public:
        int getI()
        {
            return i;
        }
        int getJ()
        {
            return j;
        }
        test()
        {
            printf("enter test\n");
            i = 10;
            j = 20;
            printf("exit test\n");
        }
};

test gt;
int main()
{
    int gti = gt.getI();
    int gtj = gt.getJ();
    printf("gt i = %d\n",gti);
    printf("gt j = %d\n",gtj);

    test t1;
    int t1i = t1.getI();
    int t1j = t1.getJ();
    printf("t1 i = %d\n",t1i);
    printf("t1 j = %d\n",t1j);

    test *pt = new test;
    int pti = pt->getI();
    int ptj = pt->getJ();
    printf("pt i = %d\n",pti);
    printf("pt j = %d\n",ptj);

    delete pt;//记住动态分配的类一定要释放
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/class# ./init
enter test
exit test
gt i = 10
gt j = 20
enter test
exit test
t1 i = 10
t1 j = 20
enter test
exit test
pt i = 10
pt j = 20
```

构造函数就是在类
里面创建一个和类
名字一样的函数

创建一个对象就会
自动去调用一次构
造函数

输出正常

C++可以在创建对象的时候，自己根据对象的不同来赋值不同的初始化值给构造函数

```
class test
{
public:
    test()
    {
        printf("test()\n");
    }
    test(int v)
    {
        printf("test v = %d\n",v);
    }
};

int main()
{
    test t;      //调用构造函数test()
    test t1(10); //创建对象时初始化了参数，所以调用构造函数test(int v)
    test t2 = 20; //创建对象时初始化了参数，= 和()一样,调用构造函数test(int v)
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/class# ./init2
test()
test v = 10
test v = 20
root@ubuntu:/home/xiang/C++/class#
```

这就是不同对象初始化值

C++用数组方式初始化构造函数

```
#include<stdio.h>

class test
{
public:
    test()
    {
        printf("test()\n");
    }
    test(int v)
    {
        printf("test v = %d\n",v);
    }
};

int main()
{
    test ta[3];
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/class# ./init3
test()
test()
test()
```

这是没有赋值的对象数组初始化

如何给对象数组里面的每个元素对象赋不同的初始化值呢？

```

class test
{
public:
    test()
    {
        printf("test()\n");
    }
    test(int v)
    {
        printf("test v = %d\n",v);
    }
};

int main()
{
    test ta[3] = {test(),test(1),test(2)};
    return 0;
}

```

root@ubuntu:/home/xiang/C++/class# ./init3
test()
test v = 1
test v = 2

像 C 语言一样给数组里面每个元素赋值，只不过 C++ 是像每个对象元素的构造函数赋值

ta[0]调用 test()
ta[1]调用 test(int v)
ta[2]调用 test(int v)

输出正确

C++多文件协同开发

我们开发一个数组安全的程序

- 1.提供函数获取数组长度
- 2.提供函数获取数组元素
- 3.提供函数设置数组元素

A 工程师实现的东西
test.c
intarray.h

B 工程师实现的东西
根据 A 工程师 intarray.h 文件的要求，实现
intarray.cpp

intarray.h

```

#ifndef _INTARRAY_H
#define _INTARRAY_H

class intarray
{
private:
    int m_length;
    int *m_pointer;
public:
    intarray(int len); //构造函数
    int length(); //用来得到数组的长度
    bool get(int index,int &value); //得到数组里面对应元素位置的值
    bool set(int index,int value); //设置数组里面对应元素位置的值
};

#endif

```

这就是 A 工程师创建的头文件定义的需求

头文件和前面的 class 不一样，以前定义 class 都在 cpp 文件里面，所以在 cpp 文件里面要实现类函数功能，现在 class 定义在头文件，所以只需要声明不用实现。实现交给对应的 cpp 文件就是了。

然后 B 工程师在 intarray.cpp 文件里实现 intarray.h 里面声明的函数

intarray.cpp

```
1 #include "intarray.h"
2
3 intarray(int len)
4 {
5
6 }
7 int length()
8 {
9
10 }
11 bool get(int index,int &value)
12 {
13
14 }
15 bool set(int index,int value)
16 {
17
18 }
19
```

怎么 B 工程师去实现
intarray.h 声明的函数会报
错？

这是因为你没有给函数添加作用域，编译器以为是全局函数，而不是某个类里面声明的函数

```
#include "intarray.h"

intarray::intarray(int len)
{
    m_pointer = new int[len];
    for(int i=0;i<len;i++)
    {
        m_pointer[i] = 0;
    }
    m_length = len;
}

//用来得到数组的长度
int intarray::length()
{
    return m_length;
}

//得到数组里面对应元素位置的值
bool intarray::get(int index,int &value)
{
    bool ret = (0<=index)&&(index<length());
    if(ret)
    {
        value = m_pointer[index];
    }
    return ret;
}

//设置数组里面对应元素位置的值
bool intarray::set(int index,int value)
{
    bool ret = (0<=index)&&(index<length());
    if(ret)
    {
        m_pointer[index] = value;
    }
    return ret;
}
```

用 :: 符号在函数前面增加类名，这样编译器就知道你这个函数属于哪个类了，这个 :: 符号和命名空间一模一样，使用的时候要注意

```
root@ubuntu:/home/xiang/C++/class/classarray# g++ -o test test.cpp intarray.cpp  
root@ubuntu:/home/xiang/C++/class/classarray#
```

test.cpp

```
#include<stdio.h>  
#include "intarray.h"  
  
int main()  
{  
    intarray a(5);  
    for(int i=0;i<a.length();i++)  
    {  
        a.set(i,i+1);  
    }  
    for(int i=0;i<a.length();i++)  
    {  
        int value = 0;  
        if(a.get(i,value))  
        {  
            printf("a[%d] = %d\n",i,value);  
        }  
        else  
            printf("数组越界\n");  
    }  
    return 0;  
}
```

C++ 主文件调用子文件程序可以两个 cpp 一起编译

```
root@ubuntu:/home/xiang/C++/class/classarray# g++ -o test test.cpp intarray.cpp  
root@ubuntu:/home/xiang/C++/class/classarray# ./test  
a[0] = 1  
a[1] = 2  
a[2] = 3  
a[3] = 4  
a[4] = 5
```

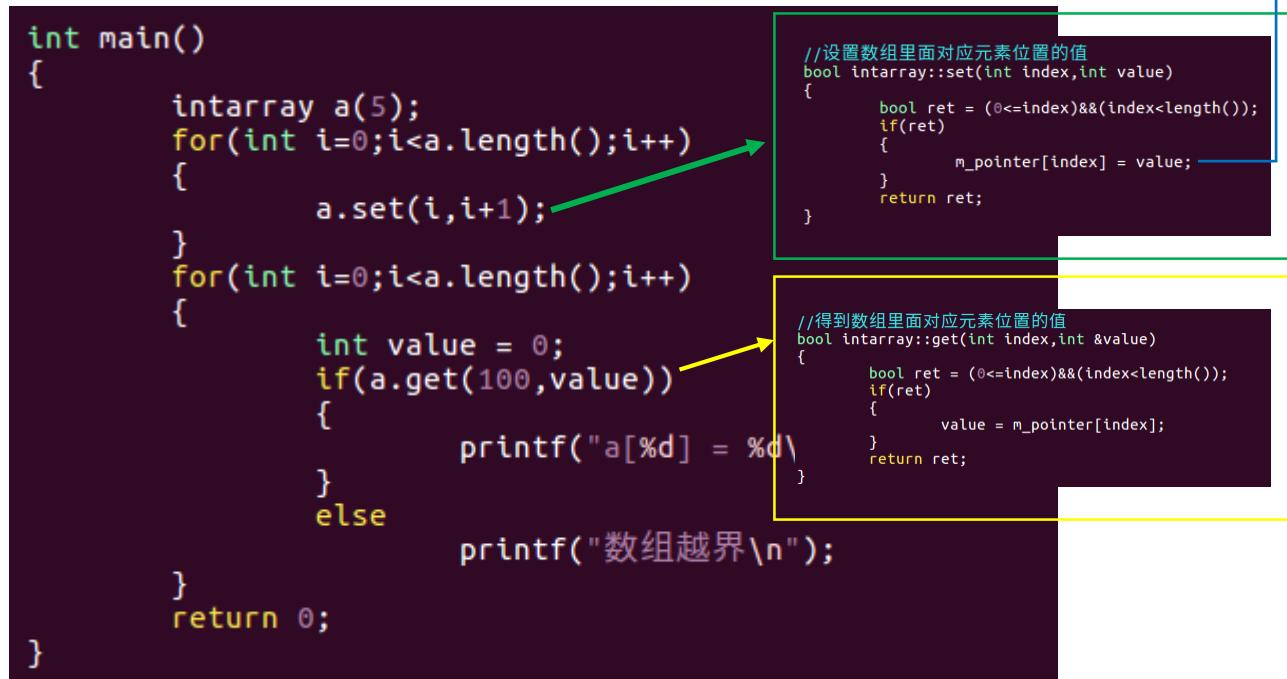
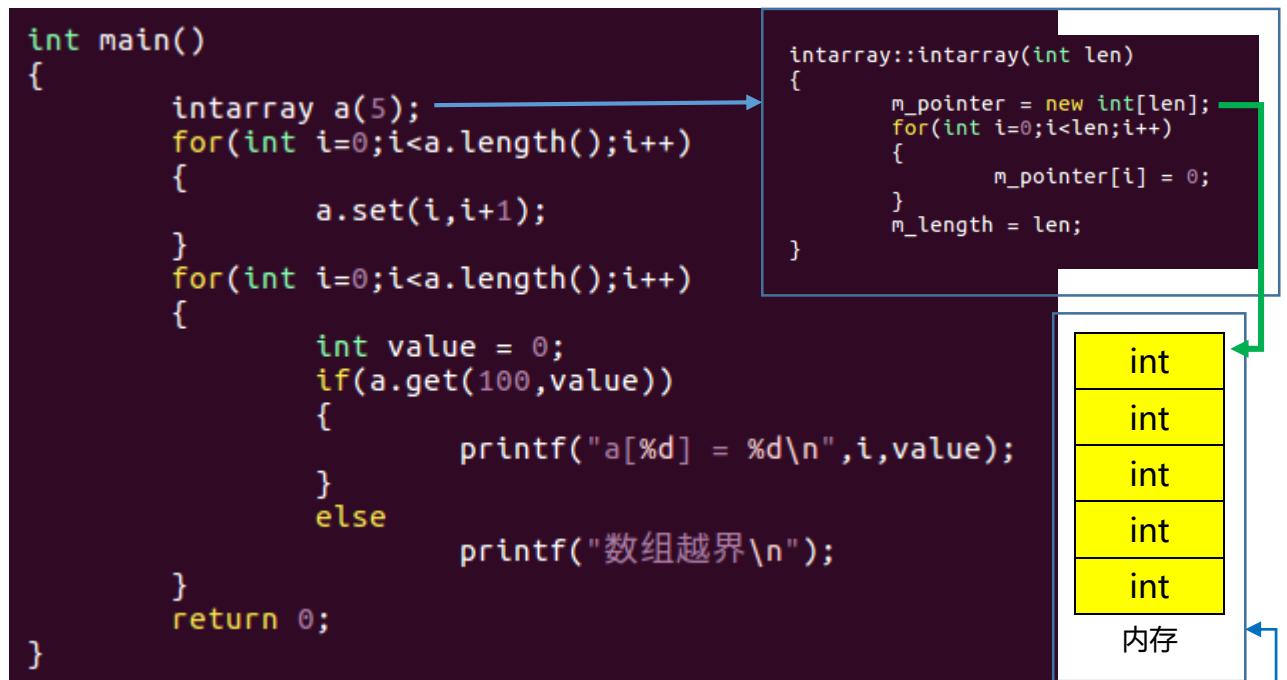
输出正常

```
for(int i=0;i<a.length();i++)  
{  
    int value = 0;  
    if(a.get(100,value))  
    {  
        printf("a[%d] = %d\n",i,value);  
    }  
    else  
        printf("数组越界\n");  
}
```

把 test.cpp 文件里面读取数组的位置改成 100 位置开始读取

```
root@ubuntu:/home/xiang/C++/class/classarray# g++ -o test test.cpp intarray.cpp  
root@ubuntu:/home/xiang/C++/class/classarray# ./test  
数组越界  
数组越界  
数组越界  
数组越界  
数组越界
```

数组安全程序具体分析



这就是数组安全代码的具体实现

两个对象相互赋值

```
class test
{
    private:
        int i;
        int j;
    public:
        int getI()
        {
            return i;
        }
        int getJ()
        {
            return j;
        }
};

int main()
{
    test t1; _____
    test t2; _____
    printf(" t1.i = %d t1.j = %d\n",t1.getI(),t1.getJ());
    printf(" t2.i = %d t2.j = %d\n",t2.getI(),t2.getJ());
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/copy# g++ -o copy copy.cpp
root@ubuntu:/home/xiang/C++/copy# ./copy
t1.i = 0 t1.j = 0
t2.i = 4195920 t2.j = 0
```

所以 t1 和 t2 两个对象的 i 和 j 值都是随机的

```
class test
{
    private:
        int i;
        int j;
    public:
        int getI()
        {
            return i;
        }
        int getJ()
        {
            return j;
        }
};

int main()
{
    test t1;
    test t2 = t1; _____
    printf(" t1.i = %d t1.j = %d\n",t1.getI(),t1.getJ());
    printf(" t2.i = %d t2.j = %d\n",t2.getI(),t2.getJ());
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/copy# g++ -o copy copy.cpp
root@ubuntu:/home/xiang/C++/copy# ./copy
t1.i = 0 t1.j = 0
t2.i = 0 t2.j = 0
```

这样 t2 里面的变量值就等于 t1 了。

其实上面这种对象赋值就是 C++ 编译器在 class 里面自动生成了拷贝构造函数

```
class test
{
    private:
        int i;
        int j;
    public:
        int getI()
        {
            return i;
        }
        int getJ()
        {
            return j;
        }
        test(const test& t)
        {
            i = t.i;
            j = t.j;
        }
};

int main()
{
    test t1;
    test t2 = t1;
    printf(" t1.i = %d t1.j = %d\n",t1.getI(),t1.getJ());
    printf(" t2.i = %d t2.j = %d\n",t2.getI(),t2.getJ());
    return 0;
}

root@ubuntu:/home/xiang/C++/copy# g++ -o copy copy.cpp
copy.cpp: In function 'int main()':
copy.cpp:26:7: error: no matching function for call to 'test::test()'
    test t1;
          ^
copy.cpp:17:3: note: candidate: test::test(const test&
    test(const test & t)
          ^
```

这是 C++ 编译器自动加的拷贝构造函数 (const 类名 引用) 这种格式就是拷贝构造函数, 我写在这里只是方便大家看, 其实这里是没有的

为什么编译出错, 因为你既然要在类里面写不同类型的构造函数, 那么你就必须多写一个无参构造函数

```
class test
{
    private:
        int i;
        int j;
    public:
        int getI()
        {
            return i;
        }
        int getJ()
        {
            return j;
        }
        test(const test& t)
        {
            i = t.i;
            j = t.j;
        }
        test()
        {
        }
};

int main()
{
    test t1;
    test t2 = t1;
    printf(" t1.i = %d t1.j = %d\n",t1.getI(),t1.getJ());
    printf(" t2.i = %d t2.j = %d\n",t2.getI(),t2.getJ());
    return 0;
}
```

增加一个无参
构造函数

输出结果一样, 这就是构造函数拷贝的由来

```
copy.cpp:17:3: note: candidate expects 1 argument, o
root@ubuntu:/home/xiang/C++/copy# g++ -o copy copy.cpp
root@ubuntu:/home/xiang/C++/copy# ./copy
    t1.i = 0 t1.j = 0
    t2.i = 0 t2.j = 0
root@ubuntu:/home/xiang/C++/copy#
```

上面这种两个对象赋值是属于浅拷贝，下面我们讲下深拷贝。

```
2 class test
3 {
4     private:
5         int i;
6         int j;
7         int *p;
8     public:
9         int getI()
10        {
11            return i;
12        }
13        int getJ()
14        {
15            return j;
16        }
17        int *getp()
18        {
19            return p; //返回p现在指向的地址
20        }
21        test(int v)
22        {
23            i = 10;
24            j = 20;
25            p = new int; //创建一个新的int地址让p指向它
26            *p = v; //将v的值赋值给p指向的新int地址
27        }
28    };
29
30 int main()
31 {
32     test t1(30);
33     test t2 = t1; //修改初始化格式为括号,因为类里面提供了带参数的构造函数,所以编译器就不会在类里面提供默认的构造函数了
34     printf(" t1.i = %d t1.j = %d t1.p = %p\n",t1.getI(),t1.getJ(),t1.getp());
35     printf(" t2.i = %d t2.j = %d t2.p = %p\n",t2.getI(),t2.getJ(),t2.getp());
36     return 0;
37 }
```

```
root@ubuntu:/home/xiang/C++/copy# g++ -o copy copy.cpp
root@ubuntu:/home/xiang/C++/copy# ./copy
t1.i = 10 t1.j = 20 t1.p = 0xe87c20
t2.i = 10 t2.j = 20 t2.p = 0xe87c20
```

因为类里面自带了拷贝构造函数，所以 t1 和 t2 的值是一样的，没有问题

但是 t1 和 t2 因为是 2 个对象，所以应该地址也不一样啊，怎么会一样呢？这就是内存没有释放造成的

用列表来解决类无法初始化类里面成员变量的问题

```
1 #include<stdio.h>
2
3 class test
4 {
5     private:
6         int ci = 10;
7     public:
8         int getCI()
9         {
10             return ci;
11         }
12 }
13
14 int main()
15 {
16     test t1;
17     printf("t1.ci = %d\n",t1.getCI());
18
19     return 0;
20 }
```

constclass.cpp:6:12: warning: non-static data member initializers only available with -std=c++11 or -std=gnu++11
int ci = 10;

C++给类里面的成员变量初始化值会报警告，所以C++不像JAVA那样可以给类成员赋初始化值

```
class test
{
    private:
        int ci;
    public:
        int getCI()
        {
            ci = 10;
            return ci;
        }
};

int main()
{
    test t1;
    printf("t1.ci = %d\n",t1.getCI());

    return 0;
}
```

C++要初始化类的成员变量只有交给类里面的函数来做

```
root@ubuntu:/home/xiang/C++/constdir# g++ -o constclass constclass.cpp
root@ubuntu:/home/xiang/C++/constdir# ./constclass
t1.ci = 10
```

这样就有一个麻烦的地方，就是我外部调用类的时候，如果我要初始化类的成员变量，我还要调用一句类里面的初始化变量函数。

```

3 class test
4 {
5     private:
6         int ci;
7     public:
8         test() : ci(10)
9         {
10
11     }
12         int getCI()
13         {
14             return ci;
15         }
16
17 };
18
19 int main()
20 {
21     test t1;
22     printf("t1.ci = %d\n",t1.getCI());
23
24     return 0;
25 }
```

因为我们是想主函数在调用类创建对象的时候就初始化类里面的成员变量, 所以我要创建一个构造函数

函数名 : 变量(值)
这种语法就是初始化类里面的成员

```

root@ubuntu:/home/xiang/C++/constdir# g++ -o constclass constclass.cpp
root@ubuntu:/home/xiang/C++/constdir# ./constclass
t1.ci = 10
root@ubuntu:/home/xiang/C++/constdir#
```

```

3 class test
4 {
5     private:
6         int ci;
7         long l;
8         char c;
9     public:
10        test() : ci(10),l(50000),c('c')
11        {
12
13        }
14        int getCI()
15        {
16            return ci;
17        }
18        long getl()
19        {
20            return l;
21        }
22        char getc()
23        {
24            return c;
25        }
26
27 };
28
29 int main()
30 {
31     test t1;
32     printf("t1.ci = %d\n",t1.getCI());
33     printf("t1.l = %ld\n",t1.getl());
34     printf("t1.c = %c\n",t1.getc());
35
36     return 0;
37 }
```

这是类里面多个成员变量初始化方法

```

root@ubuntu:/home/xiang/C++/constdir# g++ -
root@ubuntu:/home/xiang/C++/constdir# ./constclass
t1.ci = 10
t1.l = 50000
t1.c = c
root@ubuntu:/home/xiang/C++/constdir#
```

类初始化另外一个类的对象

```
class value
{
    private:
        int mi;
    public:
        value(int i)
        {
            printf("value i = %d\n", i);
            mi = i;
        }
        int getI()
        {
            return mi;
        }
};

class test
{
    private:
        value v1(1);
        value v2(2);
        value v3(3);
    public:
        test()
        {
        }
};

int main()
{
    test t1;

    return 0;
}
```

root@ubuntu:/home/xiang/C++/constdir# g++ -o constclass constclass.cpp
constclass.cpp:23:12: error: expected identifier before numeric constant
 value v1(1);
 ^
constclass.cpp:23:12: error: expected ‘,’ or ‘...’ before numeric constant
constclass.cpp:24:12: error: expected identifier before numeric constant
 value v2(2);
 ^
constclass.cpp:24:12: error: expected ‘,’ or ‘...’ before numeric constant
constclass.cpp:25:12: error: expected identifier before numeric constant
 value v3(3);
 ^
constclass.cpp:25:12: error: expected ‘,’ or ‘...’ before numeric constant

编译直接报错

我们在类里面调用另外
一个类创建对象, 然后按
照我们常规对象初始化
方法是不行的, 因为我是
是类调用类

```

class value
{
    private:
        int mi;
    public:
    value(int i)
    {
        printf("value i = %d\n",i);
        mi = i;
    }
    int getI()
    {
        return mi;
    }
};

class test
{
    private:
        value v1;
        value v2;
        value v3;
    public:
        test() : v1(1),v2(2),v3(3)
    {
    }
};

int main()
{
    test t1;
    return 0;
}

```

C++的类成员变量不管是类
创建的对象，还是创建的变
量都必须用列表来初始化

```

root@ubuntu:/home/xiang/C++/constdir# ./constclass
value i = 1
value i = 2
value i = 3
root@ubuntu:/home/xiang/C++/constdir# 

```

输出结果正常

这里有个列表初始化顺序的问题

```

class test
{
    private:
        value v2;
        value v1;
        value v3;
    public:
        test() : v1(1),v2(2),v3(3)
    {
    }
};

```

列表初始化的顺序是和
定义的时候顺序有关

列表初始化顺序和函数
填写列表的顺序无关

```

root@ubuntu:/home/xiang/C++/const
value i = 2
value i = 1
value i = 3

```

const 在类中的初始化也要用到列表

我们前面([const 常量修饰 C 语言和 C++ 的区别](#))中定义的 const 是在主函数或者子函数里面定义的
现在我们把 const 定义在类里面

```
class test
{
    private:
        const int ci;
    public:
        int getCI()
        {
            return ci;
        }
};

int main()
{
    test t1;
    return 0;
}
```

root@ubuntu:/home/xiang/C++/constdir# g++ -o constclass constclass.cpp
constclass.cpp: In function ‘int main()’:
constclass.cpp:18:7: error: uninitialized const member in ‘class test’
 test t1;
 ^
constclass.cpp:8:13: note: ‘const int test::ci’ should be initialized
 const int ci;

在类里面定义常量没有初始化编译时要出问题的

```
class test
{
    private:
        const int ci;
    public:
        int getCI()
        {
            ci = 10;
            return ci;
        }
};

int main()
{
    test t1;
    return 0;
}
```

root@ubuntu:/home/xiang/C++/constdir# g++ -o constclass constclass.cpp
constclass.cpp: In member function ‘int test::getCI()’:
constclass.cpp:12:7: error: assignment of read-only member ‘test::ci’
 ci = 10;
 ^
constclass.cpp: In function ‘int main()’:
constclass.cpp:19:7: error: uninitialized const member in ‘class test’
 test t1;
 ^
constclass.cpp:8:13: note: ‘const int test::ci’ should be initialized
 const int ci;

在类函数里面初始化常量是不对的，因为这样写是给 ci 赋值，但是常量是不需要赋值的

```

class test
{
private:
    const int ci = 10;
public:
    int getCI()
    {
        return ci;
    }
};

int main()
{
    test t1;
    printf("t1.ci = %d\n", t1.getCI());
    return 0;
}

```

这样就给 ci 常量初始化了，这样是可以，但是 C++ 认为这种做法不妥会给出警告，因为我们说过类里面的变量初始化不能直接用 = 符号，只有赋值才能用 = 符号

```

root@ubuntu:/home/xiang/C++/constdir# g++ -o constclass constclass.cpp
constclass.cpp:8:18: warning: non-static data member initializers only ava
    const int ci = 10;
               ^
root@ubuntu:/home/xiang/C++/constdir# ./constclass
t1.ci = 10
root@ubuntu:/home/xiang/C++/constdir# 

```

```

class test
{
private:
    const int ci;
public:
    test() : ci(10)
    {
    }
    int getCI()
    {
        return ci;
    }
};

int main()
{
    test t1;
    printf("t1.ci = %d\n", t1.getCI());
    return 0;
}

```

常量在类里面正确的初始化方法还是构造函数加列表的方法。

```

root@ubuntu:/home/xiang/C++/constdir# g++ -o constclass constclass.cpp
root@ubuntu:/home/xiang/C++/constdir# ./constclass
t1.ci = 10

```

析构函数帮你自动释放 new 创建的对象占用的内存

```
class test
{
    private:
        int *m_pointer;
    public:
        test()
        {
            m_pointer = new int[5];
        }
        void arrayfor(int i)
        {
            for(int z=0;z<5;z++)
            {
                m_pointer[z] = i;
            }
            for(int z=0;z<5;z++)
            {
                printf("m_pointer = %d\n",m_pointer[z]);
            }
        }
        void arrayfree()
        {
            delete[] m_pointer;
        }
};

int main()
{
    test t1;
    t1.arrayfor(50);
    t1.arrayfree(); //记住要手动释放堆空间的内存

    return 0;
}
```

root@ubuntu:/home/xiang/C++/destruct# g++ -o destruct destruct.cpp
root@ubuntu:/home/xiang/C++/destruct# ./destruct
m_pointer = 50
m_pointer = 50
m_pointer = 50
m_pointer = 50
m_pointer = 50

我们前面的做法就是在 class 里面
创建堆内存释放函数

用完了 t1 对象,一定要记住在这里
调用 class 里面的 arrayfree 函数
来释放堆内存

如果我主函数代码很多,忘记最后调用对象里面的释放内存函数,那岂不是就造成了内存泄漏

所以这里用析构函数就能解决这个问题

```

class test
{
    private:
        int *m_pointer;
    public:
        test()
        {
            m_pointer = new int[5];
            printf("test\n");
        }
        void arrayfor(int i)
        {
            for(int z=0;z<5;z++)
            {
                m_pointer[z] = i;
            }
            for(int z=0;z<5;z++)
            {
                printf("m_pointer = %d\n",m_pointer[z]);
            }
        }
        ~test() //析构函数在类里面是没有返回值和形参的，而且在类里面只能有一个，析构函数和构造函数一样都是用类名
        {
            delete[] m_pointer;
            printf("~~test~~\n");
        }
};

int main()
{
    test t1;
    t1.arrayfor(50);

    printf("-----\n");
    return 0;
}

```

这是析构函数

```

root@ubuntu:/home/xiang/C++/destruct# ./destruct
test
m_pointer = 50
-----
~~test~~
root@ubuntu:/home/xiang/C++/destruct#

```

在程序执行完之后会自动执行类里面的析构函数释放内存

```

class test
{
    private:
        int *m_pointer;
    public:
        test()
        {
            m_pointer = new int[5];
            printf("test\n");
        }
        void arrayfor(int i)
        {
            for(int z=0;z<5;z++)
            {
                m_pointer[z] = i;
            }
            for(int z=0;z<5;z++)
            {
                printf("m_pointer = %d\n",m_pointer[z]);
            }
        }
        ~test() //析构函数在类里面是没有返回值和形参的，而且在类里面只能有一个，析构函数和构造函数一样都是用类名
        {
            delete[] m_pointer;
            printf("~~test~~\n");
        }
};

int main()
{
    test t1;
    t1.arrayfor(50);

    printf("-----\n");
    return 0;
}

```

但是我发现个问题，析构函数不是对象执行完了就执行，而是整个主函数结束了才执行，这其实不是我想要的

```

class test
{
    private:
        int *m_pointer;
    public:
        test()
        {
            m_pointer = new int[5];
            printf("test\n");
        }
        void arrayfor(int i)
        {
            for(int z=0;z<5;z++)
            {
                m_pointer[z] = i;
            }
            for(int z=0;z<5;z++)
            {
                printf("m_pointer = %d\n",m_pointer[z]);
            }
        }
        ~test() //析构函数在类里面是没有返回值和形参的，而且在类里面只能有一个，析构函数和构造函数一样都是用类名
        {
            delete[] m_pointer;
            printf("~test~\n");
        }
};

void func()
{
    test t1;
    t1.arrayfor(50);
}

int main()
{
    func();
    printf("-----\n");
    return 0;
}

```

经过代码修改，我把对象放在子函数里面

```

root@ubuntu:~/home/xiang/C++/destruct# g++ -o destruct des
root@ubuntu:/home/xiang/C++/destruct# ./destruct
test
m_pointer = 50
~test~
-----
root@ubuntu:/home/xiang/C++/destruct#

```

发现子函数执行完了析构函数就会执行，不像上面那样必须等待主函数执行完析构函数才执行

证明主函数里面对 class 类里面堆内存的释放还是需要手动 delete，但是子函数就不用 delete，子函数执行完，析构函数就自动执行了，所以析构函数还是比较适合用于子函数，因为子函数是放在栈空间的，一旦子函数结束，栈就会被自动释放，析构函数也会跟着执行。

用 const 修饰类的对象

```
1 #include<stdio.h>
2
3 class test
4 {
5     int mi;
6 public:
7     int mj;
8     test(int i);
9     test(const test& t);
10    int getMi();
11 };
12
13 test::test(int i)
14 {
15     mi = i;
16 }
17 test::test(const test& t)
18 {
19 }
20
21 int test::getMi()
22 {
23     return mi;
24 }
25
26
27 int main()
28 {
29     test t(1);
30     t.mj = 1000;
31     return 0;
32 }
33
```

root@ubuntu:/home/xiang/C++/constdir# g++ -o objectconst objectconst.cpp
root@ubuntu:/home/xiang/C++/constdir# ./objectconst
root@ubuntu:/home/xiang/C++/constdir#

编译没有问题

```
1 #include<stdio.h>
2
3 class test
4 {
5     int mi;
6 public:
7     int mj;
8     test(int i);
9     test(const test& t);
10    int getMi();
11 };
12
13 test::test(int i)
14 {
15     mi = i;
16 }
17 test::test(const test& t)
18 {
19 }
20
21 int test::getMi()
22 {
23     return mi;
24 }
25
26
27 int main()
28 {
29     const test t(1);
30     t.mj = 1000;
31     return 0;
32 }
33
```

我给对象加入 const

编译报错,因为对象加了 const, 那么对象里面的成员就是只读变量, 不能用主函数去修改对象里面的变量

root@ubuntu:/home/xiang/C++/constdir# g++ -o objectconst objectconst.cpp
objectconst.cpp: In function 'int main()':
objectconst.cpp:30:7: error: assignment of member 'test::mj' in read-only object
t.mj = 1000;

```

3 class test
4 {
5     int mi;
6 public:
7     int mj;
8     test(int i);
9     test(const test& t);
10    int getMi();
11 };
12
13 test::test(int i)
14 {
15     mi = i;
16 }
17 test::test(const test& t)
18 {
19 }
20
21
22 int test::getMi()
23 {
24     return mi;
25 }
26
27 int main()
28 {
29     const test t(10);
30     printf("getMi = %d\n",t.getMi());
31     return 0;
32 }
33

```

因为我们对象是 const 对象

所以主程序想调用对象里面的函数必须是加了 const 的成员函数才行

```

root@ubuntu:/home/xiang/C++/constdir# g++ -o objectconst objectconst.cpp
objectconst.cpp: In function ‘int main()’:
objectconst.cpp:30:32: error: passing ‘const test’ as ‘this’ argument discards qualifiers [-fpermissive]
    printf("getMi = %d\n",t.getMi());
                                         ^
objectconst.cpp:22:5: note:   in call to ‘int test::getMi()’
    int test::getMi()
    ^

```

```

3 class test
4 {
5     int mi;
6 public:
7     int mj;
8     test(int i);
9     test(const test& t);
10    int getMi()const;
11 };
12
13 test::test(int i)
14 {
15     mi = i;
16 }
17 test::test(const test& t)
18 {
19 }
20
21
22 int test::getMi()const
23 {
24     return mi;
25 }
26
27 int main()
28 {
29     const test t(10);
30     printf("getMi = %d\n",t.getMi());
31     return 0;
32 }
33

```

我们在函数后面加上 const 关键字, 这样这个函数就变成了 const 成员函数

这里实现也不要忘了加 const, 定义加了 const, 那么函数实现也要加 const

输出没有问题

```

root@ubuntu:/home/xiang/C++/constdir# g++ -o objectconst objectconst.cpp
root@ubuntu:/home/xiang/C++/constdir# ./objectconst
getMi = 10

```

```

2 int test::getMi() const
3 {
4     mi = 20;
5     return mi;
6 }
7
8 int main()
9 {
10     const test t(10);
11     printf("getMi = %d\n", t.getMi());
12     return 0;
13 }

```

加了 const 的成员函数只能读变量，不能写变量

```

root@ubuntu:/home/xiang/C++/constdir# g++ -o objectconst objectconst.cpp
objectconst.cpp: In member function ‘int test::getMi() const’:
objectconst.cpp:24:5: error: assignment of member ‘test::mi’ in read-only object
    mi = 20;
    ^
root@ubuntu:/home/xiang/C++/constdir#

```

计算一个程序里面对象创建的次数，但是不允许用全局变量来计算

```

#include<stdio.h>

class test
{
private:
    int mCount;
public:
    test():mCount(0)
    {
        mCount++;
    }
    ~test()
    {
        --mCount;
    }
    int getCount()
    {
        return mCount;
    }
};

test gettest;

int main()
{
    test t1;
    test t2;
    printf("gettest count = %d\n",gettest.getCount());
    printf("t1 count = %d\n",t1.getCount());
    printf("t2 count = %d\n",t2.getCount());

    return 0;
}

```

```

root@ubuntu:/home/xiang/C++/quanju# ./quanju
gettest count = 1
t1 count = 1
t2 count = 1
root@ubuntu:/home/xiang/C++/quanju#

```

很明显每个对象的 mCount 变量

都是对象自己的，所以三个对象无法共用这个 mCount 变量，各是各的。

```

#include<stdio.h>

int qCount = 0;

class test
{
    private:
        int mCount;
    public:
        test():mCount(0)
        {
            qCount++;
        }
        ~test()
        {
            --qCount;
        }
        int getCount()
        {
            return qCount;
        }
};

test gettest;

int main()
{
    test t1;
    test t2;
    printf("gettest count = %d\n",gettest.getCount());
    printf("t1 count = %d\n",t1.getCount());
    printf("t2 count = %d\n",t2.getCount());

    return 0;
}

```

我用全局变量来计算对象创建次数

在函数里面修改成全局变量

这样输出符合了我们的要求，但是我们要求函数里面不能用全局变量来计算对象创建次数

因为不允许用全局变量来计算对象创建次数，所以我们要用静态成员变量

```

#include<stdio.h>

class test
{
    private:
        static int Count;
    public:
        test()
        {
            Count++;
        }
        ~test()
        {
            --Count;
        }
        int getCount()
        {
            return Count;
        }
};

test gettest;

int main()
{
    test t1;
    test t2;
    printf("gettest count = %d\n",gettest.getCount());
    printf("t1 count = %d\n",t1.getCount());
    printf("t2 count = %d\n",t2.getCount());

    return 0;
}

```

给变量加了 static 就成了静态成员变量

静态成员变量就算对象执行结束，变量还是在内存空间的不会被释放，因为静态成员变量属于这个类，而不属于某个对象，所以这种变量就可以用来计算对象创建次数

但是编译报错，因为静态成员变量除了在类里面加 static 以外，还要在类外面指定静态成员变量存储在内存某个区域

```

1 #include<stdio.h>
2
3 class test
4 {
5     private:
6         static int Count;
7     public:
8         test()
9         {
10             Count++;
11         }
12         ~test()
13         {
14             --Count;
15         }
16         int getCount()
17         {
18             return Count;
19         }
20 };
21 int test::Count = 0;
22 test gettest;
23
24 int main()
25 {
26     test t1;
27     test t2;
28     printf("gettest count = %d\n",gettest.getCount());
29     printf("t1 count = %d\n",t1.getCount());
30     printf("t2 count = %d\n",t2.getCount());
31
32     return 0;
33 }

```

这就是指定类里面的静态成员变量放在内存的全局区。格式=(数据类型 类名 :: static 变量 = 初始值)

你看全局变量计算对象创建次数的问题，用静态成员变量就解决了。我们学 C 语言的时候知道全局变量任何一个 C 文件都可以调用，所以为了解决全局变量安全性问题，我们 C++ 用静态成员变量当做全局变量来使用。但是记住静态成员变量只属于同一个类多个对象全局

```

root@ubuntu:/home/xiang/C++/quanju# ./quanju
gettest count = 3
t1 count = 3
t2 count = 3
root@ubuntu:/home/xiang/C++/quanju#

```

这里还有一个缺陷因为类里面的静态成员变量是 private 属性的。所以必须创建对象，然后对象.getCount()这种方式才能访问，太过麻烦

如果不创建对象，就访问不了类变量

```

1 #include<stdio.h>
2
3 class test
4 {
5     public: static int Count;
6     public: test()
7     {
8         Count++;
9     }
10    ~test()
11    {
12        --Count;
13    }
14    int getCount()
15    {
16        return Count;
17    }
18 };
19
20 int test::Count = 0;
21
22 int main()
23 {
24     printf("count = %d\n",test::Count);
25
26     return 0;
27 }
28
29 int main()
30 {
31     test t1;
32     test t2;
33     test t3;
34     printf("count = %d\n",test::Count);
35
36     return 0;
37 }

```

我将 private 改成 public, 这样我直接用类名就可以访问里面的静态成员变量

因为我的类成员变量是 public 的, 所以我直接用类名加 :: 就可以访问变量

```

root@ubuntu:/home/xiang/C++/quanju# ./quanju
count = 0
root@ubuntu:/home/xiang/C++/quanju#

```

```

root@ubuntu:/home/xiang/C++/quanju# ./quanju
count = 3
root@ubuntu:/home/xiang/C++/quanju#

```

你看我就可以计算我用这个类创建了多少个对象。

这个 :: 符号已经发挥了三次不同的作用

```

using namespace first;
using second::internal::p;

```

获取全局变量的值

using 定义的全局变量关键字要用这个 :: 访问

```

3 class A
4 {
5     private:
6         int i;
7     public:
8         void print();
9 };
10
11 void A::print()
12 {
13     printf("print...\n");
14 }
15
16 int main()
17 {
18     A a;
19     a.print();
20     return 0;
21 }

```

在类外面对类里面声明的函数进行实现

在类里面声明的函数, 但是没有实现, 在类外面实现类里面定义的函数, 要用 :: 来做

```

2
3 class A
4 {
5     public:
6         static int i;
7     public:
8         static void print()
9         {
10             printf("i = %d\n",i);
11         }
12 };
13 int A::i;
14
15 int main()
16 {
17     A::i=10;
18     A::print();
19     return 0;
20 }

```

直接用类名调用类
里面的静态变量或
者静态函数

直接用类名访问类里面的变量，而不是用对象去访问类里面的变量，所以也需要用 :: 来做，但是类里面必须是 static 静态变量或者静态函数

static 修饰静态成员函数

```

class test
{
    private:
        int i;
    public:
        void seti(int a)
        {
            i = a;
        }
        int geti()
        {
            return i;
        }
};

int main()
{
    test t1;
    t1.seti(10);
    printf("i = %d\n",t1.geti());
    return 0;
}

```

这是我们用函数来给类里面的
变量赋值是合法的

```

root@ubuntu:/home
root@ubuntu:/home
i = 10

```

```
3 class test
4 {
5     private: int i;
6     public:
7         static void seti(int a)
8         {
9             i = a;
10        }
11        int geti()
12        {
13            return i;
14        }
15    };
16
17 int main()
18 {
19     test t1;
20     t1.seti(10);
21     printf("i = %d\n",t1.geti());
22
23     return 0;
24 }
```

如果我们将函数设置为静态 static 类型的，那么该静态函数是无法给类里面私有变量 i 赋值的，而且静态成员函数也无法调用类里面没有加 static 的函数

```
root@ubuntu:/home/xiang/C++/quanju# g++ -o quanju quanju.cpp
quanju.cpp: In static member function ‘static void test::seti(int)’:
quanju.cpp:10:4: error: invalid use of member ‘test::i’ in static member function
    i = a;
    ^
quanju.cpp:6:7: note: declared here
    int i;
    ^
root@ubuntu:/home/xiang/C++/quanju#
```

```
3 class test
4 {
5     private:
6         static int i;
7     public:
8         static void seti(int a)
9         {
10            i = a;
11        }
12        int geti()
13        {
14            return i;
15        }
16    };
17 int test::i = 0;
18
19 int main()
20 {
21     test t1;
22     t1.seti(10);
23     printf("i = %d\n",t1.geti());
24
25     return 0;
26 }
```

如果你实在想用静态函数去访问变量，那么你给变量也定义成静态的就可以了

前面(48页)我们类里面定义静态变量讲了的，在类里面定义了静态变量，在类外部要声明

```
root@ubuntu:/h
i = 10
root@ubuntu:/h
```

```

class test
{
    private:
        static int i;
    public:
        static void seti(int a)
        {
            i = a;
        }
        int geti()
        {
            return i;
        }
};
int test::i = 0;

int main()
{
    test::seti(10);
    test t1;
    printf("i = %d\n",t1.geti());

    return 0;
}

```

你也可以用类名加 :: 符号去访问类里面静态函数，和前面访问类里面静态变量一样

我们说过的 static 静态变量创建后就在内存中一直存在，不管你对对象消除不消除，所以这里取的值是类名赋的值

```

3 class test
4 {
5     private:
6         static int count;
7     public:
8         test()
9         {
10             count++;
11         }
12         ~test()
13         {
14             --count;
15         }
16         static int getcount()
17         {
18             return count;
19         }
20 };
21 int test::count = 0;
22
23 int main()
24 {
25     test t1;
26     test t2;
27     test t3;
28     printf("i = %d\n",test::getcount());
29
30     return 0;
31 }

```

我们用类里面静态函数取获取类里面静态变量的值，这样变量就很安全

root@ubuntu:/home/xian
root@ubuntu:/home/xian
i = 3
root@ubuntu:/home/xian

二阶构造模式(设计模式的一种), 记住并不是什么新语法新关键字。

设计模式只是设计程序的一种方法, 就是所谓的偏方。

```
class test
{
    int mi;
    int mj;
public:
    test(int i,int j)
    {
        mi = i;
        return;
        mj = j;
    }
    int geti()
    {
        return mi;
    }
    int getj()
    {
        return mj;
    }
};

int main()
{
    test t1(10,20);
    printf("mi = %d\n",t1.geti());
    printf("mj = %d\n",t1.getj());
}
```

在我执行构造函数的时候, 突然被什么东西打断了, 导致构造函数没有完全执行完, 就会出现程序 BUG, 这里 return 只是打个比方

一般在多线程程序结构里面会出现这种突然被打断的问题

最后出现 BUG

```
root@ubuntu:/home/xiang/C++/twoclass# ./twoclass
mi = 10
mj = 32765
```

所以我们要用一个方法来确定构造函数执行完之后, 是不是把构造函数里面的程序都完全执行了一遍。

二阶构造函数模型

第1阶构造函数初始化常规变量, 常规变量都是些 int 啊 char 啊, 这种东西一般不会出错

第2阶构造函数初始化, socket 或者文件局部, 或者很长的内存分配, 返回初始化是否成功

```
#include<stdio.h>
class twoclass
{
private:
    twoclass() //第一阶执行构造函数
    {
    }
    bool construct()//第二阶执行构造函数
    {
        return true;
    }
public:
    static twoclass *newtwoclass()
    {
        twoclass *ret = new twoclass();
        if(!(ret && ret->construct()))
        {
            delete ret;
            ret = NULL;
        }
        return ret;
    }
};
int main()
{
    twoclass *obj = twoclass::newtwoclass();
    return 0;
}
```

既然是二阶构造函数, 当然主程序 new 新对象, 就不能像以前那样直接 new 了, 要调用类函数 new

```
#include<stdio.h>

class twoclass
{
private:
    twoclass() //第一阶执行构造函数
    {
    }
    bool construct()//第二阶执行构造函数
    {
        return true;
    }
public:
    static twoclass *newtwoclass()
    {
        twoclass *ret = new twoclass();
        if(!(ret && ret->construct()))
        {
            delete ret;
            ret = NULL;
        }
        return ret;
    }

int main()
{
    twoclass *obj = twoclass::newtwoclass();
    printf("obj addr = %p\n",obj);
    return 0;
}
```

```
twoclass.cpp
root@ubuntu:/home/xiang/C++/twoclass# ./twoclass
obj addr = 0x1834c20
root@ubuntu:/home/xiang/C++/twoclass#
```

```
#include<stdio.h>

class twoclass
{
private:
    twoclass() //第一阶执行构造函数
    {
    }
    bool construct()//第二阶执行构造函数
    {
        return false;
    }
public:
    static twoclass *newtwoclass()
    {
        twoclass *ret = new twoclass();
        if(!(ret && ret->construct()))
        {
            delete ret;
            ret = NULL;
        }
        return ret;
    }

int main()
{
    twoclass *obj = twoclass::newtwoclass();
    printf("obj addr = %p\n",obj);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/twoclass# ./twoclass
obj addr = (nil)
root@ubuntu:/home/xiang/C++/twoclass#
```

其实就是把构造函数分层了两部分，第一阶段为构造函数，第二阶段为自己创建的函数。

```
#include<stdio.h>

class twoclass
{
    int i;
    char c;
    short s;
    long l;
private:
    twoclass(int i,char c,short s,long l) //第一阶执行构造函数
    {
        this->i = i;
        this->c = c;
        this->s = s;
        this->l = l;
    }
    bool construct()//第二阶执行构造函数
    {
        /* int[] = new int[10000000]*/
        /*open file*/
        /*socket*/
        return true;
    }
public:
    static twoclass *newtwoclass(int i,char c,short s,long l)
    {
        twoclass *ret = new twoclass(i,c,s,l);
        if(!(ret && ret->construct()))
        {
            delete ret;
            ret = NULL;
        }
        return ret;
    }

int main()
{
    twoclass *obj = twoclass::newtwoclass(10,'c',50,100);
    printf("obj addr = %p\n",obj);
    return 0;
}
```

将一些常用的变量赋值，不容易出错的代码放在构造函数里面

将一些内存分配，
打开网络初始化，
复杂的类初始化放
在自己定义的函数
里面初始化

这就是二阶构造函数的
用途，初始化出现错误就
返回不成功，让程序员好
处理

二阶构造函数实际应用

我在用 QT 做图形用户界面就用到了二阶构造函数

```
class QcalculatorUI : public QWidget
{
private:
    QLineEdit *l_edit;
    QPushButton* button[20];
    QcalculatorUI();
    bool construct(QWidget* w);
public:
    static QcalculatorUI* Newinstance(QWidget* w);
    ~QcalculatorUI();
};

#endif // QCALCULATORUI_H

QcalculatorUI::~QcalculatorUI()
{
}

QcalculatorUI *QcalculatorUI::Newinstance(QWidget* w)
{
    QcalculatorUI* ret = new QcalculatorUI();
    if((ret == NULL) || !ret->construct(w))
    {
        delete ret;
        ret = NULL;
    }

    return ret;
}
bool QcalculatorUI::construct(QWidget* w)
{
    bool ret = true;
    const char *text[20] =
    {
        "7", "8", "9", "+", "(",
        "4", "5", "6", "-", ")",
        "1", "2", "3", "*", "<-",
        "0", ".", "=" , "/", "c",
    };

    l_edit = new QLineEdit(w);
    if(l_edit != NULL)
    {
        l_edit->move(10, 10);
        l_edit->resize(240, 30);
        l_edit->setReadOnly(true);
    }
}
```

二阶构造函数就是把构造函数放在类私有的属性里面,进行第1次构造

第2次构造就创建一个成员函数来解决

我主函数调用类里面的Newinstance成员函数来启动第1阶构造函数和第2阶成员函数

第1阶就是类的构造函数,我们前面说了只是用来处理一些简单的变量赋值,但是这个第2阶成员函数可是用来处理一些大型数据初始化的,所以要有 bool 返回值

```

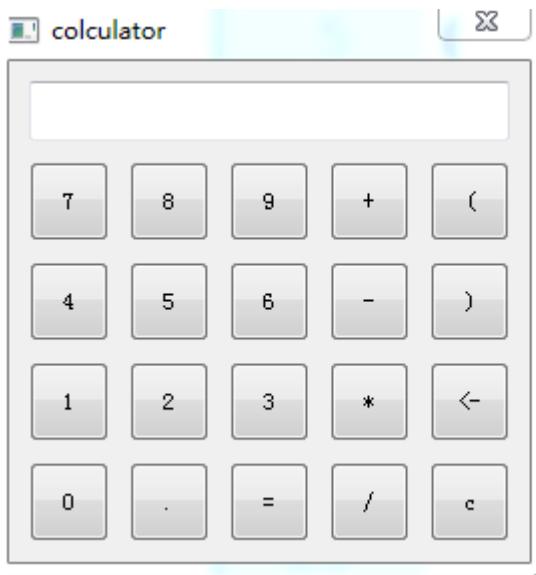
#include <QGuiApplication>
#include <QWidget>
#include "QcalculatorUI.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget* w = new QWidget(NULL, Qt::WindowCloseButtonHint);

    QcalculatorUI *cal = QcalculatorUI::Newinstance(w);
    if(cal != NULL)
    {
        }

    w->show();
    w->setFixedSize(w->width(),w->height());
    int ret = a.exec();
    delete w;
    return ret;
}

```

主函数只需要去调用
Newinstance 函数，然后
Newinstance 去帮主函数
执行类构造函数和第 2 阶
成员函数，执行成功返回对
象地址给 cal，你主函数使
用 cal 就行了



QApp
QWid

Qcal
if(c
{
}

一个无聊的计算器程序就实现了

友元这个东西很扯，知道就行了

```
1 #include<stdio.h>
2
3 class point
4 {
5     double x;
6     double y;
7     friend void func(point& p);
8 };
9
10 void func(point& p)
11 {
12
13 }
14
15 int main()
16 {
17     point p1;
18     point p2;
19     p1.x = 10;
20     p1.y = 20;
21
22     return 0;
23 }
```

编译不过的原因是 C++ 类里面的变量默认是 private 的，所以外部无法直接调用

```
root@ubuntu:/home/xiang/C++/youyuan# g++ -c youyuan.cpp
youyuan.cpp: In function 'int main()':
youyuan.cpp:5:9: error: 'double point::x' is private
    double x;
           ^
youyuan.cpp:19:5: error: within this context
    p1.x = 10;
           ^
youyuan.cpp:6:9: error: 'double point::y' is private
    double y;
           ^
youyuan.cpp:20:5: error: within this context
    p1.y = 20;
           ^
root@ubuntu:/home/xiang/C++/youyuan#
```

```
#include<stdio.h>

class point
{
    double x;
    double y;
public:
    point(double x,double y)
    {
        this->x = x;
        this->y = y;
    }
    int getx()
    {
        return x;
    }
    int gety()
    {
        return y;
    }
};

void func(point& p)
{
}

int main()
{
    point p1(10,100);
    point p2(20,200);

    printf("p1_x = %d p1_y = %d\n",p1.getx(),p1.gety());
    printf("p2_x = %d p2_y = %d\n",p2.getx(),p2.gety());
    return 0;
}
```

我增加 public 函数来操作类里面的变量

这样没有问题

```
root@ubuntu:/home/xiang/
root@ubuntu:/home/xiang/
p1_x = 10 p1_y = 100
p2_x = 20 p2_y = 200
root@ubuntu:/home/xiang/
```

```

1 #include<stdio.h>
2
3 class point
4 {
5     double x;
6     double y;
7 public:
8     point(double x,double y)
9     {
10         this->x = x;
11         this->y = y;
12     }
13     int getx()
14     {
15         return x;
16     }
17     int gety()
18     {
19         return y;
20     }
21 };
22
23 double func(point& a,point& b)
24 {
25     double funcx;
26     double funcy;
27     funcx = a.x - b.x;
28     funcy = a.y - b.y;
29     return funcx+funcy;
30 }
31
32 int main()
33 {
34     point p1(10,100);
35     point p2(20,200);
36
37     double d = func(p1,p2);
38     printf("d = %d\n",d);
39
40     return 0;
41 }

```

这样编译是会报错的, x 和 y 是类私有的, 外部函数虽然可以在形参上传入 p1, p2 对象, 但是无法直接使用类里面的 private 变量

```

root@ubuntu:/home/xiang/C++/youyuan# g++ -o youyuan youyuan.cpp
youyuan.cpp: In function ‘double func(point&, point&)’:
youyuan.cpp:5:9: error: ‘double point::x’ is private
    double x;
          ^

```

```

double func(point& a,point& b)
{
    double funcx;
    double funcy;
    funcx = a.getx() - b.getx();
    funcy = a.gety() - b.gety();
    return funcx+funcy;
}

int main()
{
    point p1(100,10);
    point p2(200,20);

    double d = func(p1,p2);
    printf("d = %f\n",d);

    return 0;
}

```

这样用类函数间接取值是没有问题的, 但是感觉好麻烦, 我 C 语言用结构体对象都可以 . 点结构体里面的成员。C++的 class 类反而让成员权限搞得复杂了

```

root@ubuntu:/home/xiang/C++/youyuan# ./youyuan
d = -110.000000
root@ubuntu:/home/xiang/C++/youyuan#

```

```

#include<stdio.h>

class point
{
    double x;
    double y;
public:
    point(double x,double y)
    {
        this->x = x;
        this->y = y;
    }
    int getx()
    {
        return x;
    }
    int gety()
    {
        return y;
    }
    friend double func(point& a,point& b);
};

double func(point& a,point& b)
{
    double funcx;
    double funcy;
    funcx = a.x - b.x;
    funcy = a.y - b.y;
    return funcx+funcy;
}

int main()
{
    point p1(100,10);
    point p2(200,20);

    double d = func(p1,p2);
    printf("d = %f\n",d);

    return 0;
}

```

我用 friend 关键字让 func 函数在类里面声明，这样 func 函数虽然是在类外面实现的，但是 func 已经关联成类里面的函数了

然后我直接在 func 函数里面 . 点 class 成员变量是没有问题的，因为 func 函数经过友元已经和 class 关联上了

```

root@ubuntu:/home/x
d = -110.000000
root@ubuntu:/home/x

```

```

3 class A
4 {
5     private:
6         int i;
7     public:
8         void print();
9 };
10 void A::print()
11 {
12     printf("print...\n");
13 }

```

但是我觉得友元没有声明是属于哪个类的函数实现，你看我以前这个，用 :: 符号还声明该函数属于哪个类

所以友元声明的函数，其实是没有封装性的，你不知道你类外部实现的函数属于哪个类，既没有::符号，也没有 C 语言结构体 . 点来指定。所以现代 C++ 开发中基本不使用友元

类里面函数重载

```
file Edit View Search Terminal Help
1 #include<stdio.h>
2 void func()
3 {
4     printf("func....\n");
5 }
6
7 void func(int i)
8 {
9     printf("func i = %d\n",i);
10}
11
12 class test
13 {
14 private:
15     int i;
16 public:
17     test()
18     {
19         printf("test...\n");
20     }
21     test(int i)
22     {
23         this->i = i;
24         printf("test i = %d\n",this->i);
25     }
26     static void func()
27     {
28         printf("class::func..\n");
29     }
30     void func(int i)
31     {
32         printf("class::func i =%d \n",i);
33     }
34 }
35
36
37 };
38
39
40 int main()
41 {
42     test t1;
43     test t2(10);
44     t1.func();
45     t1.func(20);
46     func();
47     func(100);
```

这是前面讲的在 main 外进行的全局函数重载

这是我在类里面定义的函数重载

类里面也定义了和全局函数一样的函数名

最后发现类里面的 func 函数和全局的 func 函数并不冲突

```
root@ubuntu:/home/xiang/C++/func# g++
root@ubuntu:/home/xiang/C++/func# ./f
test...
test i = 10
class::func..
class::func i =20
func....
func i = 100
root@ubuntu:/home/xiang/C++/func#
```

输出结果正常， func 函数各输出各的

this 指针的用法

```
root@ubuntu:/home/xiang/C++/this# ./this
main addr = 435c2fc7
addr = 435c2fc7
```

在类里面写不打 * 号的 this，就是获取当前类创建对象的地址

和获取对象本身的地址是一样的，只是一个是在类里面获取对象的地址，一个是在外部获取对象的地址

```
#include<stdio.h>

class test
{
public:
    test& func()
    {
        printf("addr = %x\n",this);
        return *this;
    }
};

int main()
{
    test t1;
    printf("main addr = %x\n",&t1);
    t1.func();
    return 0;
}
```

```
root@ubuntu:/home/xiang/C++/this# ./this
main addr = 435c2fc7
addr = 435c2fc7
```

类里面获取和类外部获取对象地址一样

那么这样 this 有什么用呢？

1.this 可以让类里面的同名变量不冲突

```
class test
{
    int x;
public:
    void func(int x)
    {
        this->x = x;
    }
    void setfunc()
    {
        printf("this->x = %d\n",this->x);
    }
};

int main()
{
    test t1;
    t1.func(10);
    t1.setfunc();
    return 0;
}
```

this-> 就表示操作的是对象类里面的变量，所以外部的形参 x 和类内部定义的私有属性 x 虽然同名，但是不冲突，因为它们是在两个内存空间

```
root@ubuntu:/home/xiang/C++/this# ./this
this->x = 10
root@ubuntu:/home/xiang/C++/this#
```

输出的是类内部 x 得到的值

2.this可以用作类的返回值，然外部可以连续调用该类

```
#include<stdio.h>

class test
{
public:
    void func()
    {
        printf("this ...\\n");
    }
};

int main()
{
    test t1;
    t1.func();
    return 0;
}
```

这是我们以前传统的做法

```
root@ubuntu:/home/xiang/C++/this# ./this
this ...
```

```
#include<stdio.h>

class test
{
public:
    void func()
    {
        printf("this ...\\n");
    }
};

int main()
{
    test t1;
    t1.func().func();
    return 0;
}
```

如果我不想再用 t1
来调用 func,我想直
接调用多次 func

编译就出现了报错

```
root@ubuntu:/home/xiang/C++/this# g++ -o this this.cpp
this.cpp: In function int main()':
this.cpp:15:11: error: invalid use of 'void'
    t1.func().func();
           ^~~~~~
...@ubuntu:/home/xiang/C++/this#
```

```
1 #include<stdio.h>
2
3 class test
4 {
5     public:
6         test& func()
7         {
8             printf("this ...\\n");
9             return *this;
10        }
11 };
12
13 int main()
14 {
15     test t1;
16     t1.func().func();
17     return 0;
18 }
```

我可以在函数返回值这
里定义一个引用

除了返回对象地址，我
还用*号获取对象的值

我在对对象值的基础上再
调用对象里面的函数是
没有问题的

```
root@ubuntu:/home/xiang/C++/this# ./this
this ...
this ...
root@ubuntu:/home/xiang/C++/this#
```

C++除了函数可以重载，+ - * /操作符也可以重载

$x_1 = 1 \quad y_1 = 2$
 $x_2 = 3 \quad y_2 = 4$
求 $x_1 + x_2 = ?$ 求 $y_1 + y_2 = ?$

我们将 x_1, y_1 写入一个对象， y_1, y_2 写入一个对象

```
#include<stdio.h>

class Complex
{
public:
    int a;
    int b;
};

Complex add(const Complex& p1,const Complex &p2)
{
    Complex ret;
    ret.a = p1.a + p2.a; // 然后我们用 x1 + x2
    ret.b = p1.b + p2.b; // 然后我们用 y1 + y2
    return ret;
}

int main()
{
    Complex c1 = {1,2}; // 将 x1, y1 初始值写入对象
    Complex c2 = {3,4}; // 将 x2, y2 初始值写入对象
    Complex c3 = add(c1,c2);
    printf("c3.a = %d\n",c3.a); // 求出 x1+x2 的和, y1+y2 的和
    printf("c3.b = %d\n",c3.b);
    return 0;
}
```

root@ubuntu:/home
c3.a = 4
c3.b = 6
root@ubuntu:/home

```
int main()
{
    Complex c1 = {1,2};
    Complex c2 = {3,4};
    Complex c3 = add(c1,c2);
    printf("c3.a = %d\n",c3.a);
    printf("c3.b = %d\n",c3.b);
    return 0;
}
```

下面我们将对两个对象相加的函数进行改进。

代码这样写没有错,只是看起来不直观,两个对象相加还必须用函数名来表示

```

#include<stdio.h>

class Complex
{
public:
    int a;
    int b;
};

Complex operator + (const Complex& p1,const Complex &p2)
{
    Complex ret;
    ret.a = p1.a + p2.a;
    ret.b = p1.b + p2.b;
    return ret;
}

int main()
{
    Complex c1 = {1,2};
    Complex c2 = {3,4};
    Complex c3 = operator + (c1,c2);
    printf("c3.a = %d\n",c3.a);
    printf("c3.b = %d\n",c3.b);
    return 0;
}

```

我们定义一个全局函数，关键字用 operator，这样我这个函数的 add 就可以取消了

这样调用也就是调用 operator + 这个全局函数

但是这样看起还是很别扭，你虽然把 add 函数名取消了，你加个 operator + 和 add 有什么区别呢？

```

root@ubuntu:/home
root@ubuntu:/home
c3.a = 4
c3.b = 6
root@ubuntu:/home

```

```

#include<stdio.h>

class Complex
{
public:
    int a;
    int b;
};

Complex operator + (const Complex& p1,const Complex &p2)
{
    Complex ret;
    ret.a = p1.a + p2.a;
    ret.b = p1.b + p2.b;
    return ret;
}

int main()
{
    Complex c1 = {1,2};
    Complex c2 = {3,4};
    Complex c3 = c1 + c2;
    printf("c3.a = %d\n",c3.a);
    printf("c3.b = %d\n",c3.b);
    return 0;
}

```

operator 就是让函数形参的两个对象有相加的功能

其实你什么都不用改，你把两个对象相加写成 c1+c2 就对了

```

root@ubuntu:/home
root@ubuntu:/home
c3.a = 4
c3.b = 6
root@ubuntu:/home

```

以上代码不符合面向对象的思想，类里面变量可以外部访问，类对象使用的函数不在类里面

```

1 #include<stdio.h>
2
3 class Complex
4 {
5     private:
6         int a;
7         int b;
8     public:
9         Complex(int a = 0, int b = 0)
10        {
11            this->a = a;
12            this->b = b;
13        }
14        Complex operator + (const Complex &p2)
15        {
16            Complex ret;
17            ret.a = this->a + p2.a;
18            ret.b = this->b + p2.b;
19            return ret;
20        }
21    };
22
23
24
25 int main()
26 {
27     Complex c1(1,2);
28     Complex c2(3,4);
29     Complex c3 = c1.operator + (c2);
30 //     printf("c3.a = %d\n",c3.a);
31 //     printf("c3.b = %d\n",c3.b);
32     return 0;
33 }

```

这样的代码编译出来和运行是没有问题的。为了更直观我加两个
变量获取函数

```

#include<stdio.h>

class Complex
{
private:
    int a;
    int b;
public:
Complex(int a = 0,int b = 0)
{
    this->a = a;
    this->b = b;
}
Complex operator + (const Complex &p2)
{
    Complex ret;
    ret.a = this->a + p2.a;
    ret.b = this->b + p2.b;
    return ret;
}
int geta(Complex &geta)
{
    return geta.a;
}
int getb(Complex &getb)
{
    return getb.b;
}
};

int main()
{
    Complex c1(1,2);
    Complex c2(3,4);
    Complex c3 = c1 + c2;
    printf("c3.a = %d\n",c3.geta(c3));
    printf("c3.b = %d\n",c3.getb(c3));
    return 0;
}

```

所以 c1.operator + (c2) 可以直接改成 C1 +C2

编译通过输出一切正常

我们下面用复数类来完善操作符重载，就当练习

```

1 #include<stdio.h>
2 #include<math.h>
3 class Complex
4 {
5     double a;
6     double b;
7     public:
8     Complex(double a,double b)
9     {
10         this->a = a;
11         this->b = b;
12     }
13     Complex operator + (Complex& p)
14     {
15         double na = a + p.a;
16         double nb = b + p.b;
17         Complex ret(na,nb);
18         return ret;
19     }
20     Complex operator - (Complex& p)
21     {
22         double na = a - p.a;
23         double nb = b - p.b;
24         Complex ret(na,nb);
25         return ret;
26     }
27     Complex operator * (Complex& p)
28     {
29         double na = a * p.a - b * p.b;
30         double nb = a * p.b + b * p.a;
31         Complex ret(na,nb);
32         return ret;
33     }
34     Complex operator / (Complex& p)
35     {
36         double cm = p.a * p.a + p.b * p.b;
37         double na = (a * p.a + b * p.b) / cm;
38         double nb = (b * p.a - a * p.b) / cm;
39     }
40     bool operator == (Complex& p)
41     {
42         return (a == p.a) && (b == p.b);
43     }
44     bool operator != (Complex& p)
45     {
46         return !(*this == p);
47     }
48     Complex& operator = (Complex& p)
49     {
50         if( this != &p )
51         {
52             a = p.a;
53             b = p.b;
54         }
55         return *this;
56     }
57     double getA()
58     {
59         return a;
60     }
61     double getB()
62     {
63         return b;
64     }
65 };

```

```

int main()
{
    Complex c1(1,2);
    Complex c2(3,6);

    Complex c3 = c2 + c1;
    printf("c3.a = %f  c3.b = %f\n",c3.getA(),c3.getB());
    Complex c4 = c2 - c1;
    printf("c4.a = %f  c4.b = %f\n",c4.getA(),c4.getB());
    Complex c5 = c2 / c1;
    printf("c5.a = %f  c5.b = %f\n",c5.getA(),c5.getB());
    Complex c6 = c2 * c1;
    printf("c6.a = %f  c6.b = %f\n",c6.getA(),c6.getB());

    Complex c7(2,3);
    printf("c7 == c1 : %d\n",c7 == c1);
    printf("c7 == c1 : %d\n",c7 != c1);
    Complex c8(1,2);
    printf("c8 == c1 : %d\n",c8 == c1);
    printf("c8 == c1 : %d\n",c8 != c1);

    printf("c3.a = %f  c3.b = %f\n",c3.getA(),c3.getB());
    c3 = c1;
    printf("c3.a = %f  c3.b = %f\n",c3.getA(),c3.getB());
    return 0;
}

```

```

root@ubuntu:/home/Xiang/C++/symbolheavy# g++ -o symbolheavy
root@ubuntu:/home/xiang/C++/symbolheavy# ./symbolheavyall
c3.a = 4.000000  c3.b = 8.000000
c4.a = 2.000000  c4.b = 4.000000
c5.a = 0.000000  c5.b = 0.000000
c5.a = -9.000000  c5.b = 12.000000
c7 == c1 : 0
c7 == c1 : 1
c8 == c1 : 1
c8 == c1 : 0
c3.a = 4.000000  c3.b = 8.000000
c3.a = 1.000000  c3.b = 2.000000
root@ubuntu:/home/xiang/C++/symbolheavy# █

```

STL 模板库与 << 操作符重载

```
1 #include<stdio.h>
2
3 class console
4 {
5     public:
6         void operator << (int i)
7         {
8             printf("%d",i);
9         }
10        void operator << (char c)
11        {
12            printf("%c",c);
13        }
14 };
15
16
17 int main(void)
18 {
19
20     console cout;
21     cout << 1;
22     cout << '\n';
23     return 0;
24 }
```

移位运算符也是可以重载的

```
root@ubuntu:/h
1
root@ubuntu:/h
```

写成两行太麻烦了，能不能写成一行。当然可以，用 this 指针

```
1 #include<stdio.h>
2
3 class console
4 {
5     public:
6         console& operator << (int i)
7         {
8             printf("%d",i);
9             return *this;//返回对象值本身
10        }
11        console& operator << (char c)
12        {
13            printf("%c",c);
14            return *this;//返回对象值本身
15        }
16 };
17
18
19 int main(void)
20 {
21
22     console cout;
23     cout << 1 << '\n';
24     return 0;
25 }
```

类里面增加返回对象别名，这里就可以写成一行了，相当于先把 1 写给 i，然后在对象函数执行完的时候返回*this 对象自己，然后再重复执行对象里面的其它函数

```
root@ubuntu:/h
1
root@ubuntu:/h
```

这个每次换行我都要\n，感觉不太舒服，能不能有舒服的方法换行？

```

#include<stdio.h>
const char endl = '\n';
class console
{
public:
    console& operator << (int i)
    {
        printf("%d",i);
        return *this;
    }
    console& operator << (char c)
    {
        printf("%c",c);
        return *this;
    }
};

int main(void)

{
    cout << 1<<endl;
    return 0;
}

#include<stdio.h>
const char endl = '\n';
class console
{
public:
    console& operator << (int i)
    {
        printf("%d",i);
        return *this;
    }
    console& operator << (char c)
    {
        printf("%c",c);
        return *this;
    }
    console& operator << (const char *s)
    {
        printf("%s",s);
        return *this;
    }
};

int main(void)
{
    cout << "XXXXZZZ" << endl;
    return 0;
}

#include<stdio.h>
const char endl = '\n';
class console
{
public:
    console& operator << (int i)
    {
        printf("%d",i);
        return *this;
    }
    console& operator << (char c)
    {
        printf("%c",c);
        return *this;
    }
    console& operator << (const char *s)
    {
        printf("%s",s);
        return *this;
    }
};

int main(void)
{
    int a = 10, b = 20;
    cout << "XXXXZZZ" << endl;
    cout << a + b << endl;
    return 0;
}

```

用变量代替\n,
endl 敲着舒服些

```

root@ubuntu:/h
1
root@ubuntu:/h

```

传入形参的值如果是固定的，可以在前面加 const，这样 C++ 才不会产生警告，也可以看出来是输入型参数

```

root@ubuntu:/home/xiang/C++/STL# ./stl3
XXXXZZZ
root@ubuntu:/home/xiang/C++/STL#

```

也可以让两个整形相加后的值传入对象

```

root@ubuntu:/h
XXXXZZZ
30
root@ubuntu:/h

```

这就是变量运算先运行，然后再传值

这些方法 C++ 语言的 STL 库就已经帮我们实现好了，不需要我们做这些。

STL 模板库使用

```
#include<iostream>
using namespace std;

int main(void)
{
    cout << "xxxxxx" << endl;

    int a;
    cout<<"input a"<<endl;
    cin >> a;//类似c语言scanf，接受键盘数据给变量a
    cout<< a << endl;
    return 0;
}

root@ubuntu:/
xxxxxx
input a
10
10
```

输出正常

字符串大小排序，和字符串拼接

```
1 #include<iostream>
2 #include<string>
3
4 using namespace std;
5
6 void string_sort(string a[],int len)
7 {
8     for(int i=0; i<len; i++)
9     {
10         for(int j=i; j<len; j++)
11         {
12             if(a[i] > a[j])//为什么两个string对象能比较大小，这是因为string类里面实现了操作符重载
13             {
14                 swap(a[i],a[j]);//两个字符串交换函数swap
15
16             }
17         }
18     }
19 }
20
21 string string_add(string a[],int len) //多组字符串拼接成一串字符串
22 {
23     string ret = "";
24     for(int i=0; i<len; i++)
25     {
26         ret += a[i] + ";" //用分号将各组字符串隔离开 /*ret = ret + a[i] + ";" 也是操作符重载*/
27     }
28     return ret;
29 }

30
31 int main()
32 {
33     string sa[7] =
34     {
35         "Gsoftware",
36         "Fsoftxzz",
37         "Exzzzzz",
38         "Dxxxx",
39         "Ccccc",
40         "Bbccc",
41         "AAAAA"
42     };
43     string_sort(sa,7);//按照ABCD大小顺序来排序
44     for(int i=0; i<7; i++)
45     {
46         cout << sa[i] << endl;
47     }

48     cout << "-----" << endl;
49     cout << string_add(sa,7) << endl; //拼接字符串
50
51     return 0;
52 }
```

```
root@ubuntu:~/home/xiang/C++/STL# g++ -o paixu paixu.cpp
root@ubuntu:~/home/xiang/C++/STL# ./paixu
AAAAA
Bbccc
Ccccc
Dxxxx
Exzzzzz
Fsoftxzz
Gsoftware
-----
AAAAA;Bbccc;Ccccc;Dxxxx;Exzzzzz;Fsoftxzz;Gsoftware;
root@ubuntu:~/home/xiang/C++/STL#
```

字符串转换成数字

```
#include<iostream>
#include<sstream> //我们用到字符串流，必须包含sstream头文件
using namespace std;

int main(void)
{
    istringstream iss("123.45");
    double num;
    iss >> num;
    cout << num << endl;
    return 0;
}
```

定义一个 istringstream 的对象，该对象可以得到一个字符串，自动将其转换成数字

将转换后的数字写入 double 变量

```
root@ubuntu:/home
123.45
root@ubuntu:/home
```

字符串转换成数字是否成功是可以判断的

```
#include<iostream>
#include<sstream> //我们用到字符串流，必须包含sstream头文件
using namespace std;

int main(void)
{
    istringstream iss("123.45");
    double num;

    if(iss >> num)
    {
        cout << "true" << endl;
    }
    else
    {
        cout << "false" << endl;
    }
    cout << num << endl;
    return 0;
}
```

istringstream 对象 iss 转换后会返回一个状态，为 1 就是转换成功，为 0 就是转换失败

转换成功毫无悬念

```
root@ubuntu:/home/
true
123.45
```

为什么要判断转换是否成功呢？

```
#include<iostream>
#include<sstream> //我们用到字符串流，必须包含sstream头文件
using namespace std;

int main(void)
{
    istringstream iss("abcd");
    double num;

    if(iss >> num)
    {
        cout << "true" << endl;
    }
    else
    {
        cout << "false" << endl;
    }
    cout << num << endl;
    return 0;
}
```

因为有可能会手抖写成了字符，而不是数字型字符串，这样就转换失败

```
root@ubuntu:
false
0
```

```
int main(void)
{
    istringstream iss("12345");
    int num; //用 int 来承载字符串转换的数字也是可以的

    if(iss >> num)
    {
        cout << "true" << endl;
    }
    else
    {
        cout << "false" << endl;
    }
    cout << num << endl;
    return 0;
}
```

root@ubuntu:~/Documents\$ g++ test.cpp
root@ubuntu:/home/zheng\$./test
true
12345

数字转换成字符串

```
#include<iostream>
#include<sstream> //我们用到字符串流，必须包含sstream头文件
using namespace std;

int main(void)
{
    ostringstream oss; //用 ostringstream 创建的对象来完成数字转换成字符串
    oss << 543 << "." << 15;
    string s = oss.str();

    cout << s << endl;
    return 0;
}
```

root@ubuntu:~/Documents\$ g++ test.cpp
root@ubuntu:/home/zheng\$./test
543.15

```
#include<iostream>
#include<sstream> //我们用到字符串流，必须包含sstream头文件
using namespace std;

int main(void)
{
    ostringstream oss; //直接传入数字，或者变量也是可以的
    int num = 1234;
    oss << num;
    string s = oss.str();

    cout << s << endl;
    return 0;
}
```

root@ubuntu:~/Documents\$ g++ test.cpp
root@ubuntu:/home/zheng\$./test
1234

string 类定义的字符串长度计算

```
1 #include<iostream>
2
3 using namespace std;
4
5
6 int main()
7 {
8     string str = "xxxxzz";
9     cout << str.length() << endl;
10    cout << str.size() << endl;
11    return 0;
12 }
```

String 里面的
length 和 size 都是
返回字符串长度的

```
root@ubuntu:/
6
6
```

String 获取一整串字符串里面的某一段字符串

```
1 #include<iostream>
2
3 using namespace std;
4
5
6 int main()
7 {
8     string str = "abcdefghijkl";
9     cout << str.substr(5) << endl;
10    cout << str.substr(0,5) << endl;
11    cout << str.substr(0) << endl;
12    return 0;
13 }
```

substr 传入 5 是表示从字符串第 5 个字符之后
开始获取，就是从第 6 个字符开始获取到结束

传入 0,5 就表示从字符串第 0 个字符开始，一直
获取到第 5 个字符

```
root@ubuntu:/
fghijk
abcde
abcdefghijkl
```

字符串将右移的字符放在前面

比如将 abcdefg >>3 得到 efgabcd

```
1 #include<iostream>
2
3 using namespace std;
4
5 string char_moveright(string s , int n)
6 {
7
8     string ret = " ";
9     unsigned int pos = 0;
10
11     n = n % s.length();
12     pos = s.length() - n;
13     ret = s.substr(pos);
14     ret += s.substr(0,pos);
15
16     return ret;
17 }
18
19 int main()
20 {
21     string str = "abcdefg"; //abcdefg >>3 = efgabcd
22     string st = char_moveright(str,3);
23     cout << st << endl;
24     return 0;
25 }
```

```
root@ubuntu:
efgabcd
root@ubuntu:
```

```

1 #include<iostream>
2
3 using namespace std;
4
5 string operator >> (string s , int n)
6 {
7
8     string ret = " ";
9     unsigned int pos = 0;
0
1     n = n % s.length();
2     pos = s.length() - n;
3     ret = s.substr(pos);
4     ret += s.substr(0,pos);
5
6     return ret;
7 }
8
9 int main()
0 {
1     string str = "abcdefg"; //abcdefg >>3 = efgabcd
2     string st = str >> 3;
3     cout << st << endl;
4     return 0;
5 }

```

我觉得我记不住字符串右移的函数名，那么我们可以用操作符重载让你记住右移符号

```

root@ubuntu:
efgabcd
root@ubuntu:

```

C++的 string 对象支持单个字符访问，单个字符访问方式就是 C 语言数组下标

```

1 #include<stdio.h>
2 #include<iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     string s = "abcdef";
9
10    printf("%c\n",s[2]);
11    return 0;
12 }

```

可以用数组下标方式取字符串对象里面的单个字符

```

root@ubuntu:
c
root@ubuntu:

```

C++的 string 对象，判断对象里面有几个是数字

```

#include<stdio.h>
#include<iostream>

using namespace std;

int main()
{
    string s = "abcdef";
    string num = "ab8def";
    int a = isdigit(s[2]); //判断单个字符是否为数字，为数字返回1，否则返回0
    printf("a = %d\n",a);

    int b = isdigit(num[2]); //判断单个字符是否为数字
    printf("b = %d\n",b);
    return 0;
}

```

```

root@ubuntu:
a = 0
b = 1

```

C++的 string 对象就是用来取代 C 语言的 char * 数组指针的。

数组符号[] 用作操作符重载

```
1 #include<iostream>
2
3 using namespace std;
4
5 class test
6 {
7     public:
8         int operator [] (int i)
9         {
10             return 0;
11         }
12         int operator [] (const char *s)
13         {
14             return 0;
15         }
16     };
17
18 int main()
19 {
20     return 0;
21 }
```

[] 符号操作符重载就是这样定义的

这样定义后，如果我在 main 函数里定义一个 test 类的对象 t，我直接 t[10] 就相当于调用 [] (int i) 函数

如果我 t["xzz"] 就相当于调用 [] (const char *s) 函数

注意：不管你定义多少个 [] 符号重载函数，函数形参只能有一个

```
1 #include<iostream>
2 #include<stdio.h>
3 using namespace std;
4
5 class test
6 {
7     public:
8         int operator [] (int i)
9         {
10             printf("i = %d\n",i);
11         }
12         int operator [] (const char *s)
13         {
14             printf("%s\n",s);
15         }
16     };
17
18 int main()
19 {
20     test t;
21     t[10];
22     t["xzz"];
23
24     return 0;
25 }
```

```
root@ubuntu:/home
root@ubuntu:/home
i = 10
xzz
```

这样确实可以像操作数组一样去操作类里面的函数，但是有什么意义呢？

```

#include<iostream>

using namespace std;

class test
{
private:
    int a[5];
public:
    int operator [] (int i)
    {
        return a[i];
    }
    int operator [] (const char *s)
    {
        return 0;
    }
    int length()
    {
        return 5;
    }
};

int main()
{
    test t;
    for(int i=0; i<t.length(); i++) //给t对象里面的数组a[5]赋值
    {
        t[i] = i; //这是我们c语言常规给数组赋值的方法 ←
    }
    for(int i=0; i<t.length(); i++)
    {
        cout << t[i] << endl;
    }

    return 0;
}

```

比如我要想给对象里面的
一个数组赋值

我就不用对象去 点 . 什么函数,
然后给函数赋值, 这样很麻烦。我
就像给数组下标赋值那样给对象
里面的数组变量赋值, 你看多方
便

```

root@ubuntu:/home/xiang/C++/STL# g++ -o strarray5 strarray5.cpp
strarray5.cpp: In function ‘int main()’:
strarray5.cpp:30:8: error: lvalue required as left operand of assignment
    t[i] = i; //这是我们c语言常规给数组赋值的方法
    ^

```

但是为什么报错呢? 其实问题就是出现在这里

```

int main()
{
    test t;
    for(int i=0; i<t.length(); i++) //给t对象里面的数组a[5]赋值 ←
    {
        t.operator[](i) = i; //其实就是这样, 但是c/c++语言规定函数调用的返回值是不能用于 = 号左边的
    }
}

```

那怎么办呢?

```

class test
{
    private:
        int a[5];
    public:
        int& operator [] (int i)
        {
            return a[i];
        }
        int& operator [] (const char *s)
        {
            return a[0];
        }
        int length()
        {
            return 5;
        }
};

int main()
{
    test t;
    for(int i=0; i<t.length(); i++) //给t对象里面的数组a[5]赋值
    {
        t[i] = i;//t.operator[](i) = i;
    }
    for(int i=0; i<t.length(); i++)
    {
        cout << t[i] << endl;
    }

    return 0;
}

```

我为了解决函数返回值不能当左值，这里我们可以用引用返回，这样 a[i] 就变成了*a[i]返回，相当于指针变量

```

root@ubuntu:/home/xiang/C++/STL#
root@ubuntu:/home/xiang/C++/STL#
0
1
2
3
4

```

输出没有问题

其实 C++ 数组对象就是取代 C 语言里面的数组。

除了给对象里面数组赋值步骤简化以外，还有什么好处呢？

比如说我在初始化的时候给对象里面数组已经设置好了固定参数，然后我其它函数调用这个数组对象时，可以根据数组下标决定使用对象里面数组的哪个参数。

```

3 using namespace std;
4
5 class test
6 {
7     private:
8         int a[5];
9     public:
10        int& operator [] (int i)
11        {
12            return a[i];
13        }
14        int& operator [] (const char *s)
15        {
16            if(s == "abc")
17            {
18                return a[1];
19            }
20            else if(s == "xzz")
21            {
22                return a[2];
23            }
24            else
25            {
26                cout << "input error" << endl;
27            }
28        }
29        int length()
30        {
31            return 5;
32        }
33    };
34
35 int main()
36 {
37     test t;
38     for(int i=0; i<t.length(); i++)
39     {
40         t[i] = i;
41     }
42     cout << t["abc"] << endl;
43     cout << t["xzz"] << endl;
44     cout << t["451"] << endl;
45     return 0;
46 }

```

```

root@ubuntu:/home/x
1
2
input error
864680328
root@ubuntu:/home/x

```

这就是[]数组符号操作符重载的好处

根据字符串选择要返回的参数

初始化的时候把数据固定写入对象类

根据写入对象数组的字符串
下标决定取出哪一个数

C 语言和 C++ 在函数内部使用 static 的区别

C 语言全局 static 静态变量是在静态存储区，所以代码没有结束的话是一直存在的。不像子函数里面的变量，子函数调用结束变量就消失了。

```
1 #include<stdio.h>
2 int func()
3 {
4     static int a = 0;
5     a++;
6     return a;
7 }
8
9 int main()
10 {
11     int ret = 0, i = 0;
12     for(int i=0;i<5;i++)
13     {
14         ret = func();
15     }
16     printf("ret = %d\n",ret);
17     for(int i=0;i<5;i++)
18     {
19         ret = func();
20     }
21
22     printf("ret = %d\n",ret);
23     return 0;
24 }
```

但是在子函数里面，如果变量前面加了 static，那么该变量就是放在静态存储区，所以就算子函数执行结束了，static 变量的值还是存在的。并没有销毁

```
root@ubuntu:~/Documents/PycharmProjects/untitled/
root@ubuntu:/
ret = 5
ret = 10
```

```
1 #include<iostream>
2
3 using namespace std;
4
5 int func()
6 {
7     static int a = 0;
8     a++;
9     return a;
10 }
11
12 int main()
13 {
14     int ret = 0;
15     for(int i = 0;i<5;i++)
16     {
17         ret = func();
18     }
19     cout << "ret = "<<ret << endl;
20     for(int i = 0;i<5;i++)
21     {
22         ret = func();
23     }
24     cout << "ret = "<<ret << endl;
25     return 0;
26 }
```

在 C++ 里面子函数的 static 变量，也是放在静态存储区，也是不会随子函数结束而销毁，会一直存在，直到整个程序结束

```
root@ubuntu:~/Documents/PycharmProjects/untitled/
root@ubuntu:/
ret = 5
ret = 10
```

这就是函数内部静态变量的特性

函数括号 () 作为操作符重载

```
1 #include<iostream>
2
3 using namespace std;
4
5 int func()
6 {
7     static int a = 0;
8     a++;
9     return a;
10}
11
12 int main()
13 {
14     int ret = 0;
15     for(int i = 0; i < 5; i++)
16     {
17         ret = func();
18     }
19     cout << "ret = " << ret << endl;
20     for(int i = 0; i < 5; i++)
21     {
22         ret = func();
23     }
24     cout << "ret = " << ret << endl;
25     return 0;
26 }
```

整个代码就是调用一次函数静态变量加 1 一次，但是我现在想的是当静态变量加到 5 的时候，再次调用这个函数希望静态变量从 0 开始再次加，这样可行吗？明显这个代码是做不到的

```
root@ubuntu:~/Documents/2023/03/01/01 C++/operator_overload/1/
root@ubuntu:/
ret = 5
ret = 10
```

所以我们要用到函数括号 () 做操作符重载

```
1 #include<iostream>
2
3 using namespace std;
4 class fib
5 {
6     int a;
7 public:
8     fib()
9     {
10         a = 0;
11     }
12     int operator () ()
13     {
14         int ret = a;
15         a++;
16         return ret;
17     }
18 };
19
20 int main()
21 {
22     fib f1;
23
24     for(int i=0; i < 5; i++)
25     {
26         f1();
27     }
28     cout << "f1 = " << f1() << endl;
29
30     fib f2;
31     for(int i=0; i < 5; i++)
32     {
33         f2();
34     }
35     cout << "f2 = " << f2() << endl;
36
37     return 0;
38 }
```

这样就可以，我将对象里面的函数名用 operator () 括号表示，

f1 对象调用的 f1(), 就是对应的类成员()号名的函数

如果变量加到 5，我想从 0 从新开始加，我就剑走偏锋再定义一个对象就是了

```
root@ubuntu:~/Documents/2023/03/01/01 C++/operator_overload/1/
f1 = 5
f2 = 5
```

其实 C++ 函数操作符重载生成的函数对象是来取代 C 语言函数指针的。

智能指针就是帮助你自动释放内存的

智能指针主要是 C++11 标准提出来的，boost 库有专门的智能指针，我们这里模拟实现以下

```
1 #include<iostream>
2
3 using namespace std;
4
5 class test
6 {
7     int i;
8 public:
9     test(int i)
10    {
11        this->i = i;
12    }
13    int value()
14    {
15        return i;
16    }
17    ~test()
18    {
19    }
20 };
21
22 int main()
23 {
24     for(int i = 0; i < 5; i++)
25     {
26         test *p = new test(i);
27         cout << p->value() << endl;
28     }
29     return 0;
30 }
```

定义了一个 test 类的指针对象。用来存放在堆空间 new 的 test 对象

看起来打印没有错误，但是你在堆空间申请的 5 个对象并没有释放掉，所以会造成内存泄漏，我们要循环 5 次 delete 来释放 p 感觉太麻烦

```
root@ubuntu:/home/xzz/
0
1
2
3
4
```

有没有一种我忘记使用 delete，但是申请的内存会像 JAVA 那样有自动回收机制释放内存的方法呢？

那就必须使用智能指针

```
#include<iostream>
using namespace std;
class test
{
    int i;
public:
    test(int i)
    {
        this->i = i;
        cout << " test " << endl;
    }
    int value()
    {
        return i;
    }
    ~test()
    {
        cout << " ~~~~test " << endl;
    }
};
```

```

class pointer
{
    test *mp;
public:
    pointer(test *p = NULL)
    {
        mp = p;
    }
    test * operator ->()
    {
        return mp;
    }
    test & operator * }()
    ~pointer()
    {
        delete mp;
    }
};

int main()
{
    for(int i = 0;i<5; i++)
    {
        pointer p = new test(i);
        cout << p->value() << endl;
    }
    return 0;
}

```

```

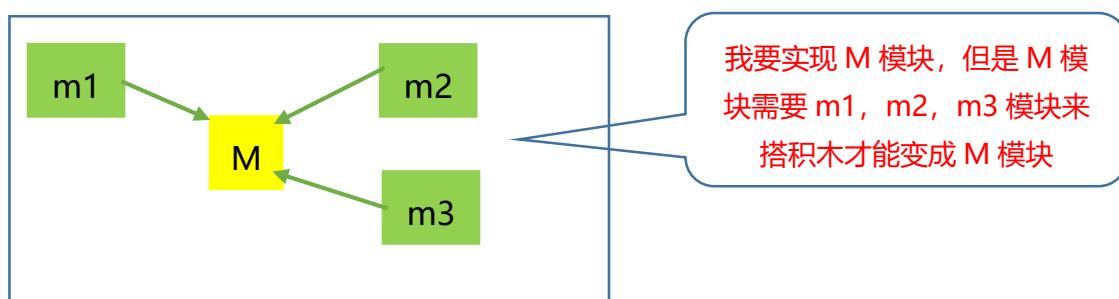
root@ubuntu:/home/xz
test
0
~~~test
test
1
~~~test
test
2
~~~test
test
3
~~~test
test
4
~~~test

```

New 的时候会自动执行
test 类里面的构造函数,

然后再把申请的 test 对
象地址赋值给 pointer
静态申请的指针 p, 注意
这个 p 是对象, 所以
是用对象模拟指针来接
收 new 分配的地址

类的继承关系，和面向对象



```

1 #include<iostream>
2
3 using namespace std;
4
5 class m1
6 {
7     public:
8         m1()
9         {
10             cout << "m1" << endl;
11         }
12         ~m1()
13         {
14             cout << "~~~m1" << endl;
15         }
16     };
17
18 class m2
19 {
20     public:
21         m2()
22         {
23             cout << "m2" << endl;
24         }
25         ~m2()
26         {
27             cout << "~~~m2" << endl;
28         }
29     };
30
31 class m3
32 {
33     public:
34         m3()
35         {
36             cout << "m3" << endl;
37         }
38         ~m3()
39         {
40             cout << "~~~m3" << endl;
41         }
42     };
43
44
45 class M
46 {
47     m1 objectm1;
48     m2 objectm2;
49     m3 objectm3;
50     public:
51         M()
52         {
53             cout << "M" << endl;
54         }
55         ~M()
56         {
57             cout << "~~~M" << endl;
58         }
59     };
60
61 int main()
62 {
63     M M1;
64     return 0;
65 }

```

定义 m1 模块

定义 m2 模块

定义 m3 模块

用 M 将 m1, m2, m3 模块组合起来

因为 m1, m2, m3 模块都和 M 一样是在栈上面定义的，所以一起创建，一起自动销毁

root@ubuntu:/home

m1
m2
m3
M
~~~~M  
~~~~m3  
~~~~m2  
~~~~m1

这种做法叫做组合关系，就是将 M 需要的模块 m1, m2, m3 组合进 M 类里面，然后一起使用。

但是我如果举例电脑，M 是电脑硬件，m1, m2, m3 分别是 CPU, 硬盘, 内存。那么我现在要安装两台电脑。一台惠普(HP)，一台苹果(mac)，我是不是每台电脑都要重写一遍 M, m1, m2, m3 呢？，因为每台电脑都有 CPU, 硬盘, 内存。

继承的功能就用上了

```
#include<iostream>
using namespace std;

class Parent
{
    int mv;
public:
    Parent()
    {
        mv = 100;
    }
    void func()
    {
        cout << "Parent mv = " << mv << endl;
    }
};

class child:public Parent
{
};

int main()
{
    child chd;
    chd.func();

    return 0;
}
```

因为父类的函数是用 public 修饰的，所以为了能够外部调用，这里要声明 public

```
class Parent
{
    int mv;
public:
    Parent()
    {
        mv = 100;
    }
    void func()
    {
        cout << "Parent mv = " << mv << endl;
    }
};

class child:public Parent
{
public:
    void childfunc()
    {
        cout << "child class" << endl;
    }
};

int main()
{
    child chd;
    chd.func();
    chd.childfunc();
    return 0;
}
```

C++和JAVA不一样，child(子类)继承 Parent(父类)用：分号表示
JAVA 是用 extends 修饰

这样子类对象可以调用父类函数，也可以调用自己的函数

```
root@ubuntu:/home
Parent mv = 100
root@ubuntu:/home
```

```
root@ubuntu:/home/>
Parent mv = 100
child class
root@ubuntu:/home/>
```

现在我们来实现 HP 电脑和 mac 电脑的程序

```
class m1
{
public:
    m1()
    {
        cout << "m1" << endl;
    }
    ~m1()
    {
        cout << "~~~m1" << endl;
    }
};

class m2
{
public:
    m2()
    {
        cout << "m2" << endl;
    }
    ~m2()
    {
        cout << "~~~m2" << endl;
    }
};

class m3
{
public:
    m3()
    {
        cout << "m3" << endl;
    }
    ~m3()
    {
        cout << "~~~m3" << endl;
    }
};

class M
{
    m1 objectm1;
    m2 objectm2;
    m3 objectm3;
public:
    M()
    {
        cout << "M" << endl;
    }
    void power()
    {
        cout << " power ON" << endl;
    }
    ~M()
    {
        cout << "~~~M" << endl;
    }
};

class HP:public M
{
public:
    void install()
    {
        cout << "HP install OS" << endl;
    }
};

class mac
{
};

int main()
{
    HP h;
    h.power();
    h.install();
    return 0;
}
```

我们先实现所有电脑都有的功能 m1(CPU), m2(硬盘), m3(内存)功能

我将通用硬件功能组合在 M 类里面, 然后我 M 类再实现一个开机的功能

然后我惠普(HP)类继承电脑通用功能类 M, 然后自己增加一个 HP 电脑的操作系统安装函数

我惠普(HP)类的对象可以用通用类 M 的电脑开机功能

然后我惠普类还有自己的操作系统可以安装

```
root@ubuntu:/ho
m1
m2
m3
M
power ON
HP install OS
~~~M
~~~m3
~~~m2
~~~m1
```

这是惠普类继承通用类 M 实现的功能。

下面我们再操作苹果电脑

```
class HP:public M
{
public:
void install()
{
    cout <<"HP install OS" <<endl;
}

class mac:public M
{
public:
void install()
{
    cout << "install mac OS " <<endl;
}

int main()
{
    HP h;
    h.power();
    h.install();

    mac ap;
    ap.power();
    ap.install();
    return 0;
}
```

苹果(mac)类直接和 HP 类一样将电脑通用的功能 M 继承(抄袭)过来，直接用于开机。然后苹果类自己只增加了一个安装苹果操作系统的功能

调用电脑通用的开机函数

调用苹果电脑自己的操作系统

```
root@ubuntu:/home/xzz
m1
m2
<m3
M
power ON
HP install OS
m1
m2
m3
M
power ON
install mac OS
~~~M
~~~m3
~~~m2
~~~m1
~~~M
~~~m3
~~~m2
~~~m1
root@ubuntu:/home/xzz
```

这就是类继承和类组合的区别，类组合是把其它功能模块组合在一起实现自己需要的功能。但是类继承是父类代码都不写，直接将父类代码抄过来自己调用，然后再增加点自己的功能。

类里面 Public, Private, Protected 三个权限的区别

```
class Parent
{
private:
int mv;
public:
Parent()
{
    mv = 10;
}
int value()
{
    return mv;
}
};

class child:Parent
{
    int get()
    {
        return mv;
    }
};

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

定义类私有变量或者函数

类里面 private 修饰的变量或者函数都是类私有的，所以只有类自己可以调用，其它子类或者外部函数无法调用

```
root@ubuntu:/home/xzz/code/C++/class/subclass# g++ -o sub subclass1.cpp: In member function ‘int child::get()’:
sub class1.cpp:10:13: error: ‘int Parent::mv’ is private
int mv;
```

```

6 class Parent
7 {
8     protected:
9         int mv;
10    public:
11        Parent()
12        {
13            mv = 10;
14        }
15        int value()
16        {
17            return mv;
18        }
19
20    };
21
22 class child:Parent
23 {
24     public:
25     int get()
26     {
27         return mv;
28     }
29 };
30
31 int main()
32 {
33     child chd;
34     cout << chd.get() << endl;
35     return 0;
36 }

```

如果你子类确实想调用父类的私有变量或者函数，就给变量或者函数定义为 protected

然后我的子类就可以调用父类的变量了

```

root@ubuntu:/h
10
root@ubuntu:/h

```

但是有一点要注意 protected 修饰的变量或函数只能子类调用，外部其它类或者程序无法调用

```

5 class Parent
6 {
7     protected:
8         int mv;
9    public:
10        Parent()
11        {
12            mv = 10;
13        }
14        int value()
15        {
16            return mv;
17        }
18
19    };
20
21 class child:Parent
22 {
23     public:
24     int get()
25     {
26         return mv;
27     }
28 };
29
30 int main()
31 {
32     Parent p;
33     cout << p.mv << endl;
34     return 0;
35 }

```

外部无法直接调用父类的 protected 修饰的变量，必须经过子类调用父类 protected 修饰的变量，来间接调用父类变量

```

10
root@ubuntu:/home/xzz/code/C++/class/subclass# g++ -o subclass
'subclass3.cpp': In function 'int main()':
'subclass3.cpp':10:13: error: 'int Parent::mv' is protected
          ^~~~~~

```

```
class child:Parent
{
    public:
        int get()
    {
        return mv;
    }
};

int main()
{
    child chd;
    cout << chd.mv << endl;
    return 0;
}
```

子类对象也无法调用父类
protected 修饰的变量或函数,
必须经过子类的函数间接调用

```
root@ubuntu:/home/xzz/code/C++/class/subclass#
subclass4.cpp: In function 'int main()':
subclass4.cpp:10:13: error: 'int Parent::mv' i
    int mv;
```

C++子类继承父类有三种权限方式, public, private, protected

这三种权限有什么区别呢?

```
3
4 class parent
5 {
6
7 };
8
9 class child_A:public parent
0 {
1
2 };
3 class child_B:private parent
4 {
5
6 };
7 class child_C:protected parent
8 {
9
0 };
1
2 int main()
3 {
4
5     return 0;
6 }
```

子类继承父类权限可以是 public

子类继承父类权限可以是 private

子类继承父类权限可以是
protected

子类 public 继承父类, 就是将父类里面的成员属性权限原封不动的继承过来。

```

5 class parent
6 {
7     private:
8         int pri;
9     protected:
10        int pro;
11    public:
12        int pub;
13
14        void set(int a,int b,int c)
15        {
16            pri = a;
17            pro = b;
18            pub = c;
19        }
20 };
21
22 class child_A:public parent
23 {
24     public:
25         void print()
26         {
27             cout << "pri" << pri << endl;
28             cout << "pri" << pro << endl;
29             cout << "pri" << pub << endl;
30         }
31 };
32
33 class child_B:private parent
34 {
35 };
36
37 class child_C:protected parent
38 {
39 };
40
41 int main()
42 {
43
44     class parent
45     {
46         private:
47             int pri;
48         protected:
49             int pro;
50         public:
51             int pub;
52
53             void set(int a,int b,int c)
54             {
55                 pri = a;
56                 pro = b;
57                 pub = c;
58             }
59     };
60
61     class child_A:public parent
62     {
63         public:
64             void print()
65             {
66                 cout << "pri" << pro << endl;
67                 cout << "pri" << pub << endl;
68             }
69     };
70     class child_B:private parent
71     {
72     };
73
74     class child_C:protected parent
75     {
76     };
77
78
79     int main()
80     {
81         child_A a;
82         a.pri = 100;
83         a.pro = 200;
84         a.pub = 300;
85         return 0;
86     }
87 }

```

父类 private 成员，子类继承过来也是 private
父类 public 成员，子类继承过来也是 public
父类 protected 成员，子类继承过来也是 protected

这里为什么不行，是因为父类 pri
这个变量是 private 私有的，只能父类里面使用

root@ubuntu:~/Documents/Code/CH/Chap07/inherit2\$ g++ -o inherit2 inherit2.cpp
inherit2.cpp: In member function ‘void child_A::print()’:
inherit2.cpp:9:7: error: ‘int parent::pri’ is private
 int pri;
^
inherit2.cpp:28:21: error: within this context
 cout << "pri" << pri << endl;

所以最后只有
public 的成员
能访问

子类 private 继承父类，就是将父类里面的成员属性权限全部改成 private 继承过来。

```
6 class parent
7 {
8     private:
9         int pri;
10    protected:
11        int pro;
12    public:
13        int pub;
14
15        void set(int a,int b,int c)
16        {
17            pri = a;
18            pro = b;
19            pub = c;
20        }
21 };
22
23 class child_A:public parent
24 {};
25
26 class child_B:private parent
27 {
28     public:
29         void print()
30         {
31             cout << "pri" << pri << endl;
32             cout << "pri" << pro << endl;
33             cout << "pri" << pub << endl;
34         }
35 };
36
37 class child_C:protected parent
38 {};
39
40
41 int main()
42 {
43     return 0;
44 }
```

用 private 方式继承类

```
3 class child_A:public parent
4 {
5 };
6 class child_B:private parent
7 {
8     public:
9         void print()
10        {
11            cout << "pri" << pro << endl;
12            cout << "pri" << pub << endl;
13        }
14 };
15
16 class child_C:protected parent
17 {};
18
19
20 int main()
21 {
22     child_B b;
23     b.pub = 100;
24
25     return 0;
26 }
```

虽然子类用 private 方式继承了父类，但是在子类内部可以访问父类的 public 属性的成员

因为子类是用 private 方式继承的父类，导致父类里面不管是 public，还是 protected 成员都变成了 private 属性，所以除了子类，外部是无法访问

至于子类用 protected 去继承父类方法，自己去网上看。但是现在大部分软件都只使用 public 继承所以子类用 protected 和 private 继承父类的方法是没什么用的。所以被人遗弃了。

C++子类继承父类，构造函数的调用顺序和析构函数的调用顺序

子类继承父类构造函数分隐式规则和显示规则

```
5 class parent
6 {
7     public:
8         parent(string s)
9         {
10             cout << "parent s=" << s << endl;
11         }
12     };
13 };
14
15 class child:public parent
16 {
17     public:
18         child()
19         {
20             cout << "child" << endl;
21         }
22         child(string s)
23         {
24             cout << "child s = " << s << endl;
25         }
26     };
27 };
28
29 int main()
30 {
31     child c; ←
32     return 0;
33 }
```

如果没有指定父类的构造函数，那么子类对象被创建的时候会自动去父类找无参构造函数

因为父类没有无参构造函数，所

```
root@ubuntu:/home/xzz/code/C++/gouzao# g++ -o parentchild parentchild.cpp
parentchild.cpp: In constructor 'child::child()':
parentchild.cpp:19:2: error: no matching function for call to 'parent::parent()'
{
^
父类增加了无参构造函数

class parent
{
public:
    parent()
    {
        cout << "parent " << endl;
    }
    parent(string s)
    {
        cout << "parent s=" << s << endl;
    }
};

class child:public parent
{
public:
    child()
    {
        cout << "child" << endl;
    }
    child(string s)
    {
        cout << "child s = " << s << endl;
    }
};

int main()
{
    child c;
    return 0;
}

先调用父类无参构造函数
再调用子类无参构造函数
root@ubuntu:/home/xzz/code/C++/gouzao# ./parentchild
parent
child
```

```

class parent
{
public:
    parent()
    {
        cout << "parent " << endl;
    }
    parent(string s)
    {
        cout << "parent s=" << s << endl;
    }
};

class child:public parent
{
public:
    child()
    {
        cout << "child" << endl;
    }
    child(string s)
    {
        cout << "child s = " << s << endl;
    }
};

int main()
{
    child c2("cc");
    return 0;
}

```

因为隐式规则所以先调用子类有参构造函数，然后默认调用父类无参构造函数

root@ubuntu:/home/
parent
child s = cc

下面我们看看显示规则的构造函数调用方法

```

class parent
{
public:
    parent()
    {
        cout << "parent " << endl;
    }
    parent(string s)
    {
        cout << "parent s=" << s << endl;
    }
};

class child:public parent
{
public:
    child()
    {
        cout << "child" << endl;
    }
    child(string s):parent(s)
    {
        cout << "child s = " << s << endl;
    }
};

int main()
{
    child c2("cc");
    return 0;
}

```

显示规则就是在子类构造函数中规定父类自动调用哪一种构造函数的方法，这个 s 可以直接传入父类

root@ubuntu:/home/xzz/cod
parent s=cc
child s = cc
root@ubuntu:/home/xzz/cod

这就是构造函数显示和隐式规则

```

5 class parent
6 {
7     public:
8         parent(string s)
9         {
10             cout << "parent s=" << s << endl;
11         }
12     };
13
14 class child:public parent
15 {
16     public:
17         child():parent("default")
18         {
19             cout << "child" << endl;
20         }
21         child(string s):parent(s)
22         {
23             cout << "child s = " << s << endl;
24         }
25     };
26
27 int main()
28 {
29     child c2("cc");
30     return 0;
31 }

```

如果我们确实不想在父类定义无参构造函数
我们就要在子类指定好父类自动调用的构造函数
这里也要指定

root@ubuntu:/home/x
parent s=cc
child s = cc

```

5
6 class parent
7 {
8     public:
9         parent(string s)
10        {
11            cout << "parent s=" << s << endl;
12        }
13    };
14
15 class child:public parent
16 {
17     parent p1;
18     parent p2;
19     public:
20         child():parent("default")
21         {
22             cout << "child" << endl;
23         }
24         child(string s):parent(s)
25         {
26             cout << "child s = " << s << endl;
27         }
28    };
29
30 int main()
31 {
32     child c2("cc");
33     return 0;
34 }

```

如果在子类又再次创建了父类的对象
根据构造函数调用口诀，先父类再子类里面的父类对象，再是子类自己，我子类创建父类对象的时候会去调用父类，但是发现没有无参构造函数

root@ubuntu:/home/xzz/code/C++/gouzao# g++ -o parentchild5 parentchild5.cpp
parentchild5.cpp: In constructor ‘child::child()’:
parentchild5.cpp:21:26: error: no matching function for call to ‘parent::parent()’
 child():parent("default")

所以编译会报错，说没有无参构造函数。为了避免父类没有无参构造函数，我们必须在列表指定对象。

```

class parent
{
public:
    parent(string s)
    {
        cout << "parent s=" << s << endl;
    }
};

class child:public parent
{
    parent p1;
    parent p2;
public:
    child():parent("default"),p1("p1"),p2("p2")
    {
        cout << "child" << endl;
    }
    child(string s):parent(s),p1("p1"),p2("p2")
    {
        cout << "child s = " << s << endl;
    }
};

int main()
{
    child c2("cc");
    return 0;
}

```

```

root@ubuntu:/home
parent s=cc
parent s=p1
parent s=p2
child s = cc

```

你看输出和口诀一样，先父类，再父类对象，在子类自己

```

class parent
{
public:
    parent(string s)
    {
        cout << "parent s=" << s << endl;
    }
    ~parent()
    {
        cout << "~~~parent" << endl;
    }
};

class child:public parent
{
    parent p1;
    parent p2;
public:
    child():parent("default"),p1("p1"),p2("p2")
    {
        cout << "child" << endl;
    }
    child(string s):parent(s),p1("p1"),p2("p2")
    {
        cout << "child s = " << s << endl;
    }
    ~child()
    {
        cout << "~~~child" << endl;
    }
};

int main()
{
    child c2("cc");
    return 0;
}

```

```

root@ubuntu:/home/xzz/code/C++/gouz
parent s=cc
parent s=p1
parent s=p2
child s = cc
~~~child
~~~parent
~~~parent
~~~parent
root@ubuntu:/home/xzz/code/C++/gouz

```

用显示规则在列表中指定对象的构造函数

这里也要跟着写，在列表中指定对象的构造函数，有多少个构造函数就重复写多少个指定对象构造函数

析构函数就是先子类析构函数，然后父类对象的析构函数，最后父类析构函数

这就是父子类之间的析构函数顺序

子类中定义了和父类一样的同名变量，同名函数，是否会和父类冲突

父子类变量同名

```
class parent
{
public:
    int mi;
    parent()
    {
        mi = 200;
    }
};

class child:public parent
{
public:
    int mi;
    child()
    {
        mi = 100;
    }
};

int main()
{
    child c;
    parent p;
    cout << "child mi = " << c.mi << endl;
    cout << "parent mi = " << p.mi << endl;
    return 0;
}
```

如果子类和父类有同名的变量

子类访问的 mi 变量是子类作用域里面的

root@ubuntu:/home/xz
child mi = 100
parent mi = 200
root@ubuntu:/home/xz

```
class parent
{
public:
    int mi;
    parent()
    {
        mi = 200;
    }
};

class child:public parent
{
public:
    int mi;
    child()
    {
        mi = 100;
    }
};

int main()
{
    child c;
    cout << "child mi = " << c.mi << endl;
    cout << "parent mi = " << c.parent::mi << endl;
    return 0;
}
```

如果想用子类对象访问父类同名的变量就需要加作用域标识符

最后输出结果和上面一样

root@ubuntu:/home/xz
child mi = 100
parent mi = 200
root@ubuntu:/home/xz

```

5 namespace A
6 {
7     int g=500;
8 }
9
10 namespace B
11 {
12     int g=700;
13 }
14
15 class parent
16 {
17     public:
18         int mi;
19
20         parent()
21     {
22         mi = 200;
23     }
24 };
25
26 class child:public parent
27 {
28     public:
29         int mi;
30
31         child()
32     {
33         mi = 100;
34     }
35 };
36
37 int main()
38 {
39     child c;
40     cout << "child mi = " << c.mi << endl;
41     cout << "parent mi = " << c.parent::mi << endl;
42     cout << "A::g = " << A::g << endl;
43     cout << "B::g = " << B::g << endl;
44
45     return 0;
46 }

```

其实父子类变量同名就类似命名空间。
在命名空间两个全局变量虽然同名，但是不会出现像 C 语言那样全局变量同名冲突，因为两个命名空间的全局变量都在自己命名空间的作用域范围内，不会影响到其它命名空间，比如 B 空间的变量不会影响到 A 空间的同名变量

调用全局变量都必须指定命名空间名

```

root@ubuntu:/home/xzz/code/C++/parentsub#
root@ubuntu:/home/xzz/code/C++/parentsub#
child mi = 100
parent mi = 200
A::g = 500
B::g = 700

```

父子类函数同名

```
1 class parent
2 {
3     public:
4         int mi;
5
6         void print()
7         {
8             cout << "parent ->print " << endl;
9         }
10        int print(int m)
11        {
12            mi = m;
13            return mi;
14        }
15    };
16
17 class child:public parent
18 {
19     public:
20         int mi;
21 };
22
23 int main()
24 {
25     child c;
26     c.print();
27     cout << "c.print(10) = " << c.print(10) << endl;
28     return 0;
29 }
```

子类继承了父类，如果子类没有父类的同名函数，可以直接使用父类的函数

也可以直接使用父类的重载函数

```
root@ubuntu:/home/xzj
parent ->print
c.print(10) = 10
8 class parent
9 {
10     public:
11         int mi;
12
13         void print()
14         {
15             cout << "parent ->print " << endl;
16         }
17         int print(int m)
18         {
19             mi = m;
20             return mi;
21         }
22    };
23
24 class child:public parent
25 {
26     public:
27         int mi;
28         int print(int x,int y)
29         {
30             return x+y;
31         }
32    };
33
34 int main()
35 {
36     child c;
37     c.print();
38     cout << "c.print(10) = " << c.print(10) << endl;
39     return 0;
40 }
```

如果子类定义了和父类同名的函数

那么你直接用子类对象调用父类的同名函数就会报错

```
parents6.cpp: In function 'int main()':
parents6.cpp:37:10: error: no matching function for call to 'child::print()'
      c.print();
      ^
parents6.cpp:28:7: note: candidate: int child::print(int, int)
      int print(int x,int y)
      ^
```

```

class parent
{
public:
    int mi;

    void print()
    {
        cout << "parent ->print " << endl;
    }
    int print(int m)
    {
        mi = m;
        return mi;
    }
};

class child:public parent
{
public:
    int mi;
    int print(int x,int y)
    {
        return x+y;
    }
};

int main()
{
    child c;
    c.parent::print();
    cout << "c.print(10) = " << c.parent::print(10) << endl;
    return 0;
}

```

只有指定父类的作用域，子类才可以调用父类的同名函数

```

root@ubuntu:/home/xzz/c
parent ->print
c.print(10) = 10
root@ubuntu:/home/xzz/c

```

其实子类和父类有同名函数，最好的方法就是在调用的时候指定作用域。比如下面这样

```

class parent
{
public:
    int mi;

    void print()
    {
        cout << "parent ->print " << endl;
    }
    int print(int m)
    {
        mi = m;
        return mi;
    }
};

class child:public parent
{
public:
    int mi;

    void print()
    {
        cout << "child ->print " << endl;
    }
    int print(int m)
    {
        mi = m;
        return mi;
    }
};

int main()
{
    child c;
    c.print();
    c.parent::print();

    cout << "parent mi = " << c.parent::print(100) << endl;
    cout << "child mi = " << c.print(10) << endl;

    return 0;
}

```

子类和父类函数重载
也是重名的

所以用作用域来解析

```

root@ubuntu:/home/xzz/code/C++/p
child ->print
parent ->print
parent mi = 100
child mi = 10

```

输出结果正确

C++的多态使用 virtual 关键字

```
#include<iostream>

using namespace std;

class parent
{
public:
    void print()
    {
        cout << "parent ...." << endl;
    }
};

class child:public parent
{
public:
    void print()
    {
        cout << "child ...." << endl;
    }
};

void point_print(parent *p)
{
    p->print();
}

int main()
{
    parent p;
    child c;
    point_print(&p);
    point_print(&c);
    return 0;
}
```

父类定义一个函数

子类继承父类后，又定义了一个同名函数

定义一个指向父类的函数，传入父类对象地址，调用父类的函数，传入子类对象地址调用子类的函数

传入父类对象

传入子类对象

```
root@ubuntu:/home
parent ....
parent ....
```

输出的结果都是调用父类的函数

```
void point_print(parent *p)
{
    p->print();
```

这是编译器为了安全，认为你的函数形参是父类指针，所以不管你传入进来的是子类对象地址还是父类对象地址，都认定为父类。前提是你的子类也是继承了这个父类，而且是同名函数

为了解决传入对象地址的区别，我们需要用到多态

- 同样的调用语句在实际运行时有多种不同的表现形态

```
void print()
{
    cout << "I'm Parent."
    << endl;
}

void print()
{
    cout << "I'm Child."
    << endl;
}
```

p->print(); { p 指向父类对象 p 指向子类对象 }

多态就是传入函数的对象地址，是父类对象就调用父类对象函数，是子类对象就调用子类对象函数

```
using namespace std;

class parent
{
public:
    virtual void print()
    {
        cout << "parent ...." << endl;
    }
};

class child:public parent
{
public:
    void print()
    {
        cout << "child ...." << endl;
    }
};

void point_print(parent *p)
{
    p->print();
}

int main()
{
    parent p;
    child c;
    point_print(&p);
    point_print(&c);
    return 0;
}
```

在父类同名函数加上 virtual，证明子类如果定义了该同名函数，也是虚函数

这样子类也就是虚函数，虽然我函数前面没有加 virtual，但是默认就是

这样的话虽然你函数形参是父类指针，但是这个父类也被子类继承了，那么你传入父类对象就调用父类函数，传入子类对象就调用子类函数

```
root@ubuntu:/home/zhengyuan/CLionProjects/untitled1$ g++ -fpermissive -std=c++11 main.cpp
root@ubuntu:/home/zhengyuan/CLionProjects/untitled1$ ./a.out
parent ...
child ...
root@ubuntu:/home/zhengyuan/CLionProjects/untitled1$
```

这就是 C++ 多态，多态的虚函数可以解决代码分离的问题。

这个 C++ 多态到底有什么用，下面举例

```
#include<iostream>

using namespace std;

class A
{
public:
    int outaaa()//在A类实现个outaaa函数，返回值为10
    {
        int ret = 10;
        cout << "outaaa = " << ret << endl;
        return ret;
    }
};

class B
{
public:
    int outbbb()//在B类实现个outbbb函数，返回值为8
    {
        int ret = 8;
        cout << "outbbb = " << ret << endl;
        return ret;
    }
};

void compare(A *a, B *b)//比较A类里面的函数和b类里面的函数大小
{
    int a1 = a->outaaa();
    int b1 = b->outbbb();
    if(a1 < b1)
    {
        cout << "outaaa < outbbb" << endl;
    }
    else
    {
        cout << "outaaa > outbbb" << endl;
    }
}

int main()
{
    A a;
    B b;
    compare(&a,&b);

    return 0;
}
```

```
root@ubuntu:/home/xzz/code/C++/virtual# ./virtual
outaaa = 10
outbbb = 8
outaaa > outbbb
```

这是 outaaa 大于 outbbb

现在我想 outaaa 小于 outbbb 该怎么办呢？我可以去修改 A 类里面的 outaaa 函数，但是如果这是别人封装好的函数库，只是提供给我们使用呢？我们认为 A 类里面的 outaaa 函数实现的功能不是很强大，想给 outaaa 函数增加点功能，或者直接修改 outaaa 函数里面的功能，但是我要保持 A 类里面的其它函数不变。怎么办？所以这时候就必须要求写这个封装库的人将 outaaa 修改成虚函数。

这样我就可以用子类去继承这个 A 类，然后重新实现 outaaa 这个函数，那么 A 类里面原有的 outaaa 函数就被屏蔽了。如果我想同时直接使用 A 类里面的 outaaa 函数，我就单独创建个 A 类的对象来使用。

```

using namespace std;

class A
{
public:
    virtual int outaaa()//把A改成虚函数
    {
        int ret = 10;
        cout << "outaaa = " << ret << endl;
        return ret;
    }
};

class B
{
public:
    int outbbb()//在B类实现个outbbb函数，返回值为8
    {
        int ret = 8;
        cout << "outbbb = " << ret << endl;
        return ret;
    }
};

class CCC:public A
{
public:
    int outaaa()//在A类里面实现的outaaa函数重新实现一次
    {
        int ret = 5;
        cout << "outCCC = " << ret << endl;
        return ret;
    }
};

void compare(A *a, B *b)//这个地方不需要修改，因为ccc子类是继承了A类的，所以用A类做形参接受子类是可以的
{
    int a1 = a->outaaa();
    int b1 = b->outbbb();
    if(a1 < b1)
    {
        cout << "outaaa < outbbb" << endl;
    }
    else
    {
        cout << "outaaa > outbbb" << endl;
    }
}

int main()
{
    B b;
    CCC c;
    compare(&c,&b); //这里传入参数把以前的A类变成ccc子类定义的对象传入
    return 0;
}

```

修改 A 这个父类

B 类不变

我定义个 CCC 子类来继承父类 A。然后重新写 A 类实现过的 outaaa 函数

多态的好处就是你子类不仅重写了 A 类里面的函数。而且依靠 A 类实现的一些函数的形参还可以继续保持 A 类形参，这时候传入子类的对象是没有问题的

你看我形参保持 A 类，但是我传入的是 CCC 子类的对象。一样没有问题，compare 里面的 a->outaaa 就直接调用 CCC 类实现的函数了

你看 outaaa 小于 outbbb

但是我有时候需要使用 A 类里面的 outaaa 怎么办呢？不是任何时候都用 CCC 类的 outaaa

```

int main()
{
    B b;
    CCC c;
    compare(&c,&b); //这里传入参数把以前的A类变成ccc子类定义的对象传入

    A a;//如果我不想使用ccc子类里面的outaaa函数，而是想使用A类里面的outaaa函数，那么就单独定义个A类
    compare(&a,&b);
    return 0;
}

```

你看父类子类的 outaaa 函数都可以用，其实多态解决了在 C 语言编程时，比如我想修改一个别人写好的库函数，但是为了不打乱 c 文件，我还要单独去新建一个 C 文件和 H 文件，而且函数名还要另外取，这样文件管理就很麻烦记忆也很混乱，因为修改一个函数而且单独建立文件很不划算，而且去单独建立个函数名也不好记，所以 C++ 多态就解决了这个问题，这就叫函数重写

Struct 和 class 内存分配问题

```
6 class A{  
7     int i;//4字节  
8     int j;//4字节  
9     char c;//本来char是1个字节，因为内存对齐所以加了3个空字节，char就占用4字节  
0     double d;//8字节  
1 };  
2  
3 struct B{  
4     int i;  
5     int j;  
6     char c;  
7     double d;  
8 };  
9  
0 int main()  
1 {  
2     cout << "sizeof A = " << sizeof(A) << endl;  
3     cout << "sizeof b = " << sizeof(B) << endl;  
4     return 0;  
5 }
```

```
root@ubuntu:/h  
sizeof A = 24  
sizeof b = 24  
root@ubuntu:/h
```

确实结构体和 class 内存分配是一样的

```
6 class A{  
7     int i;  
8     int j;  
9     char c;  
0     double d;  
public:  
1     void print(){  
2         cout << "i = " << i << endl;  
3         cout << "j = " << j << endl;  
4         cout << "c = " << c << endl;  
5         cout << "d = " << d << endl;  
6     }  
7 };  
8  
9 struct B{  
1     int i;  
2     int j;  
3     char c;  
4     double d;  
5 };  
6  
7 int main()  
8 {  
9     cout << "sizeof A = " << sizeof(A) << endl;  
0     cout << "sizeof b = " << sizeof(B) << endl;  
1     return 0;  
2 }
```

我们在类中增加了函数，会不会增加类的内存占用空间？

```
root@ubuntu:/hom  
sizeof A = 24  
sizeof b = 24  
root@ubuntu:/hom
```

根据输出结果，在类中增加函数是不会增加内存空间的？是因为变量在栈区，或者堆区，或者全局区而函数是在代码区，所以计算的时候只计算了变量的占用区域，没有计算函数的占用区域。

```

class A{
    int i;
    int j;
    char c;
    double d;
public:
    void print(){
        cout << "i = " << i << endl;
        cout << "j = " << j << endl;
        cout << "c = " << c << endl;
        cout << "d = " << d << endl;
    }
};

struct B{
    int i;
    int j;
    char c;
    double d;
};

int main()
{
    A a1;           // 创建对象，对象会在栈空间占用空间
    cout << "sizeof A = " << sizeof(A) << endl;
    cout << "sizeof B = " << sizeof(B) << endl;
    cout << "sizeof a1 = " << sizeof(a1) << endl;
    return 0;
}

```

创建对象，对象会在栈空间占用空间

root@ubuntu:/home/x:
sizeof A = 24
sizeof B = 24
sizeof a1 = 24
root@ubuntu:/home/x:

C++抽象类使用

```

class shape{
public:
    double area()
    {
        return 10;
    }
};

int main()
{
    shape s;
    cout << "area = " << s.area()<< endl;
    return 0;
}

class shape{
public:
    virtual double area(); //函数不在类中实现 然后定义了virtual就是虚函数
};

int main()
{
    shape s;
    return 0;
}

```

这是常规类调用程序

root@ubuntu:
area = 10

编译报错，系统并不知道你这个类是抽象类还是虚函数，虚函数也没有实现过程

```
root@ubuntu:/home/xzz/code/C++/abstract# g++ -o abstract2 abstract2.cpp  
/tmp/ccvdv41x.o: In function `shape::shape()':  
abstract2.cpp:(.text._ZN5shapeC2Ev[_ZN5shapeC5Ev]+0x9): undefined reference to `vtable for shape'  
collect2: error: ld returned 1 exit status
```

```
5 class shape{  
6     public:  
7         virtual double area()=0; //这里要=0 才能告诉编译器我这个函数不需要在这个类里面实现  
8 };  
9 int main()  
10 {  
11     return 0;  
12 }
```

这里写 0 才告诉了编译器这是抽象类，这个函数要交给继承的子类去实现

```
abstract# g++ -o abstract3 abstract3.cpp  
abstract#
```

```
class shape{  
    public:  
        virtual double area()=0;  
};  
  
int main()  
{  
    shape s;//抽象类是不能创建对象的，只能被子类继承，交给子类去创建对象  
    return 0;  
}
```

```
abstract3.cpp:5:7: note: because the following virtual function is unimplemented:  
class shape{  
^
```

```
abstract3.cpp:7:18: note:     virtual double shape::area()  
virtual double area()=0;
```

```
class shape{  
    public:  
        virtual double area()=0; //这里是纯虚函数了抽象类，所以要子类来实现该函数  
};  
  
class Rect:public shape//子类要继承父类，才能实现父类的虚函数  
{  
    int ma,mb;  
    public:  
        Rect(int a,int b)  
        {  
            ma=a;  
            mb=b;  
        }  
        double area()  
        {  
            return ma+mb;  
        }  
};  
  
int main()  
{  
    Rect r(5,10);  
    cout << "area = "<<r.area()<<endl;  
    return 0;  
}
```

```
root@ubuntu:/home/xzz/code/C++/abstract# g++ -o abstract4 abstract4.cpp  
root@ubuntu:/home/xzz/code/C++/abstract# ./abstract4  
area = 15
```

抽象类一般就是 A 工程师写一个抽象类，给 B 工程师去实现。然后 A 工程师拿着 B 工程师实现的抽象类和函数去设计自己擅长的程序。

抽象类还有个特点，抽象类可以用做接口，下面章节的工厂方法，代码分离，就会用到抽象类。

不怎么用的多重继承

```
#include<iostream>
#include<string>
using namespace std;

class A
{
    int ma;
};

class B
{
    int mb;
};

class child: public A,public B
{
    int mc;
};

int main()
{
    cout <<"size = " <<sizeof(child)<<endl;
    return 0;
}
```

在子类里面多继承 1 个类就是多重继承

```
root@ubuntu:/home
size = 12
root@ubuntu:/home
```

输出结果正确，计算的是子类变量和继承类的变量占用的字节空间

```
4 class A
5 {
6     int ma;
7 public:
8     A(int a)
9     {
10         ma = a;
11     }
12     int getA()
13     {
14         return ma;
15     }
16 };
17
18 class B
19 {
20     int mb;
21 public:
22     B(int b)
23     {
24         mb = b;
25     }
26     int getB()
27     {
28         return mb;
29     }
30 };
31
32 class child: public A,public B
33 {
34     int mc;
35 public:
36     child(int a,int b,int c) : A(a),B(b)
37     {
38         mc = c;
39     }
40 };
41
42 int main()
43 {
44     cout <<"size = " <<sizeof(child)<<endl;
45     return 0;
46 }
```

虽然我们增加了函数，但是计算的还是变量的占用空间。因为函数和变量占用空间是分开的。

```
root@ubuntu:/home
size = 12
root@ubuntu:/home
```

```
class child: public A,public B
{
    int mc;
public:
    child(int a,int b,int c) : A(a),B(b)
    {
        mc = c;
    }
    void print()
    {
        cout << "ma = "<<ma<<endl;
        cout << "ma = "<<mb<<endl;
        cout << "ma = "<<mc<<endl;
    }
};

int main()
{
    cout <<"size = "<<sizeof(child)<<endl;
    return 0;
}
```

multi3.cpp: In member function ‘void child::print()’
multi3.cpp:7:6: error: ‘int A::ma’ is private
int ma;
^

编译错误，因为 ma 和 mb 在父类是私有的

```
3 class child: public A,public B
4 {
5     int mc;
6 public:
7     child(int a,int b,int c) : A(a),B(b)
8     {
9         mc = c;
10    }
11    void print()
12    {
13        cout << "ma = "<<getA()<<endl;
14        cout << "ma = "<<getB()<<endl;
15        cout << "ma = "<<mc<<endl;
16    }
17};

18 int main()
19 {
20     child chd(1,2,3);
21     cout <<"size = "<<sizeof(child)<<endl;
22     chd.print();
23     return 0;
24 }
```

root@ubuntu:/h
size = 12
ma = 1
ma = 2
ma = 3

所以要用父类
函数取间接调
用私有变量

我们以上实验都没有问题，但是多重继承遇到指针就要出问题

```
class A
{
    int ma;
public:
    A(int a)
    {
        ma = a;
    }
    int getA()
    {
        return ma;
    }
};

class B
{
    int mb;
public:
    B(int b)
    {
        mb = b;
    }
    int getB()
    {
        return mb;
    }
};

class child: public A,public B
{
    int mc;
public:
    child(int a,int b,int c) : A(a),B(b)
    {
        mc = c;
    }
};

int main()
{
    child chd(10,20,30);

    A *pa = &chd;
    B *pb = &chd;
    cout <<"pa->getA = "<<pa->getA()<<endl;
    cout <<"pb->getB = "<<pb->getB()<<endl;
    return 0;
}
```

这种子类地址传给父类，指明了父类 A 地址

这种子类地址传给父类，指明了父类 B 地址

```
root@ubuntu:/home,
pa->getA = 10
pb->getB = 20
root@ubuntu:/home
```

输出的是 A 类和 B 类的函数

```
int main()
{
    child chd(10,20,30);

    A *pa = &chd;
    B *pb = &chd;

    void *paa = pa;
    void *pbb = pb;

    if(paa == pbb)
        cout <<"paa == pbb"<<endl;
    else
    {
        cout <<"error"<<endl;
    }

    return 0;
}
```

但是 pa, pb 两个指针所得到的地址是不同的，虽然都是指向子类，但是子类继承了两个父类，所以赋值地址的时候根据父类指针名来决定，而不是子类名

```
root@ub
error
root@ub
```

所以地址是不同的。这就是多继承的问题

```

class people
{
    string m_name;
    int m_age;
public:
    people(string name,int age)
    {
        m_name = name;
        m_age = age;
    }
    void print()
    {
        cout <<"age = "<<m_age<<endl;
        cout <<"name = "<<m_name<<endl;
    }
};

class teacher:public people
{
public:
    teacher(string name , int age):people(name,age)
    {

    }
};

class student:public people ←
{
public:
    student(string name , int age):people(name,age)
    {

    }
};

class doctor:public people,public student
{
};

int main()
{
    return 0;
}

```

因为 doctor 是要继承学生类的, 相当于级别比 people 类低, 又因为 student 继承了 people, 所以 student 比 doctor 高。所以孙子类不能继爷爷类, 只能继承父类

```

root@ubuntu:/home/xzz/code/C++/multi# g++ -o multib multib.cpp
multib.cpp:40:7: warning: direct base ‘people’ inaccessible in ‘doctor’ due to ambiguity
class doctor:public people,public student

```

```

class people
{
    string m_name;
    int m_age;
public:
    people(string name,int age)
    {
        m_name = name;
        m_age = age;
    }
    void print()
    {
        cout <<"age = "<<m_age<<endl;
        cout <<"name = "<<m_name<<endl;
    }
};
class teacher:public people ←
{
public:
    teacher(string name , int age):people(name,age)
    {

    }
};

class student:public people ←
{
public:
    student(string name , int age):people(name,age)
    {

    }
};

class doctor:public teacher,public student
{
public:
    doctor(string name , int age):teacher(name,age),student(name,age)
    {

    }
};

int main()
{
    doctor d("xzz",20);
    d.print(); ←
    return 0;
}

```

为什么调用会出错？，因为 doctor 继承了 teacher, student 两个父类，这两个父类又继承了同一个 people 类，这个 print 是在 people 类里面的，所以编译器不知道是调用 teacher 继承的 print 还是 student 继承的 print

```

root@ubuntu:/home/xzz/code/C++/multib2# g++ -o multib2 multib2.cpp
multib2.cpp: In function 'int main()':
multib2.cpp:51:4: error: request for member 'print' is ambiguous
    d.print();
    ^

```

解决方案就是指明类作用域

```

int main()
{
    doctor d("xzz",20);
    d.teacher::print();
    d.student::print();
    return 0;
}

```

```

root@ubuntu:/home/xzz/code/C++/multib2# ./multib2
age = 20
name = xzz
age = 20
name = xzz

```

所以 C++多重继承并不好用

C++变量强制类型转换和C语言不一样

```
int main()
{
    int v = 123456;
    char c = char(v);
    printf("v = %d\n",v);
    printf("c = %d\n", (int)c);
    return 0;
}
```

root@ubuntu:~# ./trans1
v = 123456
c = 64
root@ubuntu:~#

C 语言把长地址的 int 强制转换成 char, 就会出现数据丢失

```
struct point
{
    int x=10;
    int y=20;
};

int main()
{
    int addr = 0x123456;
    point *p = (point*)addr;
    printf("p->x = %d\n",p->x);
    printf("p->y = %d\n",p->y);
    return 0;
}
```

root@ubuntu:/home/xzz/code/C++/trans# g++ -o trans2 trans2.cpp
trans2.cpp:5:8: warning: non-static data member initializers only available with -std=c++11 or -std=gnu++11
 int x=10;
 ^
trans2.cpp:6:8: warning: non-static data member initializers only available with -std=c++11 or -std=gnu++11
 int y=20;
 ^
trans2.cpp: In function ‘int main()’:
trans2.cpp:12:21: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
 point *p = (point*)addr;
root@ubuntu:/home/xzz/code/C++/trans#
*Segmentation fault (core dumped)

运行段错误

static_cast 用于基本数据类型的转换，比如 `char, int, long, double`。不能用于指针类型转换，比如`*char, *int, *long, *double`。

```
#include<stdio.h>

int main()
{
    int i = 123456;
    char c = 'c';
    c = static_cast<char>(i);
    printf("c = %d\n",c);
    return 0;
}
```

要转换成什么类型
要转换的变量
直接将 int 转换成 char
型是没有问题的，只是
数据少了一些

root@ub
c = 64
root@ub

```

int addr = 0x1234;
int *pi = &addr;
char *pc;
pc = static_cast<char>(pi);
return 0;

trans3.cpp: In function 'int main()':
trans3.cpp:11:27: error: invalid static_cast from type 'int*' to type 'char'
    pc = static_cast<char>(pi);

```

static_cast 只能传递类型，不能传递指针，传递指针要用其它关键字

static_cast 到底有什么用？只是为了转换吗？

```

int main()
{
    double a = 1.99;
    int b = static_cast<double>(a);
    printf("b = %d\n", b);
    return 0;
}

```

当编译器隐式执行类型转换时，大多数的编译器都会给出一个警告：这样可以让使用的程序员指定
warning C4244：“初始化”：从“double”转换到“int”，可能丢失数据

但是 linux 的 g++ 编译器没有给出提示，跟编译器有关。其实 static_cast 就是让程序员知道有这么个事

```

root@ubt
b = 1
root@ubt你看 a = 1.99 给 b 后精度丢失了。

```

把精度大的(double 64 位)类型转换为精度小的(int 32 位)类型，static_cast 使用位截断进行处理。

- 使用 static_cast 可以找回存放在 void* 指针中的值。

```

int main()
{
    double a = 1.99;
    void *vptr = &a;//将double变量的地址赋值给可以存放任何类型地址的void* 变量
    double *dptr = static_cast<double *>(vptr);//用cast将void*变量里面存放的double变量地址取出来给另外一个相对于的double变量
    printf("*dptr = %f\n", *dptr);
    return 0;
}

```

```

root@ubuntu:/home/x
*dptr = 1.990000
root@ubuntu:/home/x

```

指针和指针类型进行强制转换用 const_cast

通过 const_cast 运算符，也只能将 const type* 转换为 type*，将 const type& 转换为 type&。说白了只去掉了变量的 const 属性，其它的都不变

```

const int a = 5;
int *p;
p = const_cast<int*>(&a);
printf(" *p = %d\n", *p);
printf("a addr = %p\n", &a);
printf("p addr = %p\n", p);
return 0;

```

```

root@ubuntu:/home/xzz/code/
*p = 5
a addr = 0x7ffd3bb1972c
p addr = 0x7ffd3bb1972c
root@ubuntu:/home/xzz/code/

```

我要取消掉变量的 const 属性，直接不定义 const 就行了，何必多此一举呢？

来看下面这种情况

```
class Hero {  
private:  
    std::string name;  
public:  
    Hero(const std::string &name) :name(name) {}  
  
    ~Hero() {}  
  
    std::string GetName() { //非 const 函数  
        return name;  
    }  
};  
  
void PrintHeroName(const Hero &h) {  
    std::cout << h.GetName() << std::endl; //调用了非 const 函数  
}
```

PrintHeroName 函数的形参是一个 const 类型的 Hero 对象，但是在函数体内，这个 const 类型的 Hero 对象调用了一个非 const 的成员函数，这是不允许的。这种情况下，编译器会报错：

```
const_cast.cpp:19:18: error: 'this' argument to member function 'GetName' has type 'const Hero', but funct  
    std::cout << h.GetName() << std::endl;  
          ^  
const_cast.cpp:13:17: note: 'GetName' declared here  
    std::string GetName() {  
          ^  
1 error generated.
```

如果想要在 PrintHeroName 函数中调用 h.GetName 方法，只需要把形参里的 const 去掉就可以了。或者把 GetName() 修改成一个 const 属性的成员函数

其实最开始我也没有发现这段代码的问题，作为亲手写下这段代码的始作俑者，我甚至还捉摸了一段时间。但这其实就是一个简单的用法，因为使用得过于习惯，并且自认为非常熟悉，所以就忽视了问题所在

const_cast 的作用是去掉变量的 const 或者 volatile 限定符——这看起来很鸡肋

```
class Hero {  
private:  
    std::string name;  
public:  
    Hero(const std::string &name) :name(name) {}  
  
    ~Hero() {}  
  
    std::string GetName() {  
        return name;  
    }  
};  
  
void PrintHeroName(Hero &h) { //取消掉了 const  
    std::cout << h.GetName() << std::endl;  
}
```

程序可以正常编译了

问题是，如果调用它的上层函数，Hero 对象已经被 const 限定了，应该怎么办？

```
void PrintHeroName(Hero &h) {  
    std::cout << h.GetName() << std::endl;  
}  
  
void CreateHero(const Hero &h) { //上层函数传入进来的同一个对象地址，有 const 的属性  
    PrintHeroName(h); //上层传入有 const 的对象给没有 const 的 PrintHeroName 函数  
}
```

可以看到，CreateHero 的形参是 const 的，但 PrintHeroName 是非 const 的，编译报错。

这种情况下就要用到 `const_cast` 了，这里才是 `const_cast` 大展拳脚的地方因为我们很明确能知道 `GetName()` 是不会修改任何成员对象的值的，所以可以在这里通过 `const_cast` 去掉 `Hero` 的 `const` 限定，使得程序可以正常往下调用。即，只要把 `h` 通过 `const_cast` 包裹起来就可以了。

```
void CreateHero(const Hero &h) {  
    PrintHeroName(const_cast<Hero &>(h));  
}
```

这就是一个类里面有些函数可能成员变量会被修改，但是有些函数不会。那么就用 `const_cast` 来去掉 `const` 的限定。

dynamic_cast，就是动态类型转换。如子类和父类之间的类型转换。

```
Scratch()[]  
4  
5 class baseclass  
6 {  
7     public:  
8         baseclass(string str,int age)  
9         {  
10             cout<<"str = "<< str << "    age = "<<age<< endl;  
11         }  
12         void baseprint()  
13         {  
14             cout<<"baseprint()"<<endl;  
15         }  
16 };  
17  
18 class child:public baseclass  
19 {  
20     public:  
21         child():baseclass("xzz",20) //如果父类有了构造函数，子类再创建自己的构造函数就需要同时初始化父类的构造函数  
22         {  
23             cout<<"child"<<endl;  
24         }  
25         virtual void print();  
26 };  
27  
28 void child::print()  
29 {  
30     cout<<"child:: print()"<<endl;  
31 }  
32  
33 int main()  
34 {  
35     baseclass *base = new child();  
36     base->baseprint();  
37     return 0;  
38 }
```

因为子类继承父类，创建子类的时候就申请了父类空间和子类空间

baseclass 空间
child 空间
内存模型

str = XZZ age = 20
child
baseprint()

因为接受父子类空间首地址的是父类指针，所以 `base` 只能操作父类空间

baseclass 空间
child 空间
内存模型

```
int main()  
{  
    baseclass *base = new child();  
    base->baseprint();  
    base->child::print();  
    return 0;  
}
```

虽然 `base` 有了子类的地址，但是也无法操作子类，所以 C++ 父类不能调用子类函数

```
root@ubuntu:/home/xzz/code/C++/trans# g++ -o trans4 trans4.cpp  
trans4.cpp: In function ‘int main()’:  
trans4.cpp:37:15: error: ‘child’ is not a base of ‘baseclass’  
    base->child::print();
```

```

int main()
{
    baseclass *base = new child();
    base->baseprint();

    child *chd = base;
    return 0;
}

```

如果我把 base 里面的子类首地址赋值给子类指针呢? (因为 base 父类指针接受的是 new child 子类地址), 本来是可以的, 但是需要强转, 因为编译器表面上以为是两个不同的类相互通赋值, 所以不准编译过

```

root@ubuntu:/home/xzz/code/C++/trans# g++ -o trans4 trans4.cpp
trans4.cpp: In function ‘int main()’:
trans4.cpp:38:15: error: invalid conversion from ‘baseclass*’ to ‘child*’ [-fpermissive]
    child *chd = base;
              ^

```

```

int main()
{
    baseclass *base = new child();
    base->baseprint();

    child *chd = (child *)base;
    chd->print();
    return 0;
}

```

用 C 语言的强转方式, 告诉编译器 base 里面的子类首地址, 可以强转给子类

```

root@ubuntu:/home/xzz/code/C++/tr
str = xzz    age = 20
child
baseprint()
child:: print()

```

这样其实没有问题, C 语言写法强转类是没有问题的

```

int main()
{
    baseclass *base = new child();
    base->baseprint();

    child *chd = (child *)base;
    chd->print();

    delete base;
    delete chd;
    return 0;
}

```

因为 base 和 chd 都是指向子类的首地址, 所以这里是多释放了一次

```

root@ubuntu:/home/xzz/code/C++/trans# ./trans4
str = xzz    age = 20
child
baseprint()
child:: print()
*** Error in `./trans4': double free or corruption (fasttop): 0x8000000000ed4c20 ***
=====
Backtrace:
=====
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7fdc38b217e5]
/lib/x86_64-linux-gnu/libc.so.6(+0xb037a)[0x7fdc38b2a37a]
/lib/x86_64-linux-gnu/libc.so.6(cfree+0x4c)[0x7fdc38b2e35c]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7fdc38aca830]
./trans4[0x400c9]
=====
Memory map:
=====
00400000-00402000 r-xp 00000000 08:01 546488
00601000-00602000 r-p 00001000 08:01 546488
00602000-00603000 r--p 00002000 08:01 546488

```

```

int main()
{
    baseclass *base = new child();
    base->baseprint();

    child *chd = (child *)base;
    chd->print();

    delete base;
    return 0;
}

```

释放一次就可以了

```

root@ubuntu:/home/xzz/code/C++/tr
str = xzz    age = 20
child
baseprint()
child:: print()

```

```

int main()
{
    baseclass *base = new child();
    base->baseprint();

    child *chd = (child *)base;
    chd->print();

    delete chd;
    return 0;
}

```

root@ubuntu:/home/xzz/code/C++/trans# ./trans
str = xzz age = 20
child
baseprint()
child:: print()

但是如果遇到强制转换的类结构和接受地址的类结构不一致就会出问题

```

int main()
{
    baseclass *base = new baseclass("baseclass",50);
    base->baseprint();

    return 0;
}

```

root@ubuntu:/home/xzz/code/C++/trans# ./trans
str = baseclass age = 50
baseprint()

只操作父类没有问题

```

int main()
{
    baseclass *base = new baseclass("baseclass",50);
    base->baseprint();

    child *chd = (child *)base;
    chd->print();
    return 0;
}

```

root@ubuntu:/home/xzz/code/C++/trans# ./trans
str = baseclass age = 50
baseprint()
Segmentation fault (core dumped)

所以编译不会出错，运行时候段错误。

这种 C 语言的强转转换方式程序员很容易在看代码的时候看走眼，编译又不会报错，所以导致了运行 bug

```

int main()
{
    baseclass *base = new baseclass("baseclass",50);
    base->baseprint();

    child *chd = dynamic_cast<child*>(base);
    chd->print();
    return 0;
}

```

root@ubuntu:/home/xzz/code/C++/trans# g++ -o trans4 trans4.cpp
trans4.cpp: In function 'int main()':
trans4.cpp:43:40: error: cannot dynamic_cast 'base' (of type 'class baseclass') to type 'class child*' (source type is not polymorphic)
 child *chd = dynamic_cast<child*>(base);

用 `dynamic_cast` 来做类强制转换的检查，如果有问题就直接让你编译不过。所以你程序员再怎么眼花也

能找到问题。

```
int main()
{
    child *base = new child();
    child *chd = dynamic_cast<child*>(base);
    chd->print();
    return 0;
}
```

```
root@ubuntu:/home/xzz/cod
str = xzz    age = 20
child
child:: print()
```

这样就只有符合强制条件的类才能编译通过，减轻代码错误负担

reinterpret_cast

```
#include<iostream>
using namespace std;

int main()
{
    char* y = "ibcd";
    unsigned char *x = reinterpret_cast<unsigned char *>(y);
    cout<<x<<endl;
    return 0;
}
```

reinterpret_cast<unsigned char *>(y) = 将y变量的ibcd字符串转换成二级制0x009e7898
x * = 0x009e7898

再将这个二进制赋值给 x*

```
root@ubuntu:/home/xzz/code/C++/zhuanhuan# g++ -o conversion conversion.cpp
conversion.cpp: In function ‘int main()’:
conversion.cpp:6:12: warning: deprecated conversion from string constant to ‘char*’ [-Wwrite-strings]
    char* y = "ibcd";
               ^
root@ubuntu:/home/xzz/code/C++/zhuanhuan# ./conversion
i
```

所以字符串 i 还是可以被取出来，但是这里的警告是什么呢？

原来 g++ 编译器 **char *背后的含义是：给我个字符串，我要修改它**

而我们给 y 传递的字符串是初始化写好的，无法修改

所以说，比较合理的办法是**把参数类型修改为 const char ***

这个类型说背后的含义是：**给我个字符串，我只要读取它。**

但是我们要用 reinterpret_cast 转换，所以不能再 char* 前加 const

函数模板解决函数重载这种体力活问题

```
5 #define SWAP(t,a,b) | \
6 |   do | \
7 |   { | \
8 |   |   t c=a; | \
9 |   |   a=b; | \
|   |   b=c; | \
|   } | \
|   while(0)| \
2
3
4 int main()
5 {
6     int a=0;
7     int b=1;
8     SWAP(int ,a,b);
9     cout<<"a = "<<a<<endl;
0     cout<<"b = "<<b<<endl;
1     return 0;
2 }
```

用宏来做交换功能的
函数，没有问题
但是，我们知道 C++
编译器在编译的时候
宏是被展开了的，所
以不安全

```
root@ubuntu:~# ./a.out
a = 1
b = 0
root@ubuntu:~#
```

```
1 void swap(int &a,int &b)
2 {
3     int c = a;
4     a = b;
5     b = c;
6 }
7
8 int main()
9 {
10     int a=0;
11     int b=1;
12     swap(a,b);
13     cout<<"a = "<<a<<endl;
14     cout<<"b = "<<b<<endl;
15     return 0;
16 }
```

用函数重载的方式就可
以很安全的交换变量

```
root@ubuntu:~# ./a.out
a = 1
b = 0
root@ubuntu:~#
```

```
1 void swap(int &a,int &b)
2 {
3     int c = a;
4     a = b;
5     b = c;
6 }
7
8 void swap(double &d1,double &d2)
9 {
10     double d = d1;
11     d1=d2;
12     d2=d;
13 }
14
15 int main()
16 {
17     int a=0;
18     int b=1;
19     swap(a,b);
20     cout<<"a = "<<a<<endl;
21     cout<<"b = "<<b<<endl;
22     double dou1=100;
23     double dou2=200;
24     swap(dou1,dou2);
25     cout<<"dou1 = "<<dou1<<endl;
26     cout<<"dou2 = "<<dou2<<endl;
27     return 0;
28 }
```

但是如果我要交换 double
就要多定义一个相同的函数
重载。那么我要交换 String,
float, char 岂不是要重复定
义很多个同样逻辑的函数？

```
root@ubuntu:/home/wangyifan/CLionProjects/untitled1/
a = 1
b = 0
dou1 = 200
dou2 = 100
root@ubuntu:/home/wangyifan/CLionProjects/untitled1/
```

下面我们使用函数模板来解决这个问题

```
template<typename T>
void Swap(T& a, T& b)
{
    T t = a;
    a = b;
    b = t;
}
```

函数模板就是这里用 template 关键字定义个通用的数据类型，比如我写为 T

把这个 T 当做数据类型，定义变量

```
#include<iostream>
#include <string>
using namespace std;

template < typename T>
void swap(T& a,T& b)
{
    T c = a;
    a = b;
    b = c;
}

int main()
{
    int a=0;
    int b=1;
    swap(a,b);
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    double dou1=100;
    double dou2=200;
    swap(dou1,dou2);
    cout<<"dou1 = "<<dou1<<endl;
    cout<<"dou2 = "<<dou2<<endl;
    return 0;
}
```

然后我们传入的变量是 int, 那么 T 就是 int

我们传入的变量是 double, 那么 T 就自动变为 double

```
root@ubuntu:/home/xzz/code/C++/template# g++ -o template4 template4.cpp
template4.cpp: In function ‘int main()’:
template4.cpp:19:10: error: call of overloaded ‘swap(int&, int&)’ is ambiguous
    swap(a,b);
           ^
template4.cpp:7:6: note: candidate: void swap(T&, T&) [with T = int]
void swap(T& a,T& b)
           ^
In file included from /usr/include/c++/5/bits/stl_pair.h:59:0,
                 from /usr/include/c++/5/bits/stl_algobase.h:64,
                 from /usr/include/c++/5/bits/char_traits.h:39,
                 from /usr/include/c++/5/ios:40,
                 from /usr/include/c++/5/ostream:38,
                 from /usr/include/c++/5/iostream:39,
                 from template4.cpp:1:
/usr/include/c++/5/bits/nove.h:176:5: note: candidate: void std::swap(_Tp&, _Tp&)
swap(_Tp& __a, _Tp& __b);
           ^
template4.cpp:24:17: error: ‘swapx’ was not declared in this scope
    swapx(dou1,dou2);
           ^
```

但是编译的时候发生错误：

不能编译运行的原因是 swap 函数与标准库的 std::swap 模板函数冲突。属于调用疏忽

所以只有换一个和 std 标准库 swap 函数不一样的名字就可以了

```

5 template < typename T>
6 void swapx(T& a,T& b)
7 {
8     T c = a;
9     a = b;
10    b = c;
11 }
12
13
14
15 int main()
16 {
17     int a=0;
18     int b=1;
19     swapx(a,b);
20     cout<<"a = "<<a<<endl;
21     cout<<"b = "<<b<<endl;
22     double dou1=100;
23     double dou2=200;
24     swapx(dou1,dou2);
25     cout<<"dou1 = "<<dou1<<endl;
26     cout<<"dou2 = "<<dou2<<endl;
27     return 0;
28 }

```

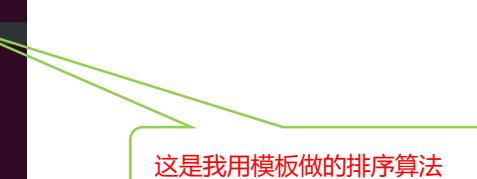
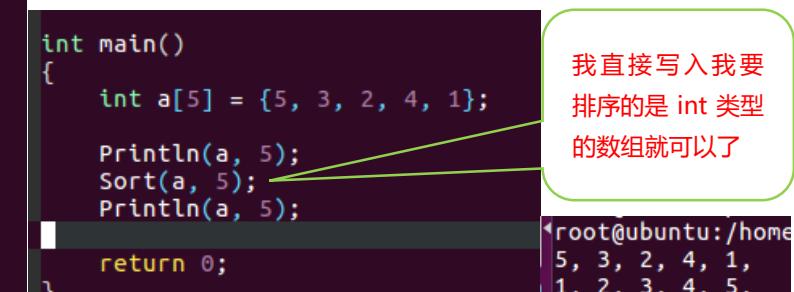
root@ubuntu:/home/xzz/code/C++# ./56

```

a = 1
b = 0
dou1 = 200
dou2 = 100
root@ubuntu:/home/xzz/code/C++#

```

我们下面用模板技术做个排序算法

```

template < typename T >
void Swap(T& a, T& b)
{
    T c = a;
    a = b;
    b = c;
}

template < typename T >
void Sort(T a[], int len)
{
    for(int i=0; i<len; i++)
    {
        for(int j=i; j<len; j++)
        {
            if( a[i] > a[j] )
            {
                Swap(a[i], a[j]);
            }
        }
    }
}

template < typename T >
void println(T a[], int len)
{
    for(int i=0; i<len; i++)
    {
        cout << a[i] << ", ";
    }
    cout << endl;
}

```

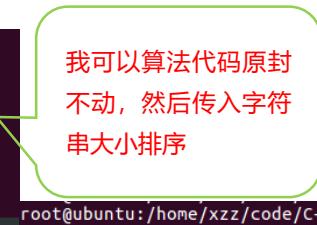
root@ubuntu:/home/xzz/code/C++# ./56

```

int main()
{
    int a[5] = {5, 3, 2, 4, 1};
    println(a, 5);
    Sort(a, 5);
    println(a, 5);
    return 0;
}

5, 3, 2, 4, 1,
1, 2, 3, 4, 5,
root@ubuntu:/home/xzz/code/C++#

```



```

int main()
{
    string s[5] = {"Java", "C++", "Pascal", "Ruby", "Basic"};
    println(s, 5);
    Sort(s, 5);
    println(s, 5);
    return 0;
}

```

root@ubuntu:/home/xzz/code/C++# ./56

```

Java, C++, Pascal, Ruby, Basic,
Basic, C++, Java, Pascal, Ruby,
root@ubuntu:/home/xzz/code/C++#

```

```

template < typename T>
void swapx(T& a,T& b)
{
    T c = a;
    a = b;
    b = c;
}

typedef void(funci)(int &,int &); ←
typedef void(funcd)(double &,double &); ←

int main()
{
    funci* pi = swapx; //编译器根据函数指针的类型自动推导出T为int
    funcd* pd = swapx; //编译器根据函数指针的类型自动推导出T为double

    cout << "pi = " << reinterpret_cast<void*>(pi)<<endl;
    cout << "pd = " << reinterpret_cast<void*>(pd)<<endl;
    return 0;
}

```

在将同一个函数模板赋值给函数指针的时候，编译器就悄悄的给这个 pi 指针创建了一个地址来指向

编译器根据函数指针的类型自动推导出T为int

编译器根据函数指针的类型自动推导出T为double

编译器悄悄再创建了一个地址给 pd 指向

```

root@ubuntu:/home
pi = 0x400964
pd = 0x400991

```

所以虽然是用的同一个函数模板，但是创建了两个相同函数的地址。

```

class test
{
public:
    test()
    {
        cout <<"test class"<<endl;
    }
};

template < typename T>
void swapx(T& a,T& b)
{
    T c = a;
    a = b;
    b = c;
}

typedef void(funci)(int &,int &);
typedef void(funcd)(double &,double &);
typedef void(funcnt)(test &,test&);

int main()
{
    funci* pi = swapx;
    funcd* pd = swapx;
    funct* pt = swapx; |
    cout << "pi = " << reinterpret_cast<void*>(pi)<<endl;
    cout << "pd = " << reinterpret_cast<void*>(pd)<<endl;
    cout << "pt = " << reinterpret_cast<void*>(pt)<<endl;
    return 0;
}

```

函数模板也可以传入类变量，在模板里面交换两个类里面的数据

```

root@ubuntu:/home
pi = 0x400a0a
pd = 0x400a37
pt = 0x400a70

```

多参数函数模板使用

```
template <typename T1 , typename T2, typename T3>
T1 add(T2 a,T3 b)
{
    return static_cast<T1>(a+b);
}

int main()
{
    int ret = add<int,float,float>(5.5,5.5);
    cout<<"ret = "<< ret << endl;
    return 0;
}
```

我们可以定义多个模板参数

在返回的时候为了避免运算结果可能是整数, 也可能
是小数, 所以我们直接全部
强制转换成整形返回

```
root@ubuntu:/|
ret = 11
root@ubuntu:/|
```

第一个 int 是返回值类型,
第 2 个 float 和第 3 个 float
代表 a 变量和 b 变量

函数模板除了返回值要确定数据类型外, 传入的形参可以自动推导, 不需要写明数据类型

```
template <typename T1 , typename T2, typename T3>
T1 add(T2 a,T3 b)
{
    return static_cast<T1>(a+b);
}

int main()
{
    int ret = add<int>(10,9.5);
    cout<<"ret = "<< ret << endl;
    return 0;
}
```

我只写返回类型, 后
面传入的参数不写类
型, 根据数值编译器
自动推导出类型

```
root@ubunt
ret = 19
root@ubunt
```

```
4 template <typename T1 , typename T2, typename T3>
5
6 T1 add(T2 a,T3 b)
7 {
8     return static_cast<T1>(a+b);
9 }
10 int main()
11 {
12     int ret = add<int,int>(20,9.5);
13     cout<<"ret = "<< ret << endl;
14     return 0;
15 }
```

返回值类型必须写,
然后可以写一个形参
类型, b 变量类型自动
推导

```
root@ubuntu
ret = 29
root@ubuntu
```

函数模板遇到函数重载会出问题吗？

```
template <typename T1>
T1 Max(T1 a,T1 b)
{
    cout<<"max T1 a = "<<a<<endl;
    cout<<"max T1 b = "<<b<<endl;
    return a+b;
}

int Max(int a,int b)
{
    cout <<"a+b = "<<a+b<<endl;
    return a+b;
}

int main()
{
    int ret = Max(10,20);
    cout<<"ret = "<<ret<<endl;
    return 0;
}
```

函数模板与函数重载冲突的时候，程序会首先看使用的函数是不是和函数重载一致，如果一致优先调用函数重载

```
root@ubuntu
a+b = 30
ret = 30
root@ubuntu
```

```
template <typename T1>
T1 Max(T1 a,T1 b)
{
    cout<<"max T1 a = "<<a<<endl;
    cout<<"max T1 b = "<<b<<endl;
    return a+b;
}

int Max(int a,int b)
{
    cout <<"a+b = "<<a+b<<endl;
    return a+b;
}

int main()
{
    int ret = Max<int>(100,200);
    cout <<"ret = "<<ret<<endl;
    return 0;
}
```

如果用<>指明了函数模板，就会优先调用函数模板

```
root@ubuntu:/home
max T1 a = 100
max T1 b = 200
ret = 300
root@ubuntu:/home
```

```

template <typename T1 >
T1 Max(T1 a,T1 b)
{
    cout<<"max T1 a = "<<a<<endl;
    cout<<"max T1 b = "<<b<<endl;
    return a+b;
}

template <typename T2 >
T2 Max(T2 a,T2 b,T2 c)
{
    cout<<"a + b + c"<<endl;
    return a+b+c;
}
int Max(int a,int b)
{
    cout <<"a+b = "<<a+b<<endl;
    return a+b;
}

int main()
{
    int ret = Max(10,10,10);
    cout<<"ret = "<<ret<<endl;
    return 0;
}

```

如果什么都不指明，但是根据参数，发现函数重载里面只有一个函数模板有3个形参，也会调用函数模板

```
root@ubuntut0:/home
a + b + c
ret = 30
```

如果有函数重载有3个形参，就会放弃3个形参的函数模板，转而去调用3个形参的函数重载。

函数模板就是让我们编写一些需要与类型无关的函数，模板带来了更高的抽象方法，抽象代码。

类模板使用

类模板主要用于数据结构，比如链表，队列，栈。

比如链表的算法是固定的，但是链表里面存放的变量类型，可以是各种类型的，而不是固定类型。

- 在类声明前使用 **template** 进行标识
- < **typename T** > 用于说明类中使用的**泛指类型 T**

类模板的定义和函数模板一样，定义一个 T 来接受任何变量类型

```

template < typename T >
class Operator
{
public:
    T op(T a, T b);
};
```

```

using namespace std;

template <typename T>
class opt
{
public:
    T add(T a, T b)
    {
        return a + b;
    }
    T minus(T a, T b)
    {
        return a - b;
    }
    T multiply(T a, T b)
    {
        return a * b;
    }
    T divide(T a, T b)
    {
        return a / b;
    }
};

int main()
{
    opt<int> optobj;
    cout<<optobj.add(10,20)<<endl;
    return 0;
}

```

我们在类里面进行任何运算，这个 T 就类似与宏，主函数传入什么变量类型，这个 T 就是什么变量类型

函数模板可以根据传入形参的值来自动推导这个值是什么数据类型，但是类模板必须在定义对象的时候事先声明数据类型，然后这个对象就只能操作声明的数据类型

root@ub:~# 30
root@ub:~# 结果正常

```

5 template <typename T>
6 class opt
7 {
8 public:
9     T add(T a, T b)
10    {
11        return a + b;
12    }
13    T minus(T a, T b)
14    {
15        return a - b;
16    }
17    T multiply(T a, T b)
18    {
19        return a * b;
20    }
21    T divide(T a, T b)
22    {
23        return a / b;
24    }
25};

26 int main()
27 {
28     opt<string> optobj;
29     cout<<optobj.add("xxxxx","zzzzz")<<endl;
30     return 0;
31 }

```

我们可以修改成字符串类型，然后两个字符串相加，没有问题

C++两个 string 相加就相当于两个 string 拼接

root@ubuntu:/h
xxxxxzzzzz
root@ubuntu:/h

```

int main()
{
    opt<string> optobj;
    cout<<optobj.minus("xxxxx","zzzzz")<<endl;
    return 0;
}

/usr/include/c++/5/bits/stl_iterator.h:916:5: note:
classtem3.cpp:15:12: note:   'std::__cxx11::basic_st
    return a - b;
    ^
root@ubuntu:/home/xzz/code/C++/classTemp#
```

两个 string 相减 C++ 直接编译报错，因为 C++ 不允许 string 相减

注意：工程应用中类模板一般在头文件里面定义，类模板不能分开实现在不同文件中。

所以类模板的定义和函数实现都在同一个文件

```

root@ubuntu:/home/xzz/code/C++/cla
classtemproj.cpp  classtemproj.h
```

定义了一个 cpp 主函数文件，和 h 头文件

classtemproj.h 文件内部

```

#ifndef __CLASSTEMPROJ_H
#define __CLASSTEMPROJ_H

template <typename T>
class opt
{
public:
    T add(T a, T b);
    T minus(T a, T b);
    T multiply(T a, T b);
    T divide(T a, T b);
};

template <typename T>
T opt<T>::add(T a ,T b)
{
    return a + b;
}

template <typename T>
T opt<T>::minus(T a, T b)
{
    return a - b;
}

template <typename T>
T opt<T>::multiply(T a, T b)
{
    return a * b;
}

template <typename T>
T opt<T>::divide(T a, T b)
{
    return a / b;
}
```

头文件定义类为类模板，所以要在类前面声明

类模板，类声明的函数也必须在头文件中实现，每个实现函数前面都要加 template

注意每个实现函数 T 的定义位置

classtemproj.cpp 文件内部

```
#include<iostream>
#include<string>
#include "classtemproj.h"
using namespace std;

int main()
{
    opt<int> optobj1;
    cout<<optobj1.add(100,200)<<endl;

    opt<string> optobj2;
    cout<<optobj2.add("xxxxx","zzzzz")<<endl;
    return 0;
}

root@ubuntu:/home/xzz/code/C++/classtemproj# g++ -o classtemproj classtemproj.cpp
root@ubuntu:/home/xzz/code/C++/classtemproj# ./classtemproj
300
xxxxxzzzzz
```

包含使用的类
模板头文件

创建类模板对象，直接调用类模
板的函数就是

直接编译主函数 cpp 文件，运行就是。所以类模板的好处就是少写一个对应 h 文件的 cpp 文件

为什么要在实现类的前面加template，而且还要在实现类的时候，在类里面加template？

```
template <typename T> // 1. 定义了一个模板类
class People
{
private:
    T age;
public:
    People(T a); // 构造函数要交给其它文件去具体实现，那么就不需要{}
    ~People(){} // 析构函数不需要其它文件实现，需要{}
    void print(T x);
};
```

3. 而且在类前面加模板名还不说，在实现类中
间也要加一样的模板名

```
template <typename T>
People<T>::People(T a) // 构造函数初始化
{
    age = a;
}
```

2. 其它文件实现类的时
候，为什么要在前面加
模板名

```
template <typename T>
void People<T>::print(T x)
{
    cout << "x = " << x << endl;
    cout << "age = " << age << endl;
}
```

```
template <typename T>
People<T>::People(T a) // 构造函数初始化
{
    age = a; // 4. 类名里面的<T>，是  
        // 要告诉别人我这是一个  
        // 模板类。
}

template <typename T> // 5. 类函数实现前  
void People<T>::print(T x) // 面加template是  
// 想给编译器说，  
// 下面这段函数实  
// 现的是模板类。
{
    cout << "x = " << x << endl;
    cout << "age = " << age << endl;
}
```

```
int main(void)
{
    People<int> p1(4);
    p1.print(23);

    return 0;
}
```

x = 23
age = 4

多参数模板类实现

```
template <typename T1, typename T2> ← 1. 如果模板类定义了2个,  
class People  
{  
private:  
    T1 age;  
public:  
    People(T1 a); //构造函数要交给其它文件去具体实现, 那么就不需要{}  
    ~People(){} //析构函数不需要其它文件实现, 需要{}  
    void func(T1 x, T2 y); ← 2. 实现多参数函数  
}; 3. 在其它文件, 进行类实现的时候, 模板定义也必须使用多参数  
template <typename T1, typename T2> ← 4. 在类里面也必须是多参数  
People<T1, T2>::People(T1 a) //构造函数初始化  
{  
    age = a;  
}  
  
template <typename T1, typename T2>  
void People<T1, T2>::func(T1 x, T2 y)  
{  
    cout << "x = " << x << endl;  
    cout << "y = " << y << endl;  
} 5. 当多参数模板类定义  
对象的时候, 必须按照  
<...>指定的变量类型  
顺序填入。  
  
int main(void)  
{  
    People<string, int> p1("Init");  
    p1.func("xzz", 50); ← 6. 我是字符串在前, 数字在  
    return 0; 后, 那么传参也必须是字符串  
}  
  
x = xzz  
y = 50
```

模板类实现运算符重载

```
template <typename T>  
class People  
{  
private:  
    T age;  
public:  
    People();  
    //模板类做操作符重载, 原始的构造函数不能取消, 因为在进行加减运算的时候, 使用的是元素构造函数。  
    People(T a):age(a){}; //创建对象的时候给age初始化值  
    ~People();  
  
    //实现加法运算符重载  
    People<T> operator+(People<T> &other);  
    void print(void); //打印加法计算之后的值  
};
```

```

//加法运算符重载实现
template <typename T>
People<T> People<T>::operator+(People<T> &other)
{
    People<T> tmp; //定义一个对象，实际调用的是People(T a){}
    tmp.age = this->age + other.age;
    return tmp; //返回的是值，也就是值的赋值，而不是返回对象的地址。
}

this是调用该函数的当前对象 + 传入的新对象 = 值，返回给外部对象

template <typename T>
void People<T>::print(void) //专用打印对象结果的函数
{
    cout << "age = " << this->age << endl;
}

int main(void)
{
    People<int> a(4); //初始化的时候使用的是有参构造函数
    People<int> b(6);
    People<int> c(0);
    ↗两个对象里面的age相加
    c = a + b; //进行对象加减的时候，使用的是原始构造函数
    c.print();
    return 0;
}

age = 10

```

```

template <typename T>
class People
{
private:
    T age;
public:
    People(); //模板类做操作符重载，原始的构造函数不能取消，因为
    →People(T a):age(a){}; //创建对象的时候给age初
    ~People();

    //实现加法运算符重载
    People<T> operator+(People<T> &other);
    void print(void); //打印加法计算之后的值
};

```

模板类继承

单模板继承方式如下：

```

template <typename T>
class People
{
private:
public:
    T x;
    People(){}
    People(T a):x(a){} //经过a给x初始化
    ~People(){}
};

template <typename T>
class Man : public People<T> {
private:
    /* data */
public:
    T y;
    Man(){}
    Man(T a):y(a){} //y值初始化
    Man(T a , T b):People<T>(a),y(b){} //这样People传入的a就是初始化父类的x, b就是初始化子类的y
    ↗父类模板名必须有
}

```

```

int main(void)
{
    Man<int> m1(3,7); //a = 3, b = 7, a对应父类模板x, b对应y
    cout << "x = " << m1.x << endl;
    cout << "y = " << m1.y << endl;
    return 0;
}

x = 3
y = 7

```

1. 继承模板和继承类方式相似，但是类名要加模板变量

2. 可以调用父类来初始化父类里面的变量。

3. 向子类初始化数据，可以同时给子类和父类的变量初始数据，只要子类有父类变量的调用。

多模板继承方式如下：

```
template <typename T1 , typename T2>
class People
{
private:
public:
    T1 x1;
    T2 x2;
    People(){}
    People(T1 a , T2 b):x1(a),x2(b){}
    ~People(){}
};
```

```
template <typename T1 , typename T2> ← 多模板继承无非就是两个模板变量
class Man : public People<T1 , T2>
{
private:
    /* data */
public:
    T1 y1;
    T2 y2;
    Man(){}
    Man(T1 a , T2 b):y1(a) , y2(b){} //y值初始化 ← 初始化父类x1, x2
    Man(T1 a1 , T2 b1 , T1 a2 , T2 b2):People<T1 , T2>(b1,b2),y1(a1),y2(a2){}
};
```

↑
多模板父类继承，多个变量

```
int main(void)
{
    Man<int,double> m1(4 , 5.5, 6 , 7.7);
    cout << "x1 = " << m1.x1 << endl;
    cout << "x2 = " << m1.x2 << endl;
    cout << "y1 = " << m1.y1 << endl;
    cout << "y2 = " << m1.y2 << endl;
    return 0;
}
```

x1 = 5
x2 = 7.7
y1 = 4
y2 = 6

仿函数(函数对象)使用

```
class Car { 在类里面，使用operator关键字
public: 重载()小括号，这个类就是带有
        仿函数的仿函数类
    Car(){}
    void operator()()
    {
        cout << "仿函数执行" << endl;
    }
};
```

```
class Car {
public: 仿函数也可以有形参
    Car(){}
    int operator()(int a) //仿函数也可以有形参和函数返回值
    {
        return a * 10;
    }
};
```

```
int main(void)
{
    Car obj; //创建一个对象
    obj(); //对象直接用小括号()，就会直接执行operator()()函数
    return 0;
}
```

仿函数执行

```
int main(void)
{
    Car obj; //创建一个对象
    int b = obj(5); //仿函数形参使用
    cout << "b = " << b << endl;
    return 0;
}
```

b = 50

仿函数使用案例1

下面我们来做一个累加器，每调用一次仿函数对象，数据就自动累加

```
class Accumulator
{
private:
    int internal_sum; //仿函数内部状态
public:
    Accumulator(int init_sum = 0) : internal_sum(init_sum) {} //给类私有变量初始化值
    ~Accumulator() {}

    int operator()(int value) ←———— 当直接使用对象，给对象传值的时候，这个函数就会被执行。
    {
        internal_sum += value ;
        return internal_sum;
    }

    int getCurrentSum() //获取当前内部状态
    {
        return internal_sum;
    }
};

int main(void)
{
    Accumulator acc;           因为对象一直存在，给对象赋值相当于就是执
                                行仿函数
    cout << "acc = " << acc(5) << endl; //第一次输出5
    cout << "acc = " << acc(3) << endl; //第二次怎么输出8了?
    return 0;
}
```

acc = 5
acc = 8

```
int main(void)
{
    Accumulator acc(20);      1. 如果创建对象的时候，先给对象赋值
    cout << "acc = " << acc(5) << endl; //第一次输出5
    cout << "acc = " << acc(3) << endl; //第二次怎么输出8了?
    return 0;                  acc = 25
                                acc = 28
}

private:
    int internal_sum; //仿函数内部状态
public:
    Accumulator(int init_sum = 0) : internal_sum(init_sum) {} //给类私有变量初始化值
    ~Accumulator() {}
```

2. 相当于先让对象类中的私有变量初始为20
3. 然后在20的基础上上去累加私有变量

仿函数案例2，仿函数作形参

```
//定义一个仿函数
class MyFunctor
{
private:
    /* data */
public:
    MyFunctor(){}
    ~MyFunctor(){}

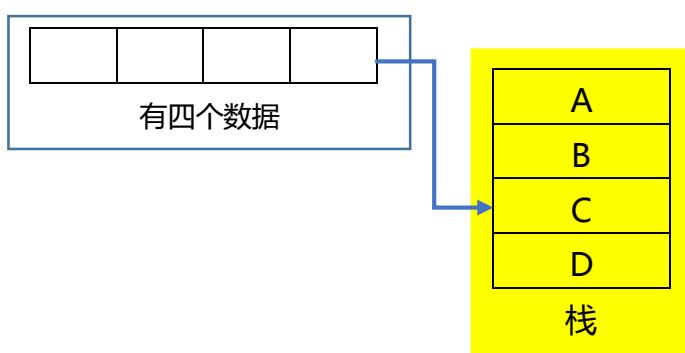
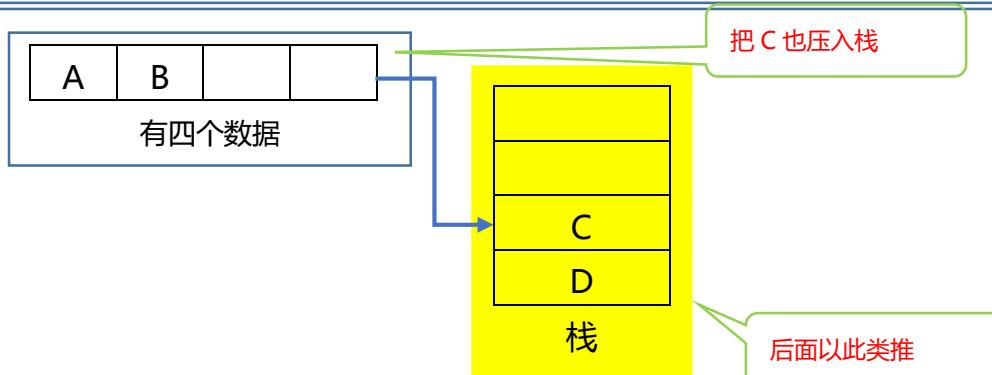
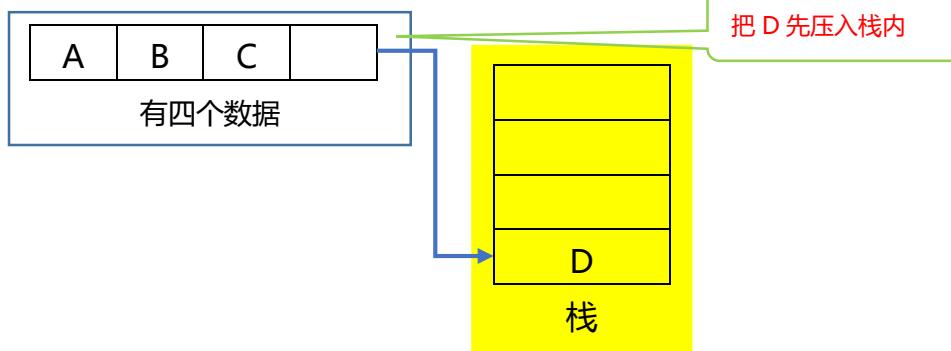
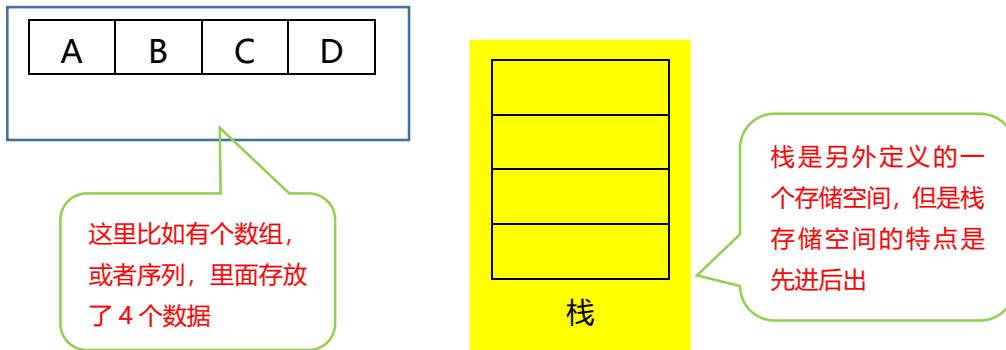
    int operator()(int value)
    {
        return value * 2;
    }
};

void applyfunctor(int Value ,  MyFunctor &functor)
{
    cout << "经过仿函数处理后的值 :" << functor(Value) << endl;
}

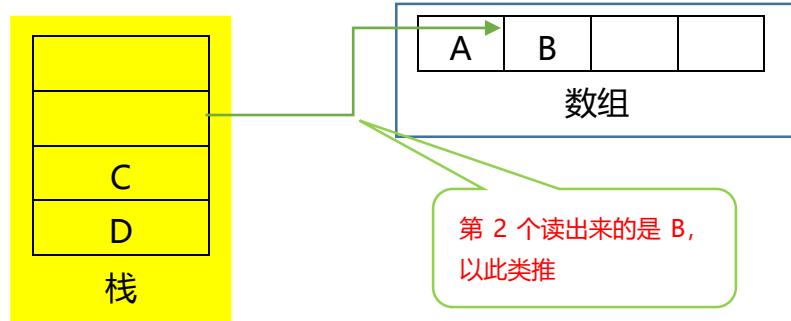
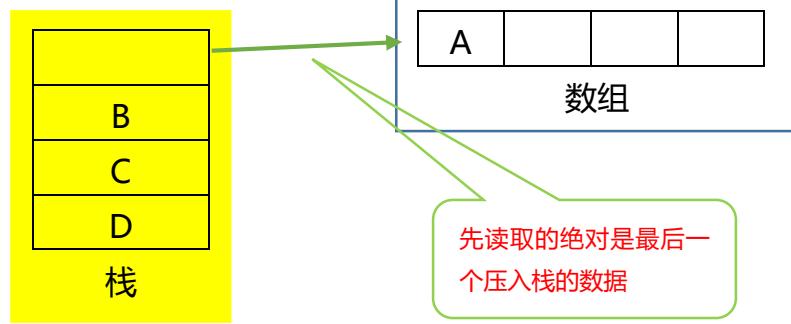
int main(void)
{
    MyFunctor functor;
    applyfunctor(5,functor);
    return 0;
}
```

经过仿函数处理后的值 :10

用类模板做一个栈



现在我有个数组想把栈的数据读取出来



```
#include<iostream>
#include<string>
using namespace std;

template <class T>
class stack
{
public:
    stack(unsigned int size = 100);
    ~stack();
    void push(T value);
    T pop();
private:
    unsigned int size;
    unsigned int sp;
    T *data;
};
```

其实这里不一定非要是 typename, 自己随便起个名字是可以的, 只要能表达模板是干什么的就行

如果创建对象不定义栈大小的话, 我们默认给 100 个空间

将 value 的值写入栈

将 value 的值读出栈

栈大小

栈指针

```

template <class T>
stack<T>::stack(unsigned int size)
{
    this->size = size;
    data = new T[size];
    sp = 0;
}

template <class T>
stack<T>::~stack()
{
    delete []data;
}

template <class T>
void stack<T>::push(T value)
{
    data[sp++] = value;
}

template <class T>
T stack<T>::pop()
{
    return data[--sp];
}

int main()
{
    stack<int> objstack(100);
    objstack.push(10);
    objstack.push(20);
    objstack.push(30);
    objstack.push(40);

    cout<<"pop = "<<objstack.pop()<<endl;
    cout<<"pop = "<<objstack.pop()<<endl;
    cout<<"pop = "<<objstack.pop()<<endl;
    cout<<"pop = "<<objstack.pop()<<endl;
}

```

调用构造函数的时候我们将申请的栈大小传递给私有的 size

根据 T 的数据类型，分配 size 大小个符合 T 类型的数组

sp 变量值就是 data 数组的下标

我们给栈传入值的时候 sp++, 就是确定值放在栈的哪个位置

我们读出数据的时候，确定读取栈哪个位置的值

| |
|------------|
| SP=4, 值 40 |
| sp=3 值 30 |
| sp=2 值 20 |
| sp=1 值 10 |
| sp=0 |

栈

弹出栈的原理也是一样

```

root@ubuntu:/home/xzz/code/C++/stack#
pop = 40
pop = 30
pop = 20
pop = 10

```

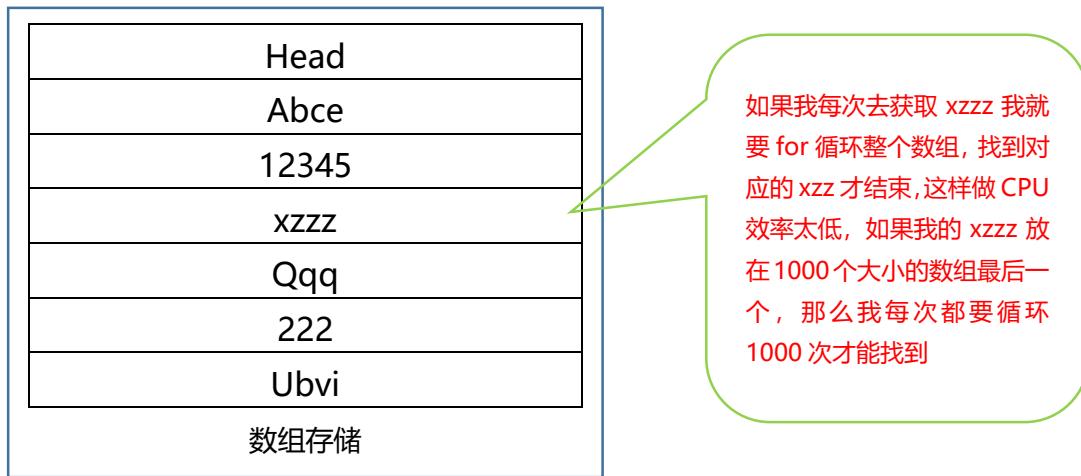
弹栈就是从 sp=4 开始，因为栈是先进后出嘛。

你不可能去给 sp 随意指定数组下标，sp 在 pop 和 push 函数里面自己按照顺序++，--。

容器实现

数组就是容器，容器就是能存放一种数据结构，或者多种数据结构的东西。

但是数组并不能解决所有问题，数组需要规定长度，比如我的数组是用来存放所有人名字的。



上面基于模板实现的 stack 容器也是容器的一种

Vector 向量使用

```
#include<iostream>
#include<vector> //向量类vector需要包含的头文件
using namespace std;

int main()
{
    std::vector<std::string> names;
    names.push_back("xxxxzz");
    names.push_back("12345678");
    names.push_back("abcdefg");

    for(int i = 0;i<names.size();i++)
    {
        cout<<names[i]<<endl;
    }

    return 0;
}
```

定义个 vector 模板类对象，这个对象存放 string 数据

可以随意的向 names 对象放数据，数据存放顺序是挨着排列的，执行一次 push_back 函数，存储地址向后+1

所以读取数据也是挨着数组下标读出来

向量 vector 看着很像数组得存储结构，但是它优化了数组需要实现设置固定长度的弊端。数组这种固定长度有可能不需要存放这么多数据从而造成数组多余空间浪费。Vector 则不会，执行 push_back 一次申请一次内存空间，很好用。

迭代器

我们上面 vector 容器是经过数组下标来访问的，但是如果是 stack 就无法用数组下标来访问，所以像访问这种顺序存储结构的容器，我们最好有一种通用的访问方法。这种通用的方法就是迭代器。

```
2 #include<vector>//向量类vector需要包含的头文件
3 using namespace std;
4
5
6 int main()
7 {
8     std::vector<std::string> names;
9     names.push_back("xxxxxxxx");
10    names.push_back("12345678");
11    names.push_back("abcdefg");
12
13    std::vector<std::string>::iterator iter = names.begin();
14
15    while(iter!=names.end())
16    {
17        cout<<*iter<<endl;
18        ++iter;
19    }
20    return 0;
21 }
```

```
root@ubunt: ~
xxxxxxxx
12345678
abcdefg
root@ubunt:
```

这就是迭代器的访问方式，哎....也是循环类型的顺序访问

```
2 #include<vector>//向量类vector需要包含的头文件
3 #include<algorithm>//算法头文件，我们用里面的排序算法
4 using namespace std;
5
6
7 int main()
8 {
9     std::vector<std::string> names;
10    names.push_back("aaa");
11    names.push_back("123");
12    names.push_back("bbbb");
13    names.push_back("uuuu");
14
15    std::vector<std::string>::iterator iter = names.begin();
16
17    while(iter!=names.end())
18    {
19        cout<<*iter<<endl;
20        ++iter;
21    }
22    cout<<"-----"<<endl;
23    std::sort(names.begin(),names.end());//sort排序函数对names向量里面的内容从新排序
24    iter = names.begin();//迭代器回到names首地址
25    while(iter!=names.end())
26    {
27        cout<<*iter<<endl;
28        ++iter;
29    }
30
31    return 0;
32 }
```

迭代器功能在 vector 里面有现成的，我们定义一个就是了，这里是访问 string 类型的顺序存储变量，所以这里要指定 string

iterator 就是定义迭代器变量 iter

iter 初始化指向存储变量的首地址

指针方式访问就是了

```

root@ubuntu:/home/xzz
aaa
123
bbbb
uuuu
-----
123
aaa
bbbb
uuuu

```

这就是排序后的结果，按照数字，字母顺序排序

类模板方式的智能指针

智能指针的好处在只能指针章节已经说过：

- 1.可以自动释放内存
- 2.防止 delete 释放对象之后，不小心再用 delete 重复释放同名的对象

auto_ptr<类名> 对象名(...new出符合类名的对象) //这就是auto_ptr智能指针模板格式



```

#include<iostream>
#include<string>
#include<memory> //智能指针类模板需要的头文件
using namespace std;

class test
{
    string my_name;
public:
    test(const char* name)
    {
        my_name = name;
    }
    void print()
    {
        cout <<my_name<<".."<<endl;
    }
    ~test()
    {
        cout <<"~test"<<endl;
    }
};

int main()
{
    auto_ptr<test> pt(new test("xxxxxx")); //将新创建的对象地址赋值给智能指针pt
    pt->print(); //智能指针pt就可以操作test类型对象里面的内容
    return 0;
}

```

pt 对象指向



这个 pt 智能指针对象经过<test>对象类型识别，拥有了指向 test 创建的对象地址能力

pt 括号里面就是新创建的对象，然后自动把对象地址赋值给 pt

```

root@ubuntu:
xxxxxx..
~test

```

这就是智能指针，程序执行完后自动释放堆空间内存，注意 pt 变量是对象哦，不是指针。

只是 pt 模板对象里面有存放堆空间地址的功能，所以叫智能指针。

有人觉得是 main 函数执行完了，整个程序都结束了才释放的堆空间，我们再来证明一下，不是这样的

```
1 #include<iostream>
2 #include<string>
3 #include<memory> //智能指针类模板需要的头文件
4 using namespace std;
5
6 class test
7 {
8     string my_name;
9 public:
10     test(const char* name)
11     {
12         my_name = name;
13     }
14     void print()
15     {
16         cout << my_name << "..." << endl;
17     }
18     ~test()
19     {
20         cout << "~test" << endl;
21     }
22 };
23
24 void func()
25 {
26     auto_ptr<test> pt(new test("xxxxzzz")); //将新创建的对象地址赋值给智能指针pt
27     pt->print(); //智能指针pt就可以操作test类型对象里面的内容
28 }
29
30 int main()
31 {
32     func();
33     cout << "-----" << endl;
34
35     return 0;
36 }
```

我把智能指针模板放在子函数里面

```
1 xxxxzzz..
2 ~test
3 -----
4 你看!! 是子函数执行完了，就自动释放堆空间内存了。
```

这就说明了 pt 变量是在栈里面的，pt 智能指针类模板里面有管理堆空间变量释放的函数。所以子函数执行完了，栈自动释放 pt 的同时，也自动释放了堆空间。这就是类模板智能指针的好处。

```

#include<iostream>
#include<string>
#include<memory>
using namespace std;

class test
{
    string my_name;
public:
    test(const char* name)
    {
        my_name = name;
    }
    void print()
    {
        cout << my_name << " . " << endl;
    }
    ~test()
    {
        cout << "~test" << endl;
    }
};

int main()
{
    auto_ptr<test> pt(new test("xxxxzzz"));
    pt->print();

    //智能指针模板可以指向另外一个智能指针，从而获取另外一个智能指针指向的堆空间地址
    auto_ptr<test> pt2(pt);
    pt2->print();
    return 0;
}

```

root@ubun:~# ./a.out
xxxxzzz..
xxxxzzz..
~test

所以我们发现虽然两个智能指针模板都指向同一片堆空间，但是释放的时候只释放一次堆空间

下面我们用 QT 的智能指针来操作试试

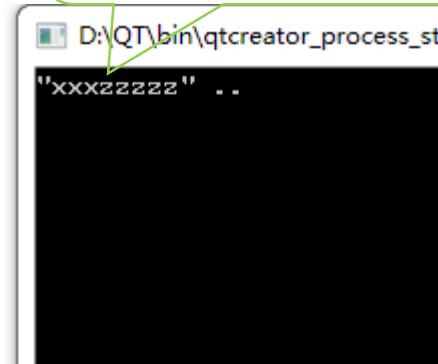
```

#include <QtCore/QCoreApplication>
#include <QPointer> //QT的智能指针类模板要靠这个头文件
#include <QString>
#include <QDebug>
class test : public QObject //QT里面自己创建的类都必须要继承QObject
{
    QString my_name;
public:
    test(QString name)
    {
        my_name = name;
    }
    void print()
    {
        qDebug() << my_name << " . ";
    }
    ~test()
    {
        qDebug() << "~test";
    }
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QPointer<test> pt(new test("xxxxzzzz"));
    //QT的智能指针模板就是QPointer，操作方法和前面讲的智能指针模板一样
    pt->print();
    return a.exec();
}

```

Printf 函数执行了，但是析构函数没有看到执行，到底执行没有我们得修改下这个程序



```

void print()
{
    qDebug() << my_name << "...";
}
~test()
{
    qDebug() << "~test";
}

};

void func() //子函数结束就知道QPointer智能指针释放堆空间没有
{
    QPointer<test> pt(new test("xxxxzzzz"));
    //QT的智能指针模板就是QPointer, 操作方法和前面讲的智能指针模板一样
    pt->print();
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    func();
    qDebug() << "end....";
    return a.exec();
}

```

Qpointer 是智能指针, 为什么没有释放堆空间的能力? 这不符合智能指针的要求啊! 哪里智能了...?

其实 Qpointer 是让 pt 对象在子函数执行完后 pt 指向 NULL。但是 test 对象堆空间还是在那里占用着的

```

#include <QDebug>
#include <QSharedPointer> //这个QT的智能指针类模板是完善QPointer的不足
class test : public QObject //QT里面自己创建的类都必须要继承QObject
{
    QString my_name;
public:
    test(QString name)
    {
        my_name = name;
    }
    void print()
    {
        qDebug() << my_name << "...";
    }
    ~test()
    {
        qDebug() << "~test";
    }
};

void func()
{
    QSharedPointer<test> pt(new test("xxxxzzzz"));
    //将QPointer改成QSharedPointer

    pt->print();
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    func();
    qDebug() << "end....";
    return a.exec();
}

```

从 func 子函数执行完就知道
Qpointer 没有释放堆空间, 如果释放
堆空间是要执行对象的析构函数的

你看子函数退出后析构程序就执行了,
说明了 QSharedPointer 才是真正的
智能指针

```
D:\QT\bin\qtcreator_.p
"xxxxzzzz" ...
end....
```

```
D:\QT\bin\qtcreator_process_stub.p
"xxxxzzzz" ...
~test
end....
```

C++11: override 和 final 关键字使用

在C++98标准当中遇到过如下情况：

```
class A 1. 比如我抽象了一个虚函数
{
public:
    virtual void Func(){}
};

class B : public A
{
    2. B类继承了A类，但是
public:    也定义了一样的虚函数
    virtual void Func(){}
};

class C : public B
{
    3. 这时候，我不想类C继承
public:    类B的时候，又去复写虚函
    numFunc
    virtual void Func(){}
};
```

```
class A
{
public:
    virtual void Func(){}
};

class B : public A
{
public:
    void Func(){}
};
```

还有中情况，本来我想新写一个函数，但是不小心名字写成和父类虚函数一样的函数。造成重写父类虚函数。

```
class A
{
public:
    virtual void Func(){}
};

class B : public A
{
public:
    virtual void FUnc(){}
};
```

比如重写虚函数的时候，我手误，写了个大小写不一样的虚函数。系统会单独定义这个FUnc虚函数。

override就是确保子类复写的虚函数名称，必须和父类虚函数名称一模一样，或者子类重写的函数名确保在父类有相同抽象的虚函数名可以继承。

```
class A
{
public:
    virtual void Func(){}
};

class B : public A
{
public:
    virtual void FUnc() override
    {}      ↑
}; 子类继承虚函数名和父类虚函数不一样，报错。
```

```
class A
{
public:
    virtual void Func(){}
};

class B : public A
{
public:
    virtual float FUnc(int x) override
    {} 子类重写父类虚函数，形参，返回值和父类虚函数不一样也会报错。
};
```

所以override就是为了保证子类里面重写的函数，必须和父类虚函数一模一样

final，防止虚函数在子类不停的重写

```
class A
{
public:
    virtual void Func(){}
};

class B : public A
{
public:
    virtual void Func() final
    {}
}; 当我B这个子类重写父类的Func函数之后，C类就算继承我B类，也不能重写Func，因为final截断了。
class C : public B
{
public:
    virtual void FUnc(){}
};
```

for 循环的四种用法

```
int main(void) 这是第1种最常规的for  
{  
    int num[3] = {1,2,3};  
    for (size_t i = 0; i < 3; i++)  
    {  
        cout << num[i] << " ";  
    }  
    return 0;  
} 1 2 3
```

- C++11标准引入了更简单的for语句，这种语句可以遍历

容器或其他序列的所有元素，语法形式如下

```
for ( declaration : expression )  
    statement
```

其中，expression表示的必须是一个**序列**，比如用大括号括起来的**初始值列表**，**数组**，或者是**vector或string类型**的对象，这些类型的共同特点是拥有能返回迭代器的begin和end成员。

其中，declaration定义一个变量，序列中的每个元素都得能转换成该变量的类型。确保类型相容最简单的办法是使用**auto**类型说明

```
int main(void)  
{  
    int num[10] = {1,2,3}; //c++声明的数组长度可以大于初始化元素个数  
    for (auto v : num) ← 1. 其实意思很简单，就是循环前，系统先计算num里面有  
    {  
        cout << v << endl; v。直到数组元素个数取完。  
    }  
  
    return 0;  
}
```

```
1  
2  
3  
0  
0  
0  
0  
0  
0  
0  
0  
0  
0
```

```
int main(void)  
{  
    int num[10] = {1,2,3}; //c++声明的数组长度可以大于初始化元素个数  
    for (auto &v : num) ← 2. 这就是运行结果  
    {  
        v = v * 2; //修改数组里面元素内容  
    }  
    for (auto i : num) //可以直接for，不需要{} 3. 如果要修改数组里面的元  
    {  
        cout << i << endl;  
    }  
    return 0;  
}
```

```
2  
4  
6  
0  
0  
0  
0  
0  
0  
0  
0  
0
```

```
int main(void)  
{  
    vector<int> num = {1,2,3}; 5. 使用vector变量，变量返回的是迭代器，所以用auto自动定义  
    //迭代器变量，这表示迭代器开始位置  
  
    for (auto it = num.begin(); it != num.end(); ++it) 6. vector 数组结束位置  
    {  
        auto &v = *it; //引用v，接收迭代器当前元素的地址  
        v = v *2; //修改v值，就是修改当前元素内容  
    }  
    for(auto i : num) 7. 每次循环，迭代器++  
    {  
        cout << i << " ";  
    }  
    return 0;  
} 2 4 6 8. 修改成功
```

```
0  
0  
0  
0  
0  
0  
0  
0  
0  
0  
0  
0
```

C++11: 字符串原始字面量

字面量主要解决字符串中一些制表符问题

```
int main(void)
{
    char str[] = "hellow\th1234"; →
    printf("%s\r\n",str);

    char str2[] = R"(hellow\th1234)"; ↗
    printf("%s\r\n",str2); ↑
    return 0; C++11为了解决这个问题，使用R字面量，这样字符串里面的特殊符号也会被当作字符输出。
}
```

所以字面量语法就是R"(.....)"

填入你要输出的字符串

```
int main(void) 4. 我如果用字面量，就解决了双反斜杠的问题。
{
    char str[] = R"(D:\doc\world\test.txt)";
    printf("%s\r\n",str); 5. 我单\就相当于linux下的/正斜杠 ↴
    return 0;           D:\doc\world\test.txt
}
```

```
int main(void) 1. 在windows中，我要找到我自己的文件，必须使用路径'\'反斜杠来找文件
{
    char str[] = "D:\doc\world\test.txt";
    printf("%s\r\n",str); 2. 但是在字符串中，单个反斜杠\是错误的。
    char str2[] = "D:\\doc\\world\\test.txt";
    printf("%s\r\n",str2);
    return 0;
}
```

3. 必须使用双斜杠来表示路径

```
int main(void)
{
    char str[] = "<html>\n<head>\n<title>\n测试浏览器" ←
    </title>\n</head>";
    printf("%s\r\n",str);
    return 0;
}
```

6. 我想后台发生html请求的时候，如果内容太多，需要换行，我必须用'\'反斜杠换行符来实现。好麻烦

```
int main(void)
{
    char str[] = R"(<html>
<head>
<title>
测试浏览器
</title>
</head>)"; 我直接用字面量R，就解决了换行使用\连接符的问题，让字符串按照我的要求排列输出
    printf("%s\r\n",str);
    return 0;
}
```

<html> 可读性提高
<head>
<title>
测试浏览器
</title>
</head>

```
int main(void)
{
    char str[] = R"hello(<html>
<head>
<title>
测试浏览器
</title>
</head>)"; 如果不小心在()前面写了字符，就会编译报错
    printf("%s\r\n",str);
    return 0;
}
```

```
int main(void)
{
    char str[] = R"hello(<html>
<head> 如果在()后面加与前面一样的字符就不会报错。但是字符串打印不会输出()两边的字符
<title> 测试浏览器
</title> hello";
    printf("%s\r\n",str);
    return 0;
}
```

<html>
 <head>
 <title>
 测试浏览器
 </title>
 </head>

C++11: auto 关键字，自动推导变量类型

```
int main(void)
{
    //没有const修饰的变量
    auto x = 3.14; //推到出来是double类型，所以auto会被替换成double
    auto y = 520; //推导出来是整形类型，所以auto会被替换成double
    auto z = 'a'; //推导出来是char类型，所以auto会被替换成char
    auto nb; //这种auto定义变量之后，没被初始化，是错误的，编译错误
    auto double nb1; //因为有了auto的自动推导，如果这时候还加入确定的变量类型，编译报错。
    return 0;
}

int main(void)
{
    //有const修饰
    int tmp = 250;
    const auto a1 = tmp; //根据传入的tmp类型，我们推导出const int a1
    auto a2 = a1; //a2因为a1是int，所以a2被推导出是int类型
    const auto& a3 = tmp; //在推导过程中，如果auto变量前面是const类型，//就要保留const。const int a3
    if (auto* pt4 = &a1) //如果 auto* pt4 = &a1。如果auto是指针，那么推导就要保留const和volatile。如果auto不是指针
        return 0; //const和volatile会自动取消。
}
```

不允许使用auto 的四个场景

```
int func(auto a, auto b)
{
    ↑
    auto不能做形参，因为
    形参没有被初始化，
    auto无法正确使用。
```

```
auto array[] = {1,2,3,4,5,6};
```

不能使用auto定义数组

```
class test
{
    ↑
    auto不能用于类里面非
    静态成员变量初始化
    auto v1 = 0;
    static auto v2 = 0;
    static const auto v3 = 10;
}
```

因为类的非静态成员，不属于类，是属于对象的。所以只有对象创建出来之后才能对类非静态成员初始化。所以auto无法使用。

```
template <typename T>
class test
{
private:
    /* data */
public:
    test(){}
    ~test(){}
};

int main(void)
{
    test<int> t; //模板直接写入确切的类型没有问题<int>
    test<auto> t1; //如果直接给模板定义auto就不得行
    return 0;
}
```

C++11 auto 适用的场景

auto 适合用在 STL 库里面容器的遍历，vector, list, map, 队列等

map 使用，STL 库 map

```
int main(void)
{
    map<int, string> mp; //map容器用于key, value使用
    mp.insert(make_pair(1,"ace")); //使用insert向map容器插入数据
    mp.insert(make_pair(2,"sab")); //因为是用的键值对，所以用make_pair创建pair对象
    mp.insert(make_pair(3,"luffy")); // 3是value值, luffy是键值
    map<int, string>::iterator it = mp.begin(); //对一个容器进行遍历，需要有迭代器
                                                //这里用iterator定义个迭代器it，对it初始化mp.begin
    for(; it != mp.end(); ++it) //如果it !=迭代器尾部，就继续++it
    {
        cout << "key: " << it->first << ", value :" << it->second << endl;
    }

    return 0;
}
```

key: 1 , value :ace
key: 2 , value :sab
key: 3 , value :luffy

```
int main(void)
{
    map<int, string> mp; //map容器用于key, value使用
    mp.insert(make_pair(1,"ace")); //使用insert向map容器插入数据
    mp.insert(make_pair(2,"sab")); //因为是用的键值对，所以用make_pair创建pair对象
    mp.insert(make_pair(3,"luffy")); // 3是value值, luffy是键值
    map<int, string>::iterator it = mp.begin(); //对一个容器进行遍历，需要有迭代器
                                                //这里用iterator定义个迭代器it，对it初始化mp.begin
    for(; it != mp.end(); ++it) //如果it !=迭代器尾部，就继续++it
    {
        cout << "key: " << it->first << ", value :" << it->second << endl;
    }
    return 0;
}
```

```
int main(void)
{
    map<int, string> mp; //map容器用于key, value使用
    mp.insert(make_pair(1,"ace")); //使用insert向map容器插入数据
    mp.insert(make_pair(2,"sab")); //因为是用的键值对，所以用make_pair创建pair对象
    mp.insert(make_pair(3,"luffy")); // 3是value值, luffy是键值
//    map<int, string>::iterator it = mp.begin();
修改后auto it = mp.begin(); //根据我调用的mp类型，自动推到出it变量就是迭代器
    for(; it != mp.end(); ++it) //如果it !=迭代器尾部，就继续++it
    {
        cout << "key: " << it->first << ", value :" << it->second << endl;
    }
    return 0;
}
template <typename T>
void func(void)
{
    auto ret = T::get(); //其实我觉得使用Auto自动推到和直接用T返回差不多
    cout << "ret:" << ret << endl;
}
```

key: 1 , value :ace
key: 2 , value :sab
key: 3 , value :luffy

这个程序在定义迭代器的时候，需要指定迭代器类型，敲这么多代码感觉很麻烦。

C++11: decltype 关键字，自动推导变量类型，和 auto 有点类型

使用方法: decltype(表达式)

```
int main(void)
{
    int a = 10;      // decltype就是经过传入的参数类型，自
                     // 动推导出后面定义变量的类型。
    decltype(a) b = 99; // b->int
    decltype(a + 3.14) c = 52.13; // c ->double
    return 0;
}
```

```
class Test
{
public:
    int num = 9;
    string text;
    static const int value = 110;
};
```

```
int main(void)
{
    int x = 99;
    const int &y = x;
    decltype(x) a = x; // x赋值给a, 因为x是int, 所以a也是int
    decltype(y) b = x; // 因为y是const int类型的引用, 所以, b也是引用
    decltype(Test::value) c = 0; // c是const int类型
    Test t;
    decltype(t.text) d = "xxxxxx"; // d是string类型
    return 0;
}
```

```
int func_int(){}           //返回值
int& func_int_r(){}       //返回左值引用
int&& func_int_rr(){}     //返回右值引用
const int func_cint(){}    //返回const int
const int& func_cint_r(){} //返回const int左值引用
const Test func_ctest(){}  //返回对象

int main(void)
{
    int n = 100;
    decltype(func_int()) a = 0;    //返回的是值, 所以推导a是int
    decltype(func_int_r()) b = n;  //返回的是左值引用, 所以推导必须将其它变量引用给b
    decltype(func_int_rr()) c = 0; //返回的是值, 所以推导c = 0, 右值引用
    decltype(func_cint()) d = 0;   //返回的是const右值, 所以d = 0;
    decltype(func_ctest()) g = Test; //返回的是对象;

    return 0;      //从以上能看出, decltype返回值是什么类型, 推导出的就是什么类型
}
```

decltype 最适合应用的场景就算模板类, 下面举个例

```
template <typename T>
class Container
{
private:
    T::iterator m_it; // 定义T类型迭代器
    //这个地方会编译报错
public:
    Container(){}
    ~Container(){}
    void print(T &t)
    {
        for(m_it = t.begin(); m_it != t.end(); ++m_it)
        {
            cout << "value: " << *m_it << endl;
        }
    }
};
int main(void)
{
    list<int> ls{1,2,3,4,5,6,7}; // c++可以用{}初始化模板序列
    Container<list<int>> c;
    c.print(ls); // 我们ls调用的begin返回值类型和it是一样的。
    return 0; // 我们就是想用ls.begin执行之后推导出迭代器类型
}
```

```

template <typename T> ← 1. 创建对象的时候T就传入了list<int>
class Container
{
private:
    decltype(T().begin()) m_it; //将T类型迭代器修改从自动推导迭代器
public:
    Container(){}           2. T()变成list<int>().begin()
    ~Container(){}          3. 经过decltype返回list<int>()类型给m_it
    void print(T &t)
    {
        for(m_it = t.begin(); m_it != t.end(); ++m_it)
        {
            cout << "value: " << *m_it << endl;
        }
    }
};

int main(void)
{
    list<int> ls{1,2,3,4,5,6,7}; //c++可以用{}初始化模板序列
    Container<list<int>> c;
    c.print(ls);
    return 0;
}

```

C++11: Lambda 表达式

C++11 的 lambda 表达式最大的特点就是匿名函数，没有函数名，用起来方便灵活。

lambda 表达式语法: [capture] (params) opt->ret{....}

capture: 表示捕捉方式，捕捉方式有很多种

params: 参数列表，这个匿名函数如果有形参，就把形参写入

opt: 是选项，可以忽略不写。

ret: 匿名函数返回值，这个返回值是后置，也可以不写

....: 函数体

lambda 表达式的捕获列表可以捕获一定范围内的变量，具体使用方式如下：

- [] - 不捕捉任何变量
- [&] - 捕获外部作用域中所有变量，并作为引用在函数体内使用（按引用捕获）
- [=] - 捕获外部作用域中所有变量，并作为副本在函数体内使用（按值捕获）
 - 拷贝的副本在匿名函数内部是只读的
- [=, &foo] - 按值捕获外部作用域中所有变量，并按照引用捕获外部变量 foo
- [bar] - 按值捕获 bar 变量，同时不捕获其他变量
- [&bar] - 按值捕获 bar 变量，同时不捕获其他变量
- [this] - 捕获当前类中的 this 指针
 - 让 lambda 表达式拥有和当前类成员函数同样的访问权限
 - 如果已经使用了 & 或者 =，默认添加此选项

```

void func(int x, int y)
{
    int a;
    int b;
    [](){
        int c = a;
        int d = x;
    };
}

```

这就是匿名函数
lambda 表达式
因为我[]什么都没写，这时候匿名函数
部使用外部变量赋值就会报错。

```

void func(int x, int y)
{
    int a;
    int b;
    [=](){
        int c = a;
        int d = x;
    };
}

```

使用=号，就可以
将外部变量使用进
匿名函数，是相互
赋值的方式

注意：[&]是引用方式，就算直接操作外部变量地址，修改地址上的值。

注意: [=] 赋值方式, 就算将外部变量的值拷贝进 lambda 里面使用, 类似 `x=a` 这种赋值运算。

The diagram shows three snippets of C++ code illustrating lambda capture by value:

- Snippet 1:** A function `func` that captures variable `a` by value using the expression `[=, &x]()`. Inside the lambda, `c = a;` and `d = x;` are shown.
- Snippet 2:** A class `Test` with a member function `output` that captures `this` by value using the expression `[this]`. Inside the lambda, `m_number` is returned.
- Snippet 3:** A class `Test` with a member function `output` that captures `this, x, y` by value using the expression `[this, x, y]`. Inside the lambda, `m_number` is returned.

A red arrow points from the text "lambda表示指定了this, 就意味着我们在匿名函数内部只能使用this这个对象对应的类里面的成员变量/成员函数。" to the `[this]` in Snippet 2.

The diagram shows three snippets of C++ code illustrating lambda capture by value and mutation:

- Snippet 1:** A function `func` that captures variables `a` and `b` by value using the expression `[=, &x]()`. Inside the lambda, `c = a;`, `d = x;`, `x++;`, and `b++;` are shown. A note explains why `x` can be modified but `b` cannot.
- Snippet 2:** A function `func` that captures variables `a` and `b` by value using the expression `[=, &x]()`. Inside the lambda, `c = a;`, `d = x;`, `x++;`, and `b++;` are shown. A note explains that because `x` is captured by value, it is modified directly, while `b` is captured by reference.
- Snippet 3:** A function `func` that captures variables `a` and `b` by value using the expression `[=, &x](){mutable}`. Inside the lambda, `c = a;`, `d = x;`, `x++;`, and `b++;` are shown. A note explains that if modification is required, `mutable` must be added.

The diagram shows two snippets of C++ code illustrating the execution of anonymous functions:

- Snippet 1:** A function `func` that captures variables `a` and `b` by value using the expression `[=, &x](){mutable}`. Inside the lambda, `c = a;`, `d = x;`, `x++;`, and `b++;` are shown. A note explains that because the anonymous function is only defined and not called, there is no output.
- Snippet 2:** A function `func` that captures variables `a` and `b` by value using the expression `[=, &x](){}`. Inside the lambda, `c = a;`, `d = x;`, `x++;`, and `b++;` are shown. A note explains that if the anonymous function has no parameters, it can be executed directly after the closing brace.

A red arrow points from the question "1. 为什么cout只执行了一次?" to the output "b: 0".

```

void func(int x, int y)
{
    int a;      4. 如果匿名函数有形参
    int b;
    [= , &x](int z)mutable{
        int c = a;
        int d = x;
        x++;
        b++;
        cout << "b: " << b << endl;
        cout << "z: " << z << endl;
    }(88);← 5. 就必须传入形参
    cout << "b: " << b << endl;
}

int main(void) 在C++里面，lambda表达式会被看成仿函数
{
    func(1,2);
    return 0;
}

```

```

using ptr = int(*)(int); //定义指针变量

ptr f = [] (int a)
{
    return a;
};

int main(void)
{
    f(11);
    return 0;
}

```

如果匿名函数[]不捕获任何外部变量，那么整个lambda就被当作函数指针看待。

C++, try catch 和 throw 使用



2. 我无法知道这个服务器到底存不存在，有没有这个服务器？服务器是否启动？
3. 这时候我们可以加入try{ ... }尝试一下链接服务器，如果链接不成功，就执行catch() {}

```

try
{
    /* 执行connect函数 */
    ret = connect();
}
catch()
{
}

```

4. 如果我有多个服务器，这次链接不成功。我可以进入catch，然后返回try再次链接。

```

try
{
    /* 执行登录数据库函数 */
    ret = SQL();
}
catch()
{
}

```

如果登录密码错误，返回try

```

double divide(int a , int b)
{
    if(b == 0) 1.如果我b输入的是0, 可以使用throw抛出异常
    {
        throw "by zero"; //如果b == 0, throw主动抛出异常
    }
    return static_cast<double>(a) / b;
}

```

```

int main(void)
{
    try
    {
        6. 如果我b输入的是1
        int numerator, denominator;
        double result = divide(1,1); ↴
        cout << " result " << result << endl;
    }
    catch(const char* error)
    {
        cout << error << endl;
    }
    7. 异常不执行, 直接try代码段执
    行完成
    return 0;           result 1
}

```

```

int main(void)
{
    try 2. 我在try里面调用divide函数
    {
        int numerator, denominator;
        double result = divide(1,0); 3. 我输入0
        cout << " result " << result << endl;
    }
    catch(const char* error) ↴ 4. 这时候divide函数
    {                                内部发生异常, 直接
        cout << error << endl;      中断try后面的函数执
    }                                行, 进入catch
    return 0;                         5. catch随便定义个变
                                        量, 接收throw扔出来
                                         的异常信息
}

```

注意: try catch 异常处理方式只能在真正需要的时候使用, 或者有必要地方使用, 不能到处滥用, 因为异常处理机制会导致程序性能方面的下降。

标准异常类型

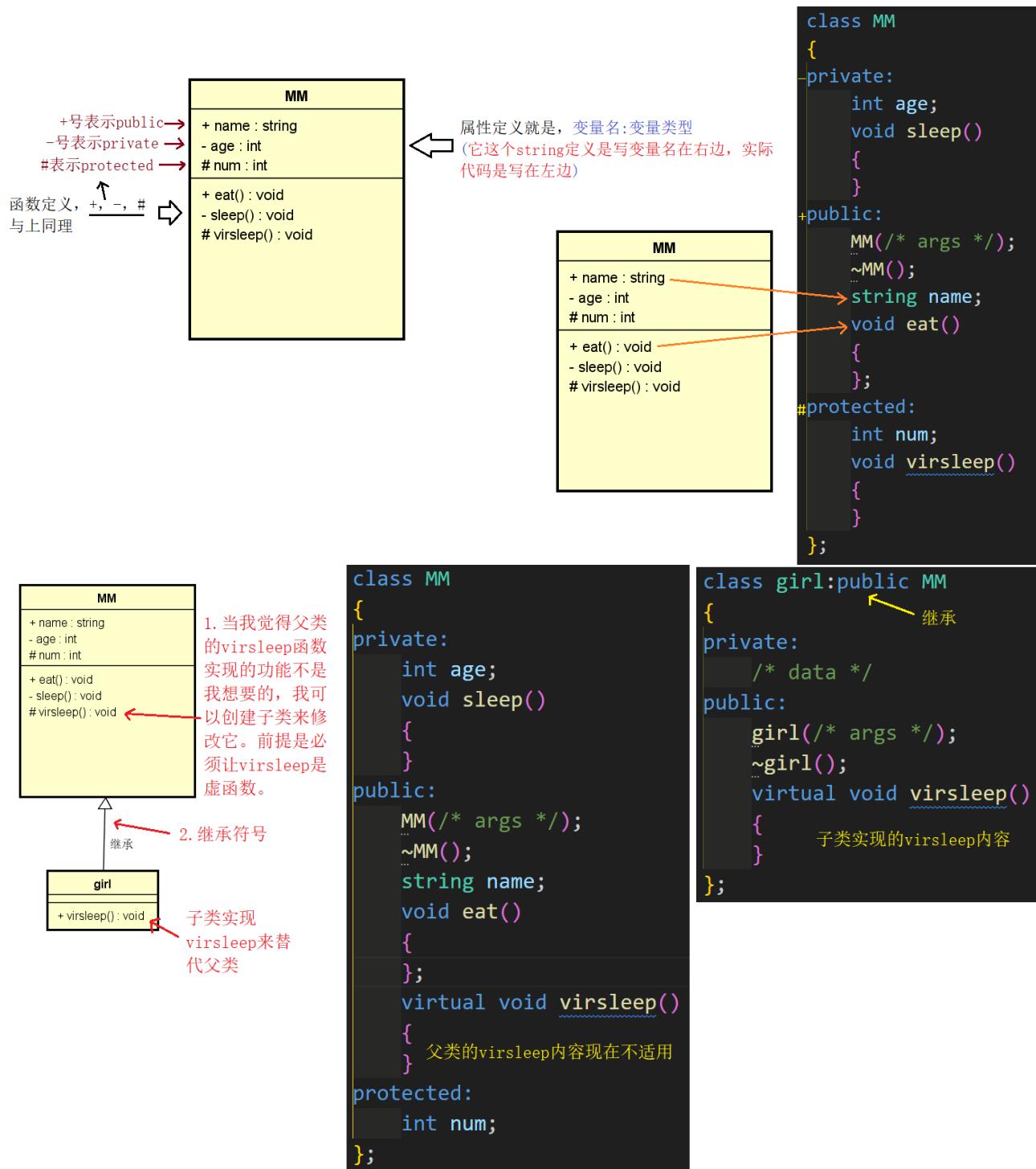
当使用 C++ 的标准异常类时, 通常需要包含 `std::` 命名空间前缀。但为了简洁起见, 下面的表格将省略 `std::` 前缀, 仅列出异常类的名称。

| 异常类 | 描述 |
|---------------------------|---|
| <code>exception</code> | 所有标准异常类的基类, 包含描述字符串的 <code>what()</code> 成员函数。 |
| <code>bad_alloc</code> | 动态内存分配失败时抛出, 例如 <code>new</code> 操作失败。
这个异常类型用得比较多 |
| <code>bad_cast</code> | <code>dynamic_cast</code> 操作失败时抛出, 用于多态类型转换。 |
| <code>bad_typeid</code> | <code>typeid</code> 操作失败时抛出, 用于获取类型信息。 |
| <code>logic_error</code> | 逻辑错误的基类。 |
| <code>domain_error</code> | 表示数学上的域错误, 例如参数超出有效范围。 |

runtime_error 运行时错误的基类。

C++设计模式与语法实战

UML 类图介绍



```

class MM
{
private:
    int age;
    void sleep()
    {
    }
public:
    MM(/* args */)//构造函数必须空实现,不然运行报错
    {
    }
    ~MM();
    string name;
    void eat()
    {
    };
    virtual void virsleep()
    {
        cout<<"父类现在不适用"<<endl;
    }
protected:
    int num;
};

这种继承多态的方式可以节省修改父类使用的代码。

```

这种都用父类定义对象，让对象去接收不同的类，有什么好处？
 ↓
 父类实现了很多函数
 main使用了父类很多函数功能。
 但是有几个功能要子类重写

```

class girl:public MM
{
private:
    /* data */
public:
    girl/* args */ //构造函数必须空实现,不然运行报错
    {
    }
    ~girl();
    virtual void virsleep()
    {
        cout<<"子类适用"<<endl;
    }
};

int main(void)
{
    MM *Mobj = new MM();
    Mobj->virsleep();

    MM *Mob2 = new girl();
    Mob2->virsleep();
    return 0;
}

```

注意，对象定义的时候都是用的父类。
 根据创建对象时候，new的哪一个类，来确定用哪一个类里面virsleep函数。
 父类现在不适用子类适用

```

class MM
{
private:
    int age;
    void sleep()
    {
    }
public:
    MM/* args *///构造函数必须空实现,不然运行报错
    {
    }
    ~MM();
    string name;
    void eat()
    {
        cout<<"吃东西"<<endl;
    };
    virtual void virsleep()
    {
        cout<<"父类现在不适用"<<endl;
    }
};

```

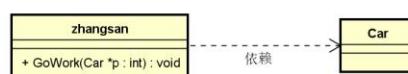
```

class girl:public MM
{
private:
    /* data */
public:
    girl/* args */ //构造函数必须空实现,不然运行报错
    {
    }
    ~girl();
    virtual void virsleep()
    {
        cout<<"子类适用"<<endl;
    }
};

int main(void)
{
   为了不动整个调用流程，只修改了类名
    MM *Mob2 = new girl();
    Mob2->virsleep();
    Mob2->eat(); //使用父类的函数
    return 0;
}

```

UML 类图继承关系和依赖关系



张三要去上班，但是没有交通工具，需要去借用一辆车，那么这个车和张三就是依赖关系，张三依赖车

```

//张三人物
class zhangsan
{
public:
    void GoWork(Car *p) //去上班动作
    {
    }
    Car* fix() //有返回值，返回的是另外一个类的对象，也是依赖关系
    {
       这也是一种依赖关系
    }
protected:
private:
};

一句话概括就是：一个类是另外一个类的函数参数或者函数返回值

```



//张三人物

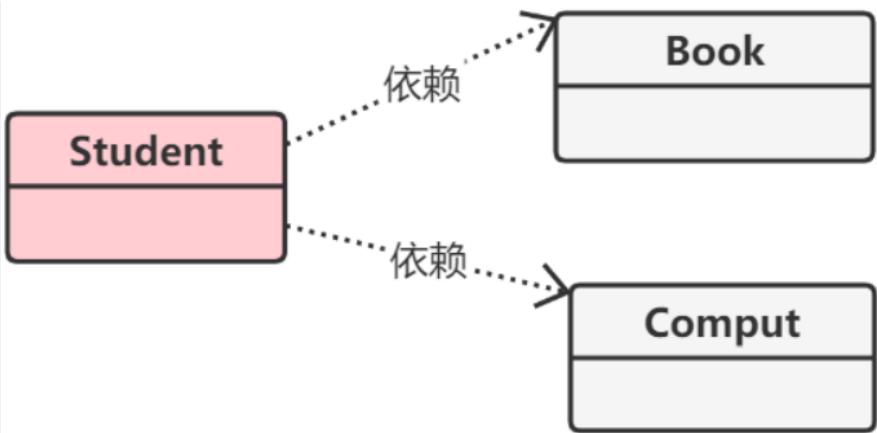
```

class zhangsan
{
public:
    void GoWork(Car *p) //去上班动作
    {
    }
protected:
private:
};

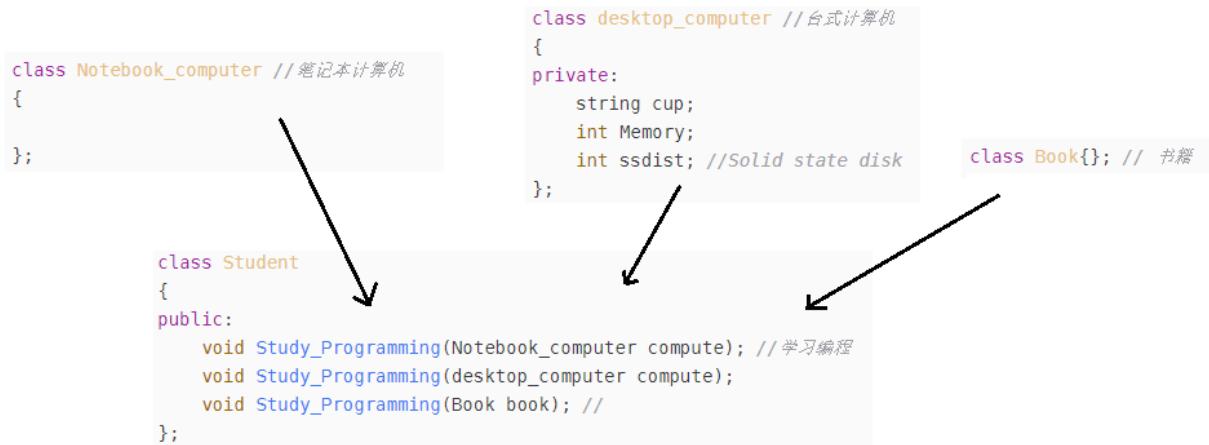
//汽车类
class Car
{
private:
    /* data */
public:
    Car/* args */(){}
    ~Car(){}
};

```

一个类里面的函数，形参是另外一个类的指针，或者是参数，就表示依赖

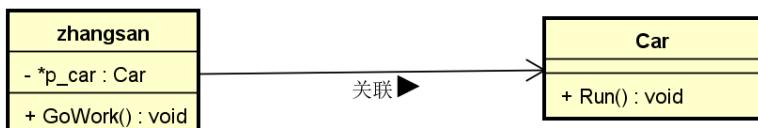


如果一个对象类内部有构造new另外一个类的对象实例，这两个对象就算关联关系。



一个student类里面创建了其它三个类的对象，就算依赖关系。

UML 类图关联关系



```

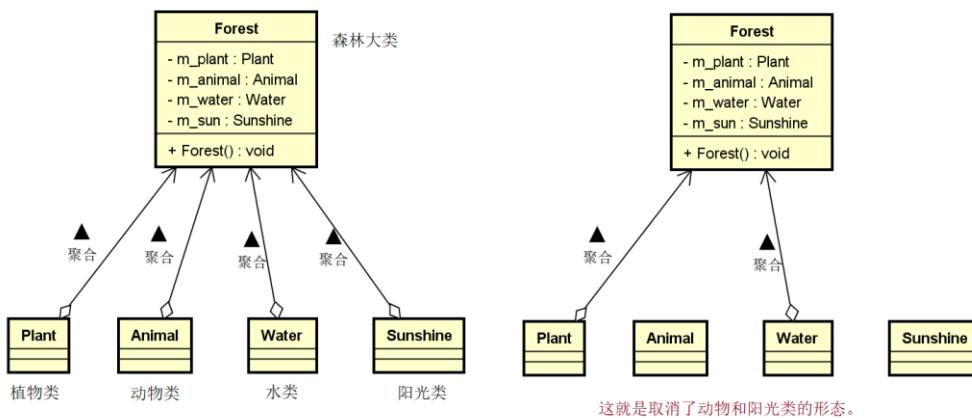
// 张三与车关联(实现)关系
class zhangsan
{
    张三类里面运行汽车类里面的函数
    public: void GoWork()
    {
        p_car->Run();
    }
    将其余的汽车类定位到张三类私有属性
    protected: Car *p_car;
};
  
```

```

// 汽车类
class Car
{
private:
    /* data */
public:
    Car(/* args */){}
    ~Car(){}
    void Run()
    {
    }
};
  
```

也就是一个类是另外一个类的成员变量

UML 类图聚合和组合关系



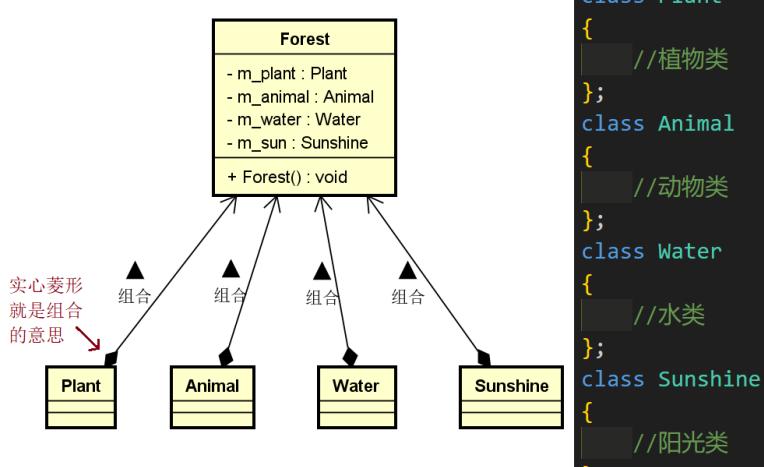
1. 聚合的意思是一个大类将其小类组合起来。但是呢也可以不完全依赖小类。

比如说，一个森林，可以有植物，有动物，有水，有阳光。这时候我给森林取消了阳光，森林还是存在的。我给森林取消了动物，森林也是存在的。所以我可以根据我的需求裁剪不要的类。



组合关系

但是在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也不存在，整体对象和成员对象之间具有同生共死的关系。



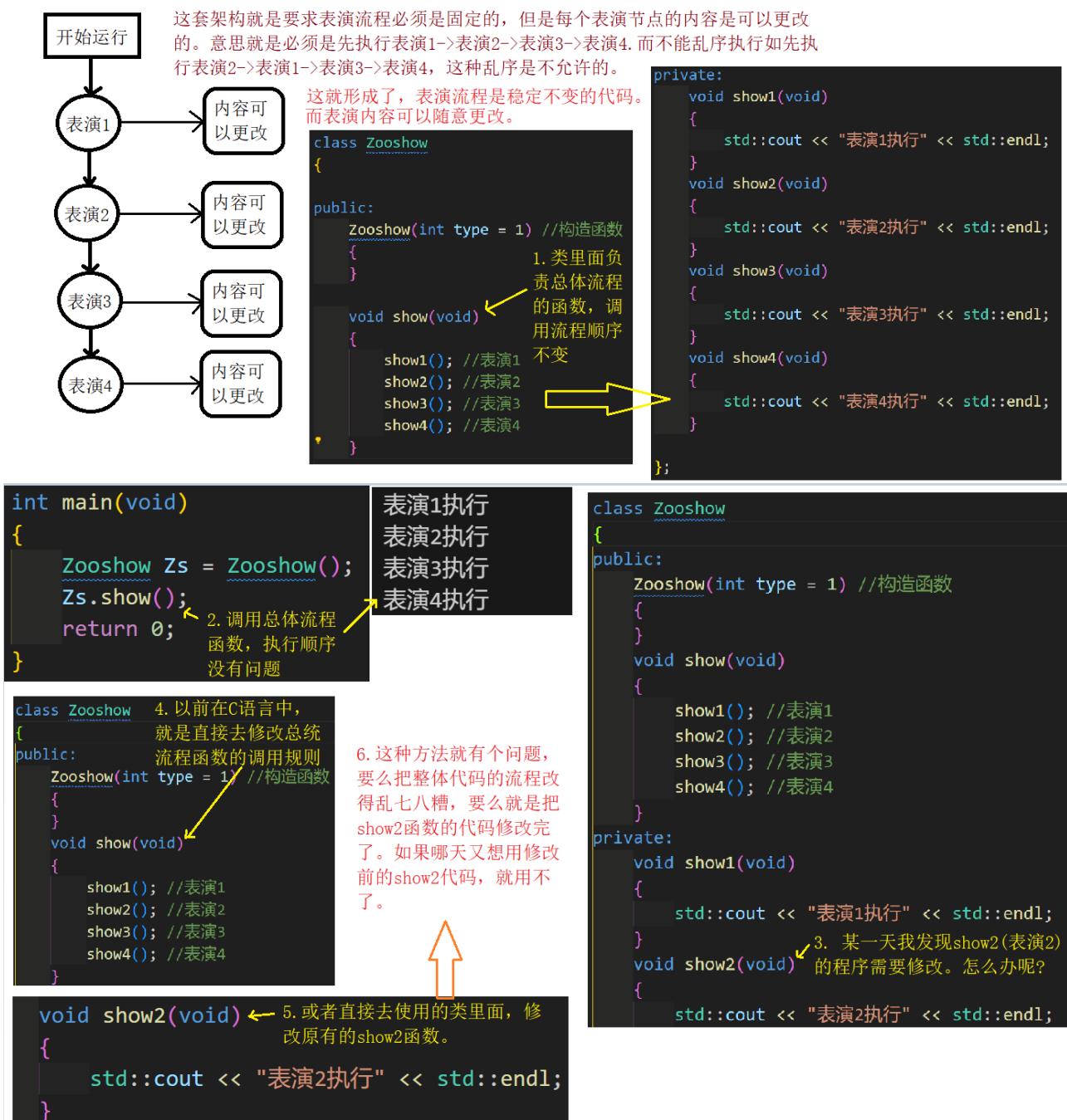
```
//森林大类
class Forest{
public:
    Forest()
    {
        m_plant = new Plant();
        m_animal = new Animal();
        m_water = new Water();
        m_sun = new Sunshine();
    }
    ~Forest()
    {
        delete m_plant;
        delete m_animal;
        delete m_water;
        delete m_sun;
    }
private:
    Plant *m_plant;
    Animal *m_animal;
    Water *m_water;
    Sunshine *m_sun;
}
```

Annotations on the right side of the code:

- 组合就是森林大类这种对象建立之后，一定要将组合进来的小类创建起来。因为大类和组合进来的小类都是不可或缺的。
- 大类内存释放的时候也要将小类全部释放，这就是组合的意义。

模板方法，设计模式的基础

案例：某个品牌动物园，有一套固定表演流程，但是其中有若干个表演子流程可以创新替换，请问该需求代码架构如何编写？



```

class Zooshow
{
public:
    Zooshow(int type = 1) //构造函数
    {
    }
    void show(void)
    {
        show1(); //表演1
        show2(); //表演2
        show3(); //表演3
        show4(); //表演4
    }
}

```

1. 现在有一个人，觉得show1()实现的内容不是他想要的，他想去修改show1的内容或者扩展增加show1内容。这时候就必须让以前类里面的show1变成虚函数。

```

protected:
    virtual void show1(void)
    {
        std::cout << "表演1执行" << std::endl;
    }
    virtual void show2(void)
    {
        std::cout << "表演2执行" << std::endl;
    }
    virtual void show3(void)
    {
        std::cout << "表演3执行" << std::endl;
    }
}

```

2. 这样使用虚函数扩展的方式，就可以让类里面的执行流程保持不变，只提供可以修改的部分代码给其它人修改。

```

class ZooshowExtend1: public Zooshow
{
protected:
    void show1(void)
    {
        std::cout << "表演1执行内容被子类修改" << std::endl;
    }
};

int main(void)
{
    ZooshowExtend1 Zs; //也可使用动态创建方法 Zooshow *Zs = new ZooshowExtend1;
    Zs.show(); //Zs->show();
    return 0;
}

```

3. 其它人扩展的时候，自己定义一个子类，一定要继承之前的类。

4. 这时候就可以修改之前类里面show1不足的地方。创建同样的函数名，进行修改。

5. 既然增加了子类来修改不足的地方，那么创建对象的时候，一定是创建子类对象。

6. 调用的时候还是保持不变，调用主类里面的函数。主类里面的整体流程函数就会去执行子类修改的show1

表演1执行内容被子类修改
表演2执行
表演3执行
表演4执行

7. 你看，show1被修改成功。

```
class ZooshowExtend2: public Zooshow
{
    // 修改父类的函数，必须
    // 是protected权限
protected:
    void show1(void)
    {
        std::cout << "表演1执行内容被子类第222次修改" << std::endl;
    }
};

int main(void)
{
    ZooshowExtend2 Zs; //也可使用动态创建方法 Zooshow *Zs = new ZooshowExtend2;
    Zs.show(); //Zs->show();
    return 0;
}
```

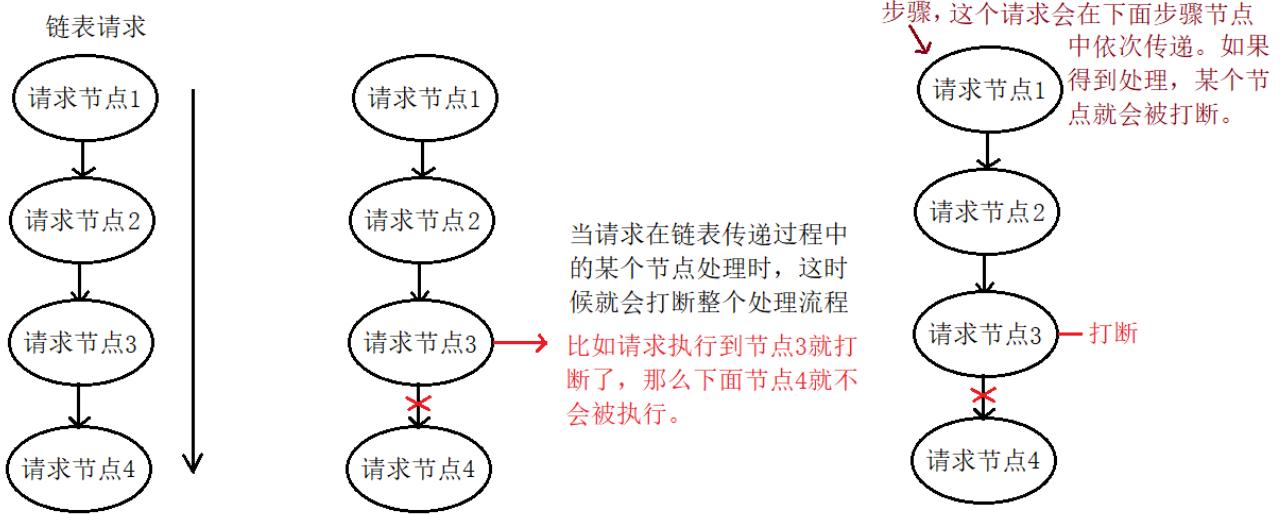
表演1执行内容被子类第222次修改
表演2执行
表演3执行
表演4执行

8. 如果还有个人觉得之前负责扩展
ZooshowExtend1类的人，写的show1代码不
过关，那么可以自己再定义一个子类，去
继承主类，修改show1的内容

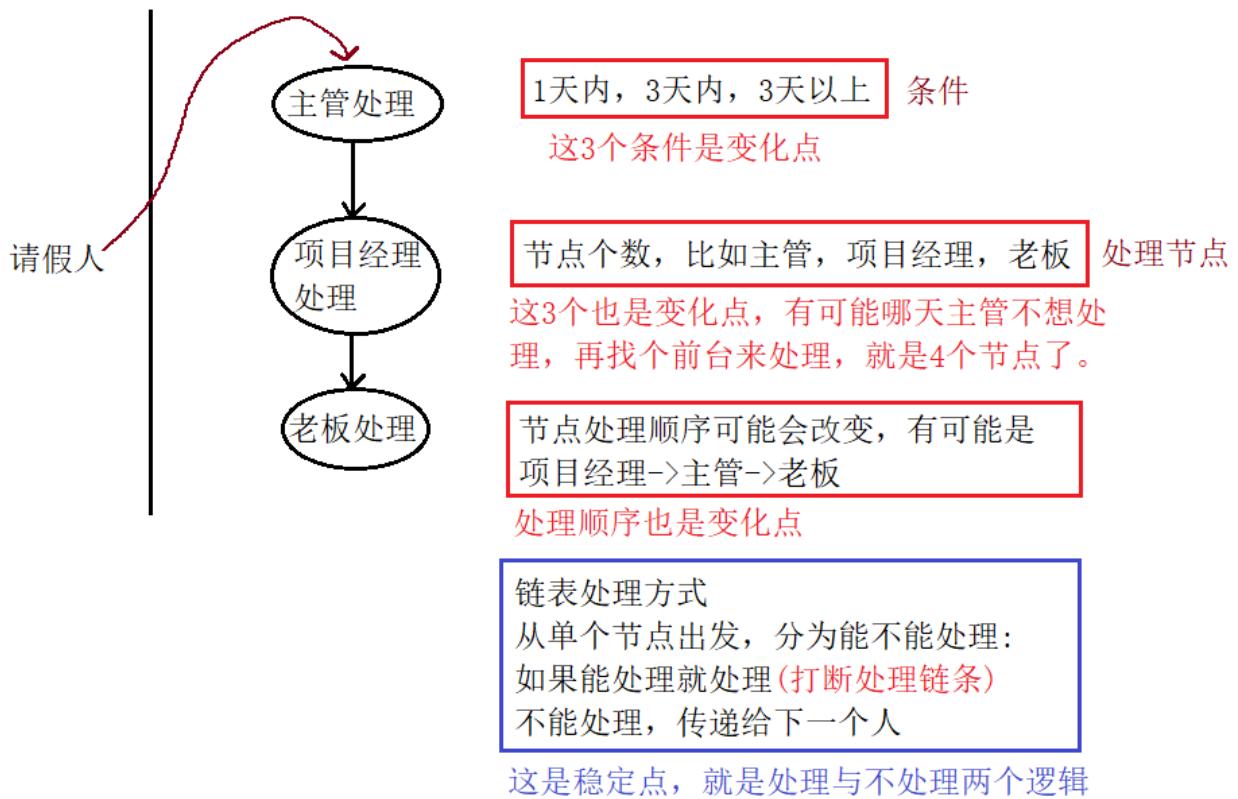
9. 这时候如果是新修改的show1使用，那么创建对象的时候，也必须使用新修
改的子类，show1才有效。

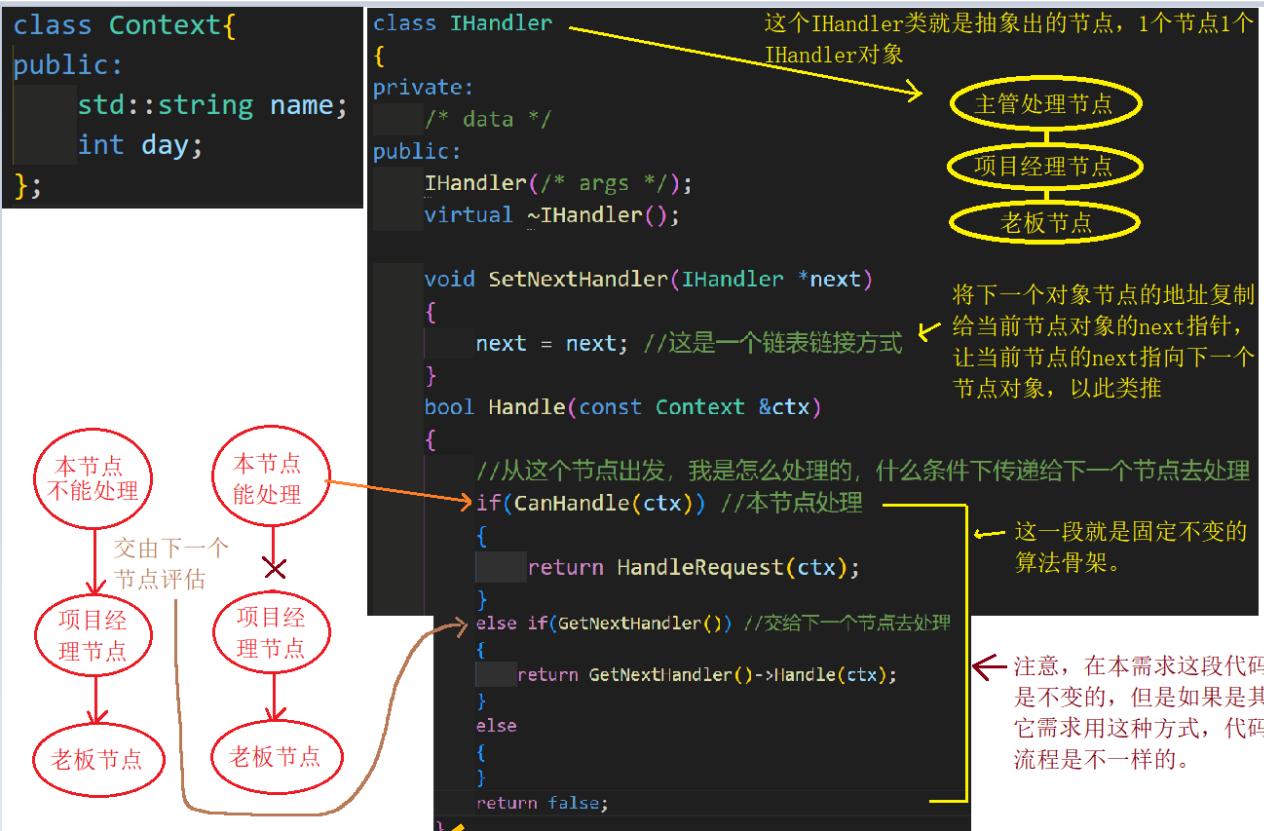
责任链模式

责任链模式就是多个对象都有机会处理请求，从而避免请求发送者与接收者之间的耦合关系。责任链模式也是属于接口隔离的一种。



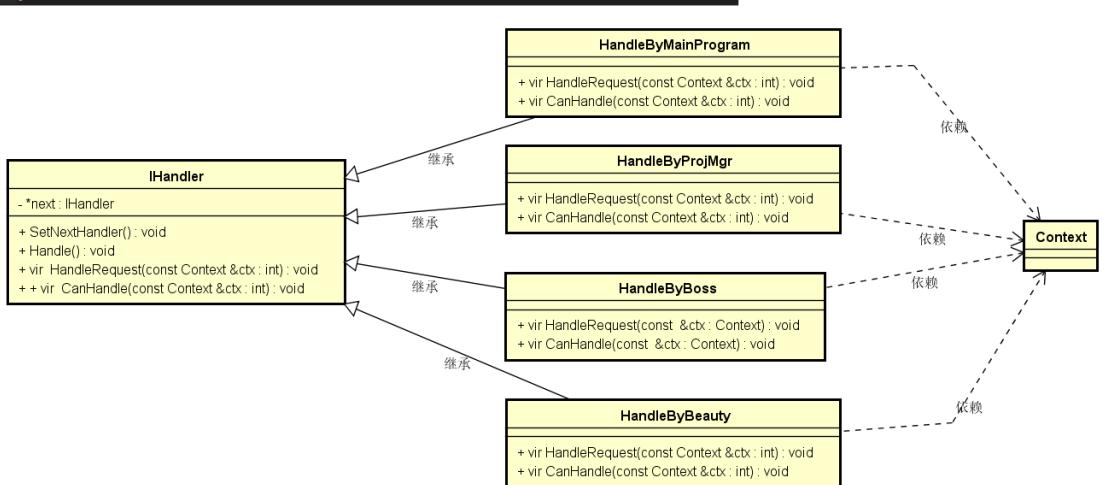
案例：有一个请假流程，1天内需要主管批准，3天内需要项目
经理批准，3天以上需要老板批准。





```
bool Handle(const Context &ctx)
{
    else if(GetNextHandler()) //交给下一个节点去处理
    {
        return GetNextHandler()->Handle(ctx);
    }
    else
    {
    }
    return false;
}
static bool handler_leavereq(Context &ctx)
{
    IHandler *h0 = new HandleByBeauty(); //前台对象
    IHandler *h1 = new HandleByMainProgram();
    IHandler *h2 = new HandleByProjMgr(); //这些就是增加的节点
    IHandler *h3 = new HandleByBoss();
    h0->SetNextHandler(h1); //这就是对象加入链表
    h1->SetNextHandler(h2); //本对象的next指针得到下一个对象的地址
    h2->SetNextHandler(h3);
    return h0->Handle(ctx); //因为是最先从前台开始处理,所以先返回前台地址
}
```

```
protected:
    virtual bool HandleRequest(const Context &ctx)
    {
        return true;
    }
    virtual bool CanHandle(const Context &ctx)
    {
        return true;
    }
    IHandler *GetNextHandler()
    {
        return next;
    }
private:
    IHandler *next; //组合基类指针
```



```

//能不能处理, 以及怎么处理, 这是主管类
class HandleByMainProgram : public IHandler{
protected: 基类增加节点的方式, 就是让其它子类继承自己
    virtual bool HandleRequest(const Context &ctx)
    {
        return true;
    }
    virtual bool CanHandle(const Context &ctx)
    {
        if(ctx.day <= 10)
            return true;
        return false;
    }
};

//这是老板处理类
class HandleByBoss : public IHandler{
protected: 继承IHandler类, 增加3个节点
    virtual bool HandleRequest(const Context &ctx)
    {
        return true;
    }
    virtual bool CanHandle(const Context &ctx)
    {
        if(ctx.day <= 30)
            return true;
        return false;
    }
};

```

```

//这是项目经理处理类
class HandleByProjMgr : public IHandler{
protected: 继承IHandler类, 增加2个节点
    virtual bool HandleRequest(const Context &ctx)
    {
        return true;
    }
    virtual bool CanHandle(const Context &ctx)
    {
        if(ctx.day <= 20)
            return true;
        return false;
    }
};

//新加入的前台
class HandleByBeauty : public IHandler{
protected: 继承IHandler类, 增加4个节点
    virtual bool HandleRequest(const Context &ctx)
    {
        return true;
    }
    virtual bool CanHandle(const Context &ctx)
    {
        if(ctx.day <= 3) //金额
            return true;
        return false;
    }
};

```

```

static bool handler_leavereq(Context &ctx)
{
    IHandler *h0 = new HandleByBeauty(); //前台对象
    IHandler *h1 = new HandleByMainProgram();
    IHandler *h2 = new HandleByProjMgr();
    IHandler *h3 = new HandleByBoss();
    h0->SetNextHandler(h1); //这就是对象加入链表
    h1->SetNextHandler(h2);
    h2->SetNextHandler(h3);
    return h0->Handle(ctx); //因为是最先从前台开始处理, 所以先返回前台地址
}

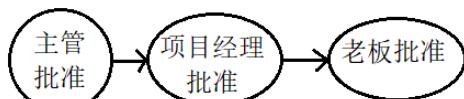
```

这段代码是构建链表处理关系

我怎么看的出来以下需求的稳定点和变化点呢?

请求流程: 1天内主管批准, 3天内项目经理批准, 3天以上老板批准

从我的角度来看, 这3个需求都是变化的。



我可以假设这个批准流程的链表链接结构是稳定点。

对于变化点的扩展, 我们还是采用虚函数覆盖的方式进行扩展。还有使用类指针的方式进行扩展, 如next

```

class IHandler
{
    bool Handle(const Context &ctx)
    {
        if (ctx.type == "leavereq")
            return true;
        else
        {
            return false;
        }
    }

    static bool handler_leavereq(Context &ctx)
    {
        IHandler *h0 = new HandleByBeauty(); //前台对象
        IHandler *h1 = new HandleByMainProgram();
        IHandler *h2 = new HandleByProjMgr();
        IHandler *h3 = new HandleByBoss();

        h0->SetNextHandler(h1); //这就是对象加入链表
        h1->SetNextHandler(h2); //如果我要增加处理节点，比如有了前台，我还要增加一个服务员，那么就在这里添加节点用作服务员 如h4
        h2->SetNextHandler(h3);
        return h0->Handle(ctx); //因为是最先从前台开始处理，所以先返回前台地址
    }
}

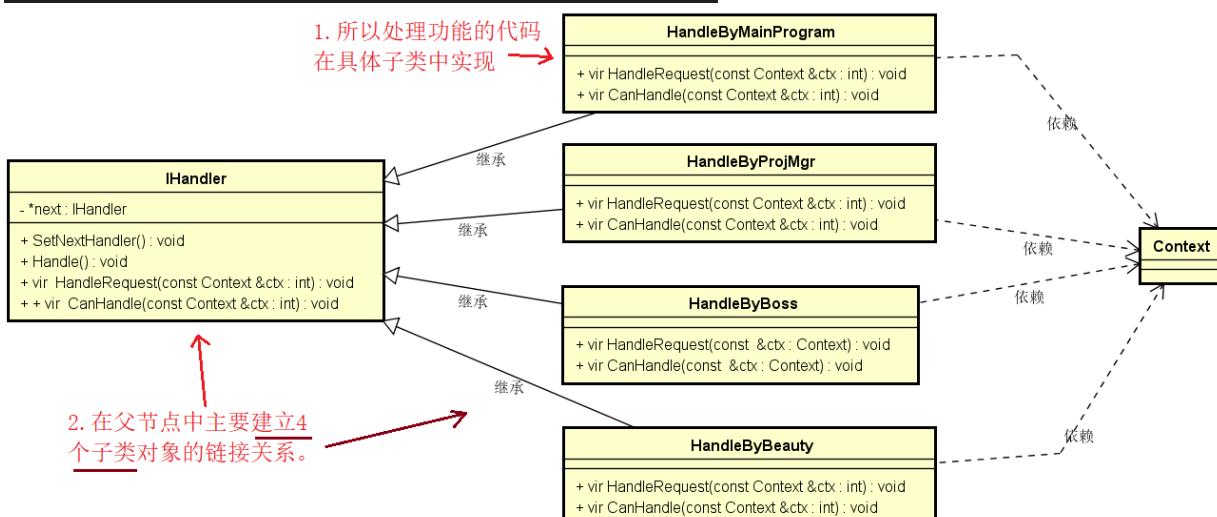
```

```

//能不能处理，以及怎么处理，这是主管类
class HandleByMainProgram : public IHandler{
protected:
    virtual bool HandleRequest(const Context &ctx)
    {
        return true; //处理逻辑就是在不同的实现当中处理，比如继承的子类
    }

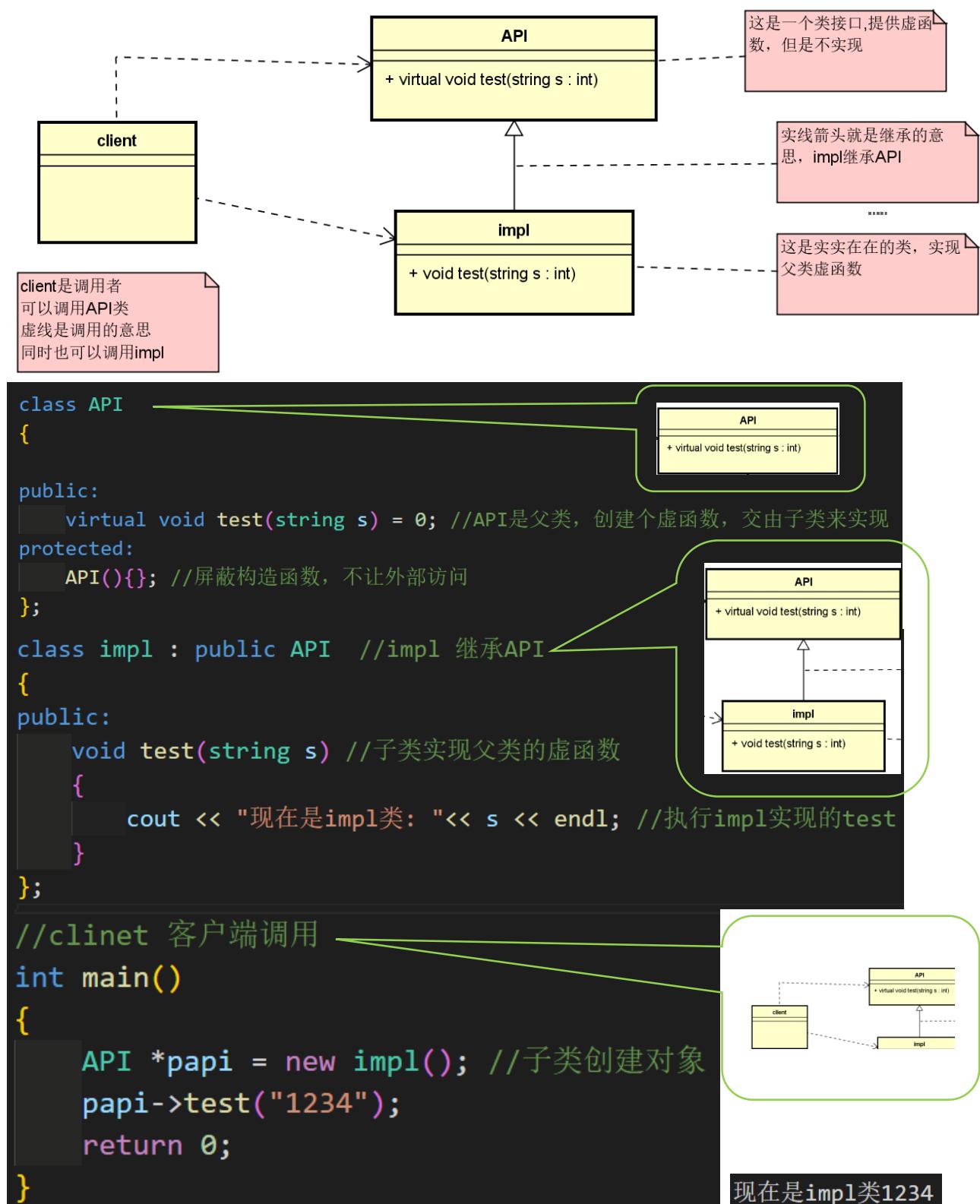
    virtual bool CanHandle(const Context &ctx)
    {
        if(ctx.day <= 10)
            return true;
        return false;
    }
};

```

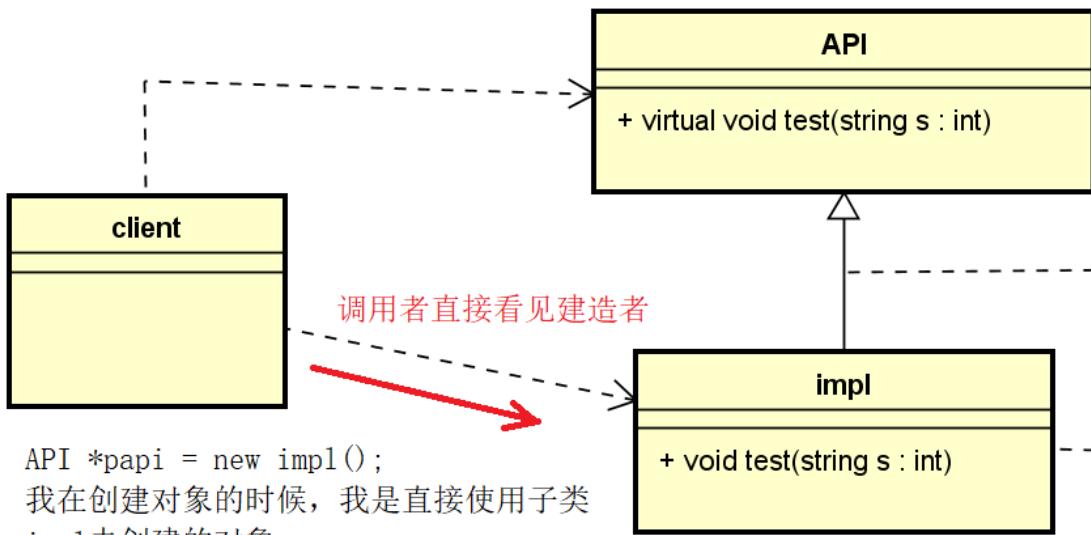


这是责任链的演示案例，没有主函数去实践。

为什么需要工厂方法



这个程序虽然运行正常, 但是最大的问题就是封装性被破坏掉了。



`API *papi = new impl();`
 我在创建对象的时候，我是直接使用子类
`impl`去创建的对象。
 如果我有多个子类，实现的都是`test()`函数
 但是每个`test`函数内部有少许差异，我是不是要记住每个子类的名字，才能使用。
 这就是把创建者直接暴露给用户的问题

所以下面我用简单工厂设计模式来解决这个问题，在解决问题之前，我们讲讲 `nullptr` 的使用方法。

C++11 版本 `nullptr` 空指针使用

引入 `nullptr` 的原因，这个要从 `NUL` 说起。对于 C 程序员来说，一定不会对 `NUL` 感到陌生。但是 C 和 C++ 中的 `NUL` 却不等价

在 C 中，习惯将 `NUL` 定义为 `void*` 指针值 0:

```
#define NUL (void*)0
```

在 C++ 中，`NUL` 却被明确定义为整常数 0:

```
// lmcons.h 中定义 NUL 的源码
#ifndef NUL
#define __cplusplus
#define NUL 0
#else
#define NUL ((void *)0)
#endif
#endif
```

如果遇到 C++ 函数重载的情况

```
// 考虑下面两个重载函数
void foo(int i);
void foo(char* p)

foo(NUL); // which is called?
```

这种就无法编译通过，有些 C++ 编译器能通过，但是会出问题。

`nullptr` 的应用场景

如果我们的编译器是支持 `nullptr` 的话，那么我们应该直接使用 `nullptr` 来替代 `NUL` 的宏定义

```

void func(int* num) //指针形参
{
    cout << "111111" << endl;
}

void func(int num) //非指针形参
{
    cout << "222222" << endl;
}

int main()
{
    func(nullptr);
}

```

nullptr 只能让指针为空，所以调用的是指针形参的函数

```

void func(int* num) //指针形参
{
    cout << "111111" << endl;
}

void func(int num) //非指针形参
{
    cout << "222222" << endl;
}

int main()
{
    func(0);
}

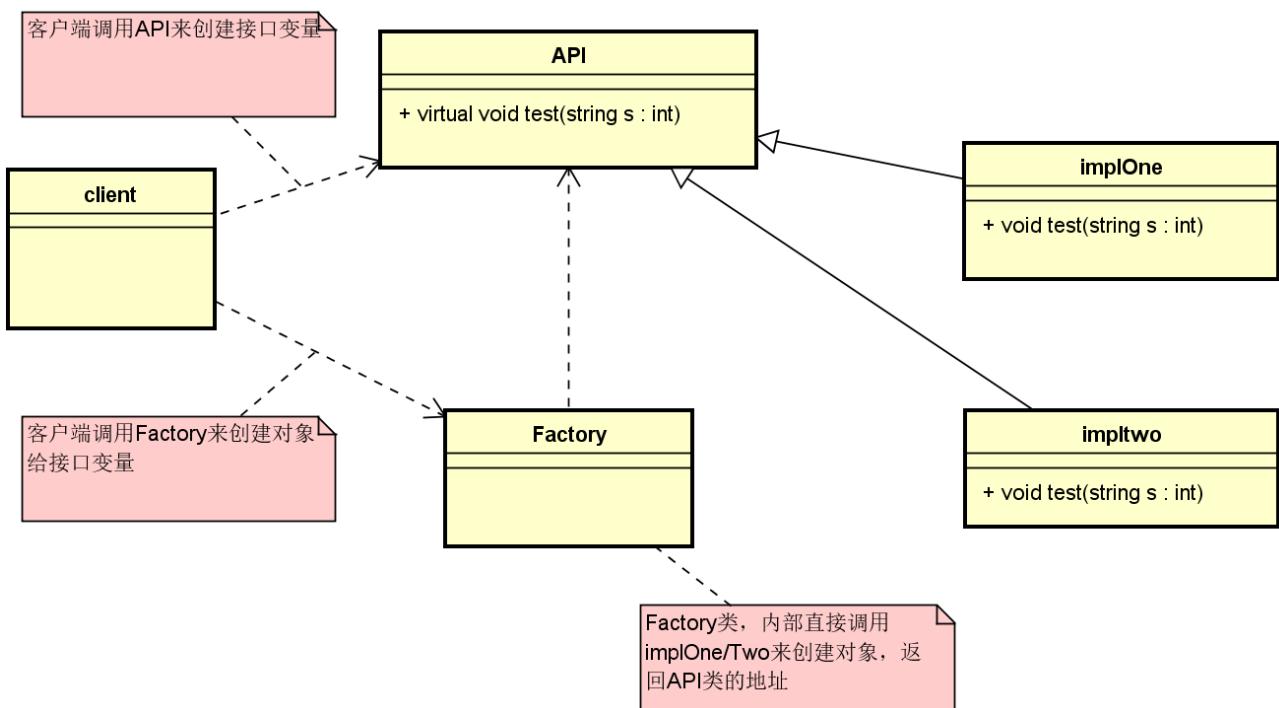
```

如果传入的是整形 0，就调用整形形参

111111 222222

这就是说明了不能将 nullptr 赋值给整形，nullptr 在 C++11 中是给指针赋值为控制。而 NULL 或者 0 是给整形赋 0，所以在 C 语言给指针赋空值的 NULL，不能在 C++11 中给指针赋控制。C++11 只能用 nullptr 给指针赋空值。

简单工厂方法使用



```

class API
{
public:
    virtual void test(string s) = 0; //API是父类，创建个虚函数，交由子类来实现
protected:
    API(); //屏蔽构造函数，不让外部访问
};

class implOne : public API //implOne 继承API
{
public:
    void test(string s) //子类实现父类的虚函数
    {
        cout << "现在是implOne类: " << s << endl; //执行implOne实现的test
    }
};

class implTwo : public API //implTwo 继承API
{
public:
    void test(string s) //子类实现父类的虚函数
    {
        cout << "现在是implTwo类: " << s << endl; //执行implTwo实现的test
    }
};

class Factory
{
public:
    static API* CreateAPI(int type)
    {
        API *pApi = nullptr; //指针为空
        if(type == 1)
        {
            pApi = new implOne();
        }
        if(type == 2)
        {
            pApi = new implTwo();
        }
        return pApi; //返回创建的对象地址给外部调用者
    }
};

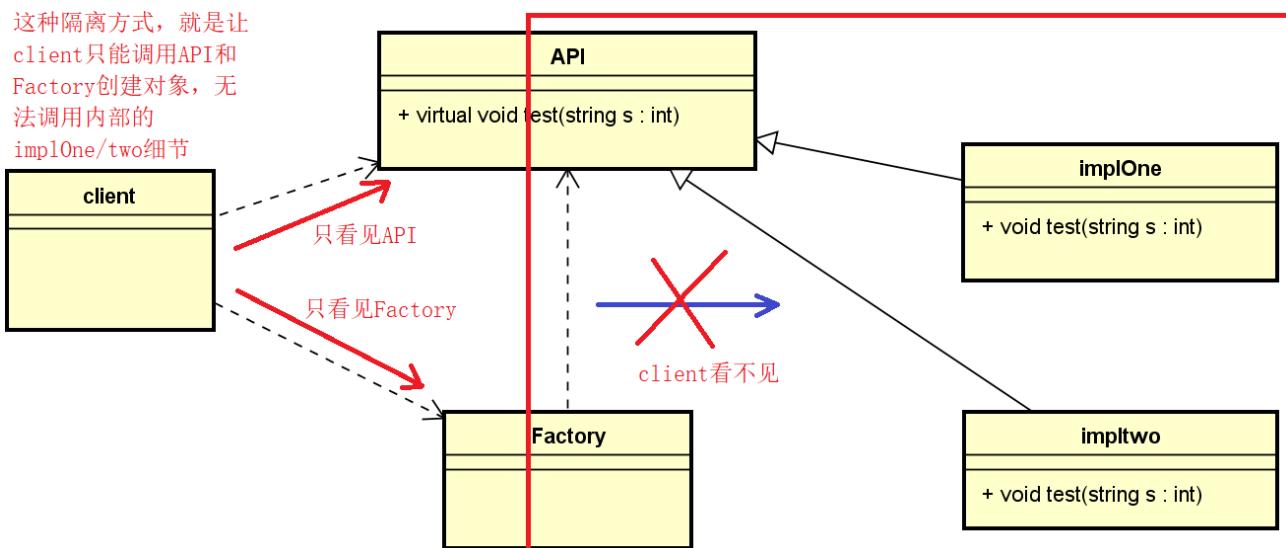
```

The diagram illustrates the Factory Method design pattern. It shows four classes: API, Factory, and two implementation classes, implOne and implTwo. The API class defines a virtual method test(string s). The Factory class contains a static method CreateAPI(int type) that returns a pointer to an API object. This method uses a conditional statement to create either an instance of implOne or implTwo. Both implOne and implTwo classes inherit from API and implement the virtual test method. A callout box provides a note: "Factory类就是将implOne和implTwo封装在CreateAPI里面，外部客户端无法直接调用implOne/two，只能用Factory间接调用".

```
//client 客户端调用
int main()
{
    API *papi = Factory::CreateAPI(1); //分配第一种implOne类创建的对象
    papi->test("1234");
    API *papi2 = Factory::CreateAPI(2); //分配第二种implTwo类创建的对象
    papi2->test("1234");
    return 0;
}
```

现在是implOne类: 1234
现在是implTwo类: 1234

这种隔离方式，就是让
client只能调用API和
Factory创建对象，无
法调用内部的
implOne/two细节



```
API *papi = Factory::CreateAPI(1); //分配第一种implOne类创建的对象
papi->test("1234");
API *papi2 = Factory::CreateAPI(2); //分配第二种implTwo类创建的对象
papi2->test("1234");
```

这种调用方式，可以在给别人写文档的时候规定

Factory::CreateAPI(1) 写数据到 txt 文件
Factory::CreateAPI(2) 写数据到 xml 文件

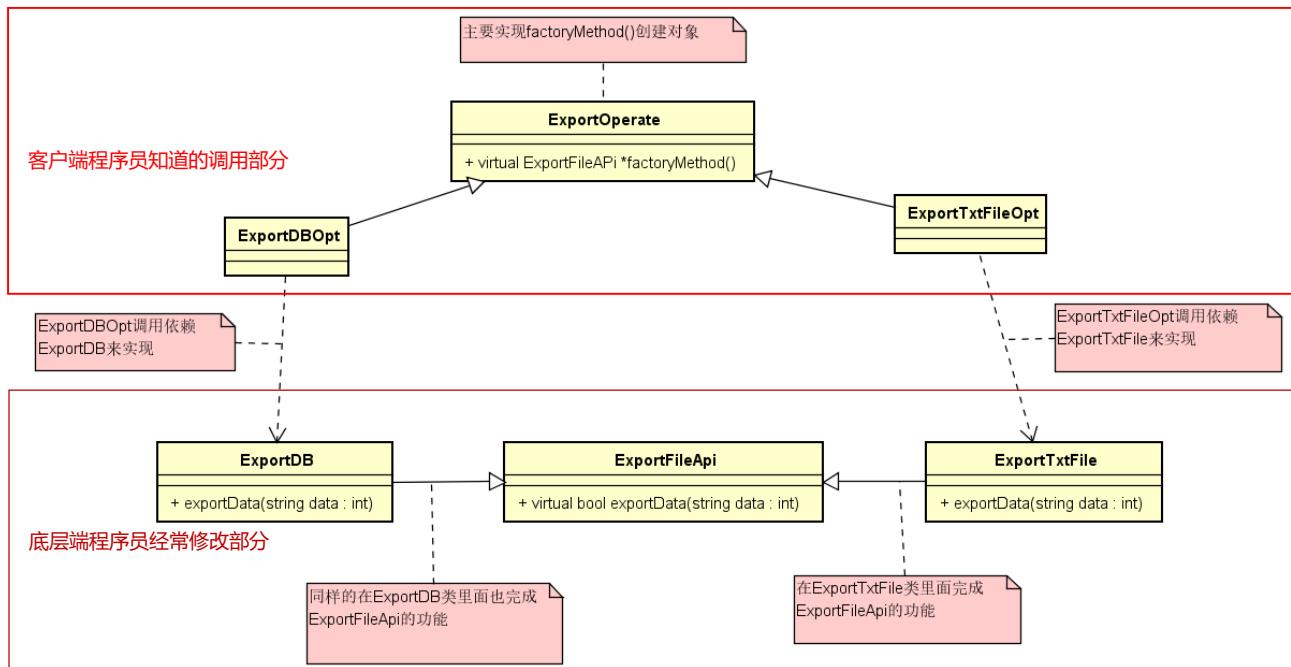
工厂模式

工厂模式和简单工厂方法是有区别的

简单工厂方法是对接口进行编程，就是对父类的虚函数进行子类实现。

工厂模式比工厂方法又升了一级，工厂模式主要应用在类似导出数据的需求上，比如导出 CSV 格式，excel 格式，xml 格式和 json 格式的东西。

我们希望做导出数据功能的客户端程序员只需要知道 ExportOperate 类和 ExportTxtFileOpt 创建对象。

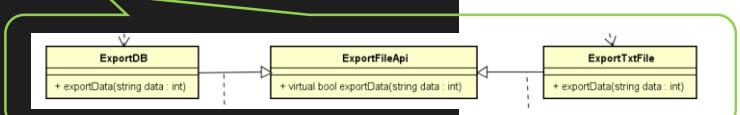


这样客户端程序和底层端代码就完全分离了。客户端使用 `ExportDBOpt/TxtFileOpt` 这种永远不变的接口。

```
#include <iostream>
using namespace std;

class ExportFileApi //外部客户端的程序员只调用这个类就可以导出数据
{
public:
    virtual bool exportData(string data) = 0;
protected:
    ExportFileApi(){}; //又不允许用这个ExportFileApi类去创建对象
};

/*子类实现exportData函数*/
class ExportTxtFile: public ExportFileApi
{
public:
    bool exportData(string data)
    {
        cout << "正在导出数据" << data << "到txt文件" << endl;
        return true;
    }
};
```



```

/*生成数据到数据库*/
class ExportDB :public ExportFileAPI //生成数据也是继承API类
{
public:
    bool exportData(string data) //也是使用exportData
    {
        cout <<"正在导出数据到"<< data<< "数据库data" << endl;
        return true;
    }
};

```



```

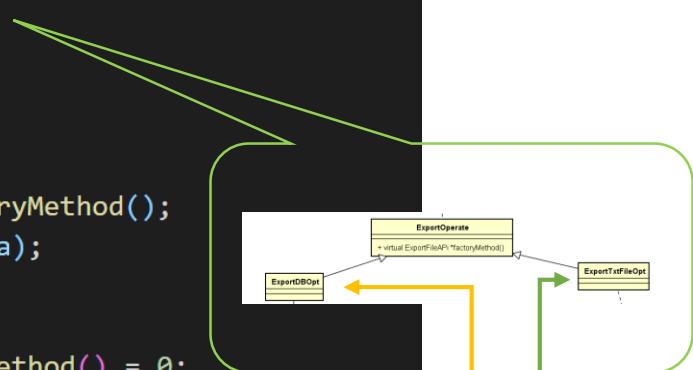
//ExportOperate 导出数据业务功能的类
class ExportOperate //它也是一个接口
{
public:
    bool exportData(string data)
    {
        ExportFileAPI* pApi = factoryMethod();
        return pApi->exportData(data);
    }
protected:
    virtual ExportFileAPI *factoryMethod() = 0;
};

//具体实现对象，完成导出工作
class ExportTxtFileOpt : public ExportOperate
{
protected:
    ExportFileAPI* factoryMethod() //这就是实现factoryMethod
    {
        return new ExportTxtFile(); //这里来创建导出对象
    }
};

class ExportDBOpt:public ExportOperate
{
protected:
    ExportFileAPI* factoryMethod() //这就是实现factoryMethod
    {
        return new ExportDB(); //这里来创建生成数据对象
    }
};

int main()
{
    ExportOperate* pOpt = new ExportTxtFileOpt();
    pOpt->exportData("xxxxzz");
    return 0;
}

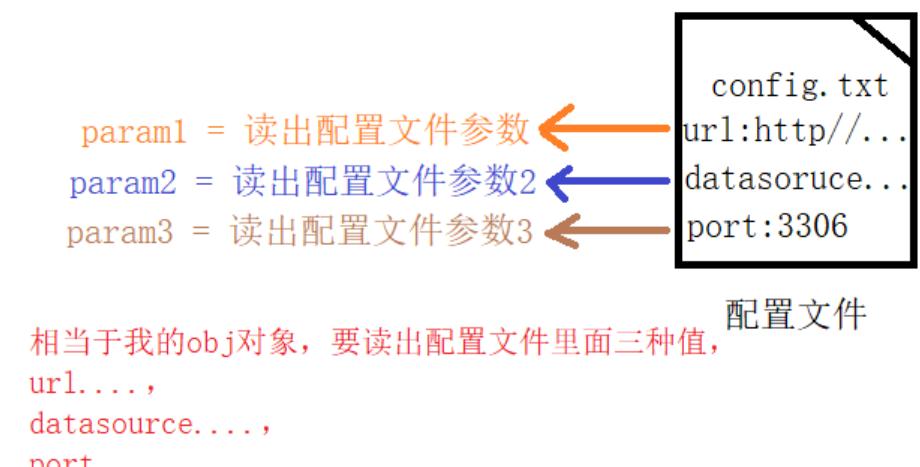
```



正在到处数据xxxxzz到txt文件

单例模式

单例模式应用场景



我们读取文件在系统里面是非常慢的操作，而且配置文件里面的数据是不会变的。因为 obj 对象可能很多地方都会用的，我们就需要将 obj 对象做成全局的，而且整个系统就只有这一个对象。不能有其它重名的对象。这就要用到单例模式了。

```
class Singleton
{
private:
    Singleton() //我们把构造函数做成私有模式，这样外部就无法进行new 这个类了，只能静态申请。
    {
        m_singer = NULL;
        cout<< "构造一个Singleton对象" << endl;
    }
private:
    static Singleton* m_singer; //定义个类指针。这是静态的
};

int main()
{
    Singleton *p = new Singleton; //因为是私有的构造函数，所以无法new
    return 0;
}
```

下面我们换种方式来做单例模式

```

class Singleton
{
private:
    Singleton() //我们把构造函数做成私有模式，这样外部就无法进行new 这个类了，只能静态申请。
    {
        m_singer = NULL;
        cout<< "构造一个Singleton对象" << endl;
    }
private:
    static Singleton* m_singer; //定义一个类指针。这是静态的
public:
    static Singleton* getInstance() //因为无法在外部调用的时候new Singleton
    {
        if(m_singer == NULL)
        {
            m_singer = new Singleton; //那我就借用getInstance函数来创建Singleton
        }
        return m_singer; //返回创建的Singleton对象地址
    }
};

Singleton *Singleton::m_singer = NULL; //必须在全局区，提前将m_singer 设置为空
int main()
{
    Singleton* p1 = Singleton::getInstance(); //我直接调用Singleton类里面getInstance创建对象
    return 0;
}

```

采用间接方式来创建 Singleton 对象

这就是前面章节讲的静态成员变量，不清楚，请看《计算一个程序里面对对象创建次数...》章节

构造一个Singleton对象

程序执行成功

下面我多创建几次对象，看看是不是不同的对象

```

int main()
{
    Singleton* p1 = Singleton::getInstance(); //我创建对象1
    Singleton* p2 = Singleton::getInstance(); //我创建对象2

    cout<< p1 << endl;
    cout<< p2 << endl;
    return 0;
}

```

构造一个Singleton对象

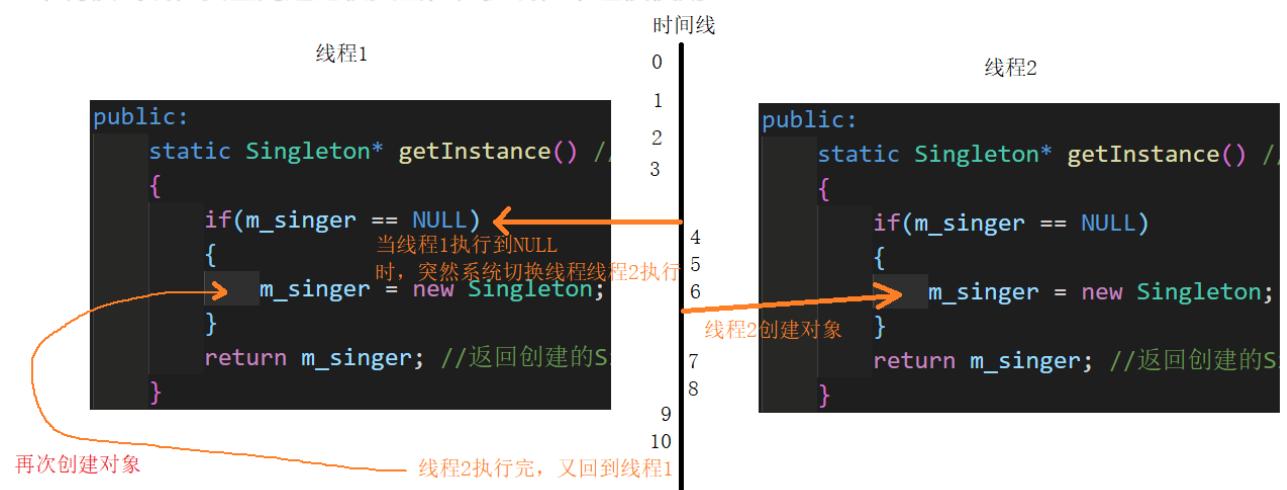
0xfb6f90

0xfb6f90

我创建了两个对象，但是两个对象地址都是一样的。

这就是单例模式，一个对象只能创建一次，然后系统从头到尾就只有这一个对象。

1. 单例模式线程安全问题比较突出，在多线程中谨慎使用



这时候，就不止有线程2创建的单例对象，线程1也创建了单例对象，一个系统有两个相同的单例对象，就会出现问题
这就是懒汉单例模式线程安全问题

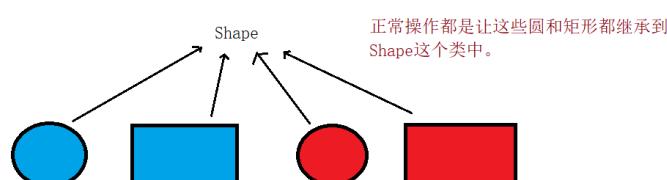
所以我们必须在创建单例对象的时候加入互斥量，或者临界段保护方式。

```
public:
    static Singleton* getInstance() //因为无法在外部调用的时候new Singleton
    {
        if(m_singer == NULL)
        {
            //加入临界段启动
            if(m_singer == NULL) //再判断一次是否被其它线程创建
                m_singer = new Singleton; //那我就借用getInstance函数来创建Singleton
            //取消临界段
        }
        return m_singer; //返回创建的Singleton对象地址
    }
```

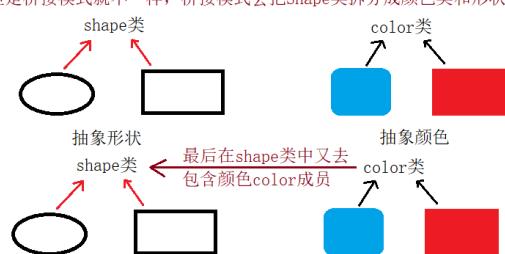
桥接模式

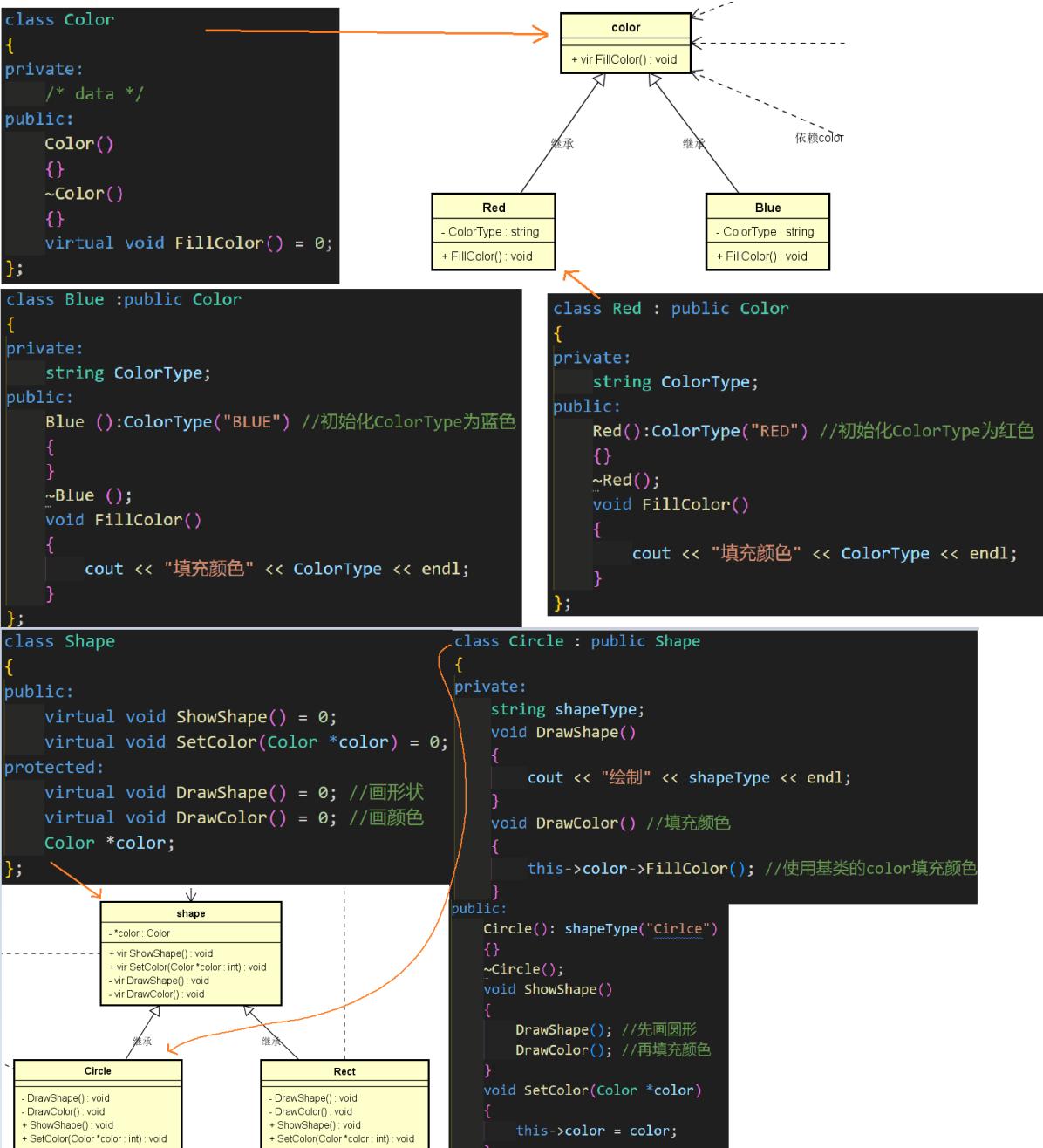
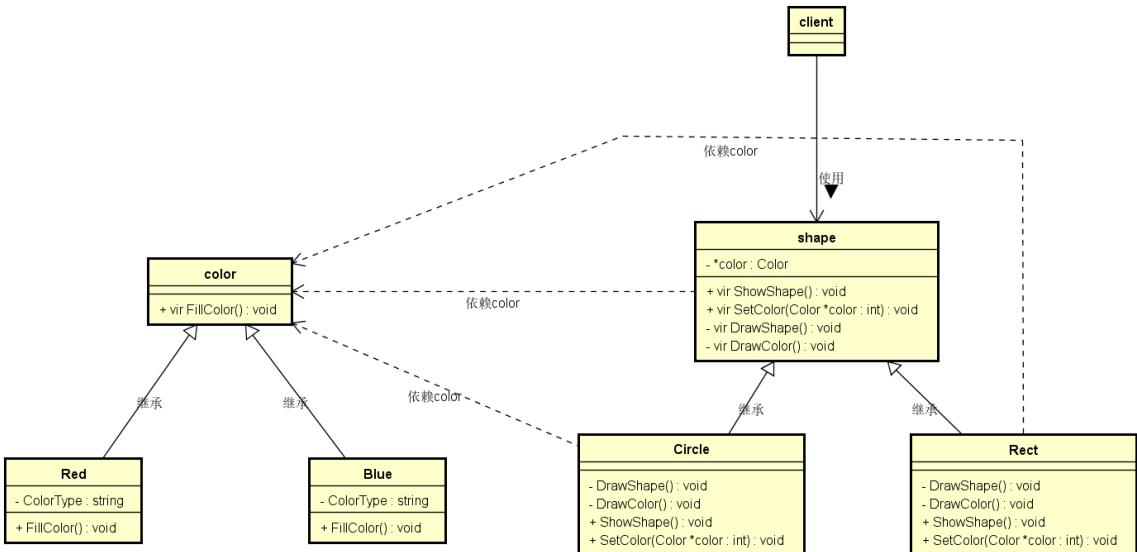
桥接模式是结构性设计模式，将复杂的大类，拆分成不同的小类，然后采用类的组合去实现。
结构性设计模式主要突出类与类的组合关系

比如我们要画不同颜色的圆和不同颜色的矩形，甚至圆也有很多种类型，矩形也有很多种类型



但是桥接模式就不一样，桥接模式会把Shape类拆分成颜色类和形状类

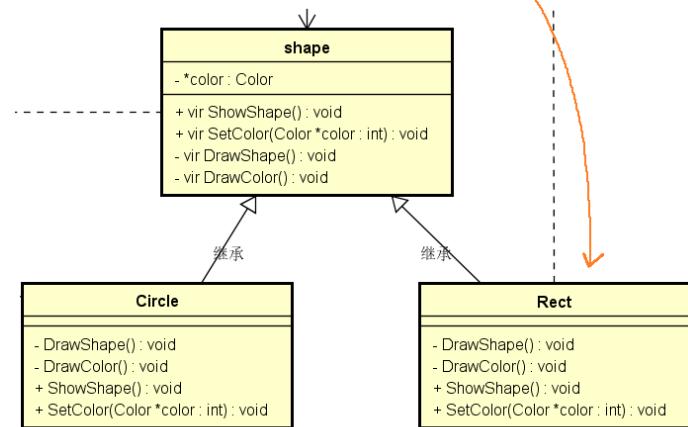




```

class Rect : public Shape
{
private:
    string shapeType;
    void DrawShape()
    {
        cout << "绘制：" << shapeType << endl;
    }
    void DrawColor()
    {
        this->color->FillColor();
    }
public:
    Rect():shapeType("Rect")
    {
    }
    void ShowShape()
    {
        DrawShape(); //先画矩形
        DrawColor(); //再填充颜色
    }
    void SetColor(Color *color)
    {
        this->color = color;
    }
};

```



```

int main(void)
{
    cout << "绘制矩形：" << endl;
    Shape* shape = new Rect;
    shape->SetColor(new Blue);
    shape->>ShowShape();
    return 0;
}

```

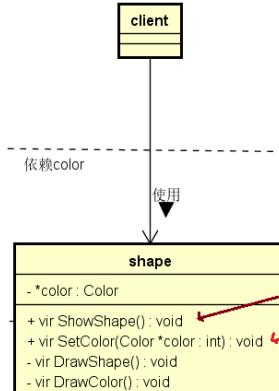
如果我想画矩形，填充蓝色

```

int main(void)
{
    cout << "绘制圆形：" << endl;
    Shape* shape = new Circle; //创建圆
    shape->SetColor(new Red); //设置圆为红色
    shape->>ShowShape();
    return 0;
}

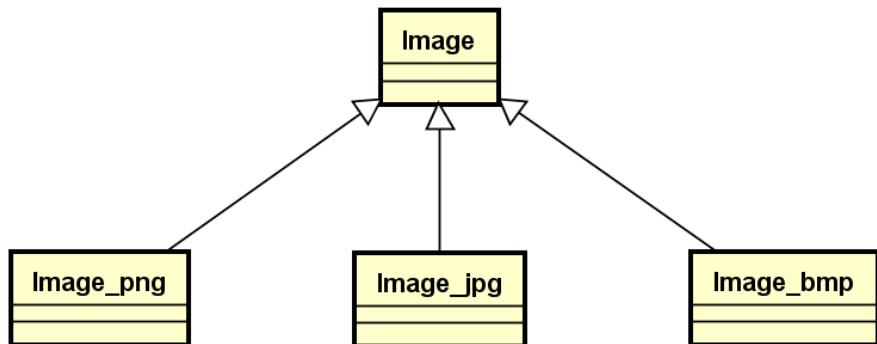
```

如果我想画圆，填充红色



桥接模式实际案例 1

针对不同格式图像文件读取，并显示相关内容



```
//图像显示类
class Image
{
public:
    //根据pData(缓冲区) 中的内容以及iDataLen所指示的缓冲区的长度，将这些数据显示出来
    void draw(const char* pfilename)
    {
        int iLen = 0;
        char* pData = parsefile(pfilename, iLen);
        if (iLen > 0)
        {
            cout << "显示pData所指向的缓冲区中的图像数据。" << endl;
            //....
            delete pData; //模拟代码中因为pData的内存是new出来的，所以这里需要释放该内存
        }
    }
    virtual ~Image() {} //做父类时析构函数应该为虚函数
private:
    //根据文件名分析文件内容，每个子类因为图像文件格式不同，会有不同的读取和处理代码
    virtual char* parsefile(const char* pfilename, int& iLen) = 0;
};

//处理png格式图像文件的显示
class Image_png : public Image
{
private:
    //读取png文件内容并进行解析，最终整理成统一的二进制数据格式返回
    virtual char* parsefile(const char* pfilename, int& iLen)
    {
        //以下是模拟代码，模拟从图像文件中读取到了数据，最终转换成了100个字节的数据格式
        //（事先约定好的格式规范）并返回
        cout << "开始分析png文件中的数据并将分析结果放到pData中。";
        iLen = 100;
        char* presult = new char[iLen];
        //...
        return presult;
    }
};
```

```

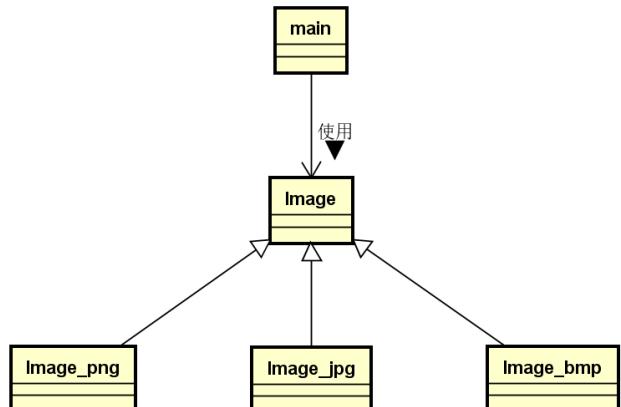
//处理jpg格式图像文件的显示
class Image_jpg :public Image
{
private:
    virtual char* parsefile(const char* pfilename, int& iLen)
    {
        cout << "开始分析jpg文件中的数据并将分析结果放到pData中。";
        //.....
    }
};

//处理jpg格式图像文件的显示
class Image_bmp :public Image
{
private:
    virtual char* parsefile(const char* pfilename, int& iLen)
    {
        cout << "开始分析bmp文件中的数据并将分析结果放到pData中。";
        //.....
    }
};

int main(void)      根据new的不同子类让pImg多态的
{
    Image* pImg = new Image_png();
    pImg->draw("c:\\somedir\\filename.jpg");
    //释放资源
    delete pImg;

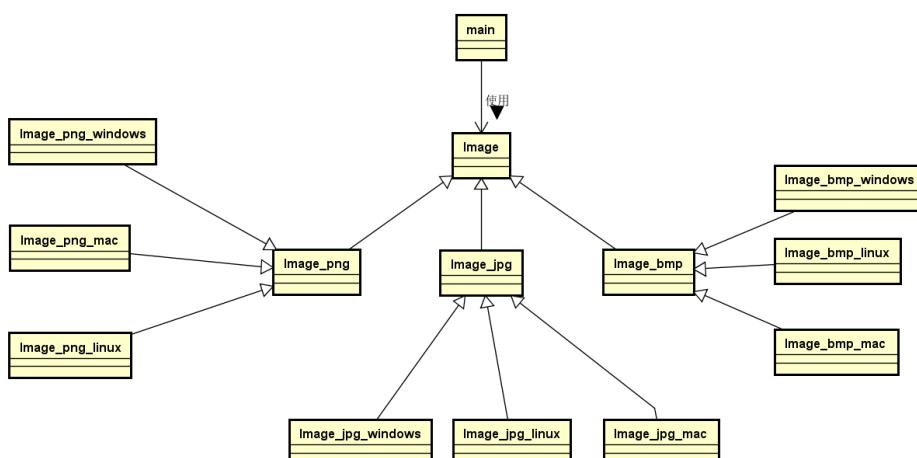
    return 0;
}

```



桥接模式实际案例 2

我现在想三种不同格式的图片，支持在三种操作系统中使用。



```

//操作系统类型抽象类
class ImageOS
{
public:
    virtual void draw(char* pData, int iLen) = 0;
    virtual ~ImageOS() {} //做父类时析构函数应该为虚函数
};

//Windows操作系统
class ImageOS_Windows :public ImageOS
{
public:
    virtual void draw(char* pData, int iLen)
    {
        cout << "在Windows操作系统下显示pData所指向的缓冲区中的图像数据。" << endl;
        //...具体处理代码略
    }
};

//Linux操作系统
class ImageOS_Linux :public ImageOS
{
public:
    virtual void draw(char* pData, int iLen)
    {
        cout << "在Linux操作系统下显示pData所指向的缓冲区中的图像数据。" << endl;
        //...具体处理代码略
    }
};

//Mac操作系统
class ImageOS_Mac :public ImageOS
{
public:
    virtual void draw(char* pData, int iLen)
    {
        cout << "在Mac操作系统下显示pData所指向的缓冲区中的图像数据。" << endl;
        //...具体处理代码略
    }
};

```

```

//图像文件格式抽象类
class ImageFormat
{
public:
    形参传入了前面操作系统类对象
    ImageFormat(ImageOS* pimgos) :m_pImgOS(pimgos) {} //构造函数
    virtual void parsefile(const char* pfilename) = 0; //根据文件名分析文件内容,
    //每个子类因为图像文件格式不同，会有不同的读取和处理代码
    virtual ~ImageFormat() {} //做父类时析构函数应该为虚函数
protected:
    ImageOS* m_pImgOS; //委托←相当于图像这个类依赖了操作系统类
};

```

```

//png格式的图像文件
class Image_png :public ImageFormat
{
public:
    Image_png(ImageOS* pimgos) :ImageFormat(pimgos) {} //构造函数
    virtual void parsefile(const char* pfilename)
    {
        cout << "开始分析png文件中的数据并将分析结果放到pData中。";
        int iLen = 100;
        char* presult = new char[iLen];
        m_pImgOS->draw(presult, iLen);

        //释放资源
        delete presult;
    }
};

//jpg格式的图像文件
class Image_jpg :public ImageFormat
{
public:
    Image_jpg(ImageOS* pimgos) :ImageFormat(pimgos) {} //构造函数
    virtual void parsefile(const char* pfilename)
    {
        cout << "开始分析jpg文件中的数据并将分析结果放到pData中。";
        //.....
    }
};

//bmp格式的图像文件
class Image_bmp :public ImageFormat
{
public:
    Image_bmp(ImageOS* pimgos) :ImageFormat(pimgos) {} //构造函数
    virtual void parsefile(const char* pfilename)
    {
        cout << "开始分析bmp文件中的数据并将分析结果放到pData中。";
        //.....
    }
};

int main(void)
{
    ImageOS* pimgos_windows = new ImageOS_Windows(); //针对Windows操作系统
    ImageFormat* pimg_png = new Image_png(pimgos_windows);
    //运行时把图像文件格式png和操作系统windows动态组合到一起。
    pimg_png->parsefile("c:\\\\somedir\\\\filename.jpg");

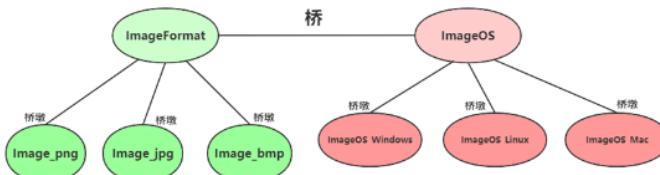
    //释放资源
    delete pimg_png;
    delete pimgos_windows;
    return 0;
}

```

开始分析png文件中的数据并将分析结果放到pData中。在Windows操作系统下显示pData所指向的缓冲区中的图像数据。

桥接模式实际案例 3

桥接模式像被多个桥墩连接起来的一座桥，怎么实现？



当在C++中使用桥接模式时，可以考虑一个简单的例子：图形界面库。假设我们有一个图形界面库，它需要支持不同的操作系统和不同的应用程序界面。我们可以使用桥接模式来处理这种情况。

首先，我们定义一个抽象类 `Window` 作为抽象部分，它包含了一些基本的窗口操作方法：

```
class Window {
public:
    virtual void draw() = 0;
    virtual void open() = 0;
    virtual void close() = 0;
    virtual ~Window() {}
};
```

然后，我们定义一个实现类 `WindowImpl` 作为实现部分，它包含了窗口的具体实现

```
class WindowImpl {
public:
    virtual void drawWindow() = 0;
    virtual void openWindow() = 0;
    virtual void closeWindow() = 0;
    virtual ~WindowImpl() {}
};
```

接下来，我们创建一个桥接类 `WindowWithBorder`，它将抽象部分和实现部分连接起来：

```
class WindowWithBorder : public Window {
public:
    WindowWithBorder(WindowImpl* impl) : m_impl(impl) {}
    void draw() override {
        m_impl->drawWindow();
    }
    void open() override {
        m_impl->openWindow();
    }
    void close() override {
        m_impl->closeWindow();
    }
private:
    WindowImpl* m_impl;
};
```

最后，我们创建具体的实现类，比如 `LinuxWindowImpl` 和 `WindowsWindowImpl`，它们分别实现了 `WindowImpl` 接口

```
class LinuxWindowImpl : public WindowImpl {
public:
    void drawWindow() override {
        // 在Linux上绘制窗口
    }
    void openWindow() override {
        // 在Linux上打开窗口
    }
    void closeWindow() override {
        // 在Linux上关闭窗口
    }
};

class WindowsWindowImpl : public WindowImpl {
public:
    void drawWindow() override {
        // 在Windows上绘制窗口
    }
    void openWindow() override {
        // 在Windows上打开窗口
    }
    void closeWindow() override {
        // 在Windows上关闭窗口
    }
};
```

下面是一个更完整的例子

```

// 定义抽象部分
class Window {
public:
    virtual void draw() = 0;
    virtual void open() = 0;
    virtual void close() = 0;
    virtual ~Window() {}
};

// 定义实现部分
class WindowImpl {
public:
    virtual void drawWindow() = 0;
    virtual void openWindow() = 0;
    virtual void closeWindow() = 0;
    virtual ~WindowImpl() {}
};

// 桥接类连接抽象部分和实现部分
class WindowWithBorder : public Window {
public:
    WindowWithBorder(WindowImpl* impl) : m_impl(impl) {}

    void draw() override {
        m_impl->drawWindow();
    }

    void open() override {
        m_impl->openWindow();
    }

    void close() override {
        m_impl->closeWindow();
    }

private:
    WindowImpl* m_impl;
};

// 具体的实现类
class LinuxWindowImpl : public WindowImpl {
public:
    void drawWindow() override {
        // 在 Linux 上绘制窗口
        std::cout << "Drawing window in Linux style" << std::endl;
    }

    void openWindow() override {
        // 在 Linux 上打开窗口
        std::cout << "Opening window in Linux style" << std::endl;
    }

    void closeWindow() override {
        // 在 Linux 上关闭窗口
        std::cout << "Closing window in Linux style" << std::endl;
    }
};

class WindowsWindowImpl : public WindowImpl {
public:
    void drawWindow() override {
        // 在 Windows 上绘制窗口
        std::cout << "Drawing window in Windows style" << std::endl;
    }

    void openWindow() override {
        // 在 Windows 上打开窗口
        std::cout << "Opening window in Windows style" << std::endl;
    }

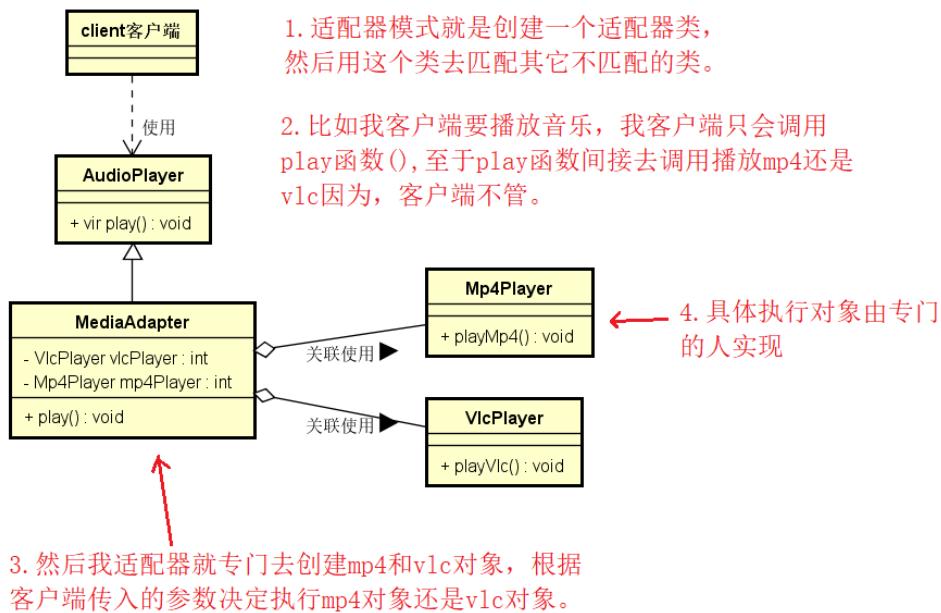
    void closeWindow() override {
        // 在 Windows 上关闭窗口
        std::cout << "Closing window in Windows style" << std::endl;
    }
};

int main() {
    // 创建使用 Linux 风格实现的窗口
    WindowImpl* linuxImpl = new LinuxWindowImpl();
    Window* linuxWindow = new WindowWithBorder(linuxImpl);
    linuxWindow->draw();
    linuxWindow->open();
    linuxWindow->close();

    // 创建使用 Windows 风格实现的窗口
    WindowImpl* windowsImpl = new WindowsWindowImpl();
    Window* windowsWindow = new WindowWithBorder(windowsImpl);
    windowsWindow->draw();
    windowsWindow->open();
    windowsWindow->close();
    delete linuxWindow;
    delete linuxImpl;
    delete windowsWindow;
    delete windowsImpl;
    return 0;
}

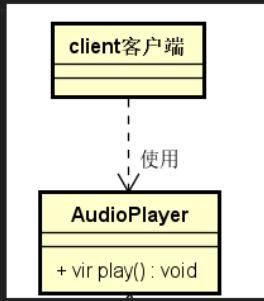
```

适配器模式



//目标接口(客户端使用)

```
class AudioPlayer
{
private:
    /* data */
public:
    AudioPlayer(){}
    ~AudioPlayer(){}
    virtual void play(string audioType, string fileName) = 0;
};
```

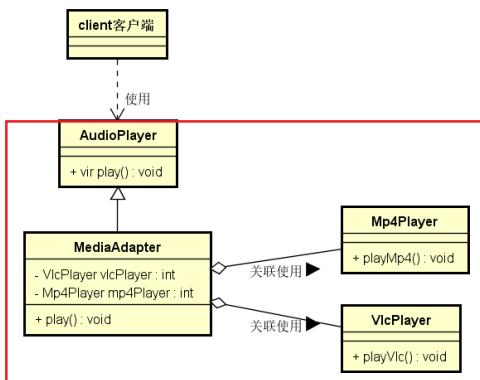


//适配者类- Vlc播放器 这就是具体实现的对象类

```
class VlcPlayer
{
private:
    /* data */
public:
    VlcPlayer(){}
    ~VlcPlayer(){}
    void playVlc(string fileName)
    {
        cout << "播放vlc文件: " << fileName << endl;
    }
};
```



```
//适配者类 - Mp4播放器 这也是专门的人实现的类
class Mp4Player
{
public:
    void playMp4(string fileName)
    {
        cout << "播放Mp4文件: " << fileName << endl;
    }
};
```



//适配类

```
class MediaAdapter : public AudioPlayer
{
private:
    VlcPlayer vlcPlayer;
    Mp4Player mp4Player;
public:
    MediaAdapter(){}
    ~MediaAdapter(){}
    void play(string audioType, string fileName) override
    {
        if(audioType == "vlc")
        {
            vlcPlayer.playVlc(fileName);
        }
        else if(audioType == "mp4")
        {
            mp4Player.playMp4(fileName);
        }
        else
        {
            cout << "不支持的音频类型: " << audioType << endl;
        }
    }
}
```

适配器类‘关联使用’就是让其它类在本类中定义。

这样，客户端可以根据输入的命令调用对应的函数。但是客户端还是使用play函数不变

如果有新的play功能需要适配进来，直接就在适配器类里面增加就是

```
int main(void)
{
    AudioPlayer *audioPlayer = new MediaAdapter(); //创建适配器对象
    audioPlayer->play("vlc","movie.vlc"); //客户端可以根据填入的名字,
    audioPlayer->play("mp4","movie.mp4"); //决定播放哪种格式的音乐
    delete audioPlayer;

    return 0;
}
```

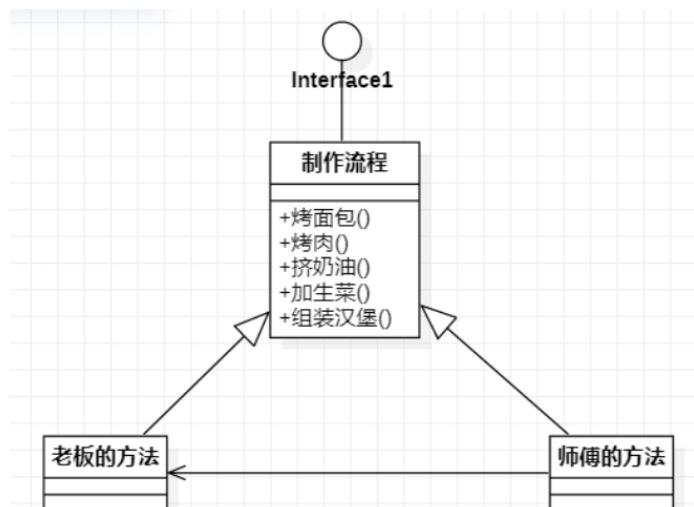
播放Vlc文件: movie.vlc
播放Mp4文件: movie.mp4

代理模式

代理模式案例如下：

今天老板出差了，临时找了个师傅帮他看店。但是老板有一个要求，一切流程必须按照老板定下来的规则走，烤面包，必须以他的方式来烤，烤肉也是一样，必须要求师傅按照老板的方式来做事。

那我们来看一下这个事件中的主要因素：代工师傅、老板、老板的流程，生产出产品。



//抽象主题 1.老板和师傅共同的接口
class abstractMan {
public:
 virtual void run() = 0;
};

//老板类，老板烤面包的方法实现
class boss :public abstractMan {
public:
 void bread() { cout << "面包" << endl; }
 void roast() { cout << "烘烤" << endl; }

 void run() {
 //暂时不插其他模式，后面会有组合模式
 bread();
 roast();
 }
}; 3. 老板烤面包的方法

2. 老板继承abstractMan类，这样后面代理函数传参的时候才能把老板创建的对象传入进去

//代理(师傅)
class proxy :public abstractMan {
public:
 proxy(abstractMan* temp) { a = temp; }

 void before(){
 //该干嘛干嘛
 }
 void after(){
 //爱干嘛干嘛
 }
 void run() {
 before();
 a->run();
 after();
 }
private:
 abstractMan* a;
};

4. 代理的师傅在创建对象的时候，就要传入老板建立的对象，这样才能得到老板烤面包的方法
5. 这是代理人师傅自己需要干的事情
6. 这是代理人师傅按照老板要求烤面包的方法

int main(void) 7. 建立老板对象，里面包含老板烤面包的方法
{
 abstractMan* bs = new boss();
 proxy* pro = new proxy(bs); //创建一个代理
 pro->run(); //代理使用老板的方法烤面包
 return 0;
}
8. 将老板烤面包方法传递给师傅
面包
烘烤

代理模式实际案例 1，客户端通过代理对象间接访问远程服务器或者远程对象

```
//假设这是远程服务端对象的头文件
class RemoteService
{
private:
    /* data */
public:
    RemoteService(){}
    ~RemoteService(){}
    virtual void foo() = 0;
};

public:
    RemoteServiceProxy (const string &host, int port): 代码
    m_host(host),m_port(port){}
    ~RemoteServiceProxy (){}
    void foo() override
    {
        connect(); //链接远程服务端
        sendRequest("foo");//向远程服务端发送请求
        string response = receiveResponse();//等待远程服务端响应
        processResponse(response);//处理响应
    }
};

int main(void)
{
    RemoteServiceProxy proxy("127.0.0.1",8080);
    proxy.foo(); //通过代理对象间接访问远程服务端对象foo()函数
}

//代理类, 用于访问远程服务器对象
class RemoteServiceProxy : public RemoteService
{
private:
    string m_host;      这个案例继承方式很简单
    int m_port;
    int m_socketFd;

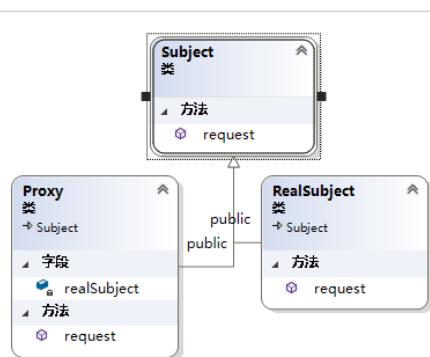
void connect()
{
    //链接远程服务的代码
}
void sendRequest(const string &request)
{
    //向远程服务的发送请求代码
}
void sendRequest(const string &request)
{
    //向远程服务的发送请求代码
}
string receiveResponse()
{
    //从远程服务的接收响应代码
}
void processResponse(const string &response)
{
    //处理响应代码
}
```

↑
这个案例代码实现了main()客户端只需要通过代理对象访问远程服务器，无需知道远程服务端代码实现细节。

我觉得这个案例和C语言函数封装调用区别不大。

代理模式实际案例 2，虚拟代理

C++虚拟代理（Virtual Proxy）模式是一种结构型设计模式，它允许你创建一个代理对象来代替一个真实对象。该代理对象可以控制对真实对象的访问，并在需要时才创建或加载真实对象。

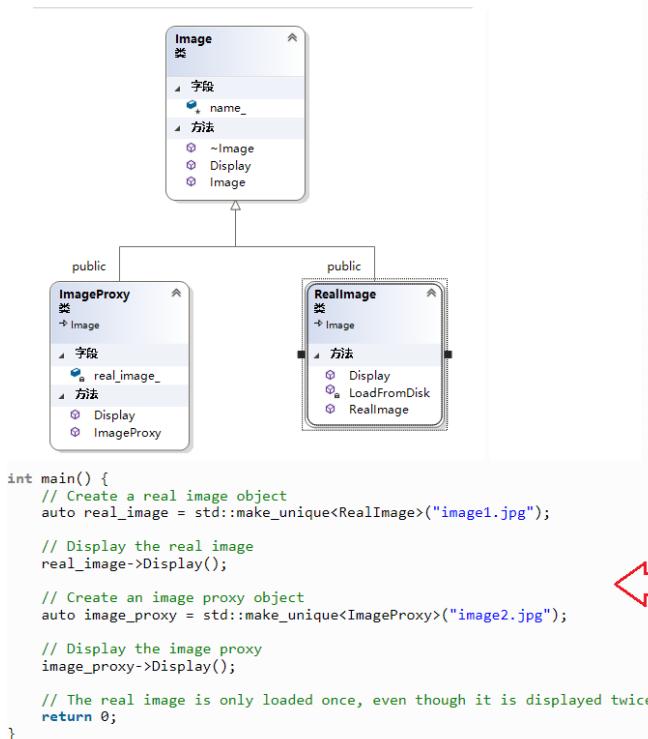


```
1 #include <iostream>
2 using namespace std;
3
4 // 定义一个抽象类Subject
5 class Subject {
6 public:
7     virtual void request() = 0;
8 };
9
10 // 定义一个真实的Subject类RealSubject
11 class RealSubject : public Subject {
12 public:
13     void request() {
14         cout << "真实的请求" << endl;
15     }
16 };
17
18 // 定义一个代理类Proxy
19 class Proxy : public Subject {
20 private:
21     RealSubject *realSubject;
22
23 public:
24     void request() {
25         if (realSubject == nullptr) {
26             realSubject = new RealSubject();
27         }
28         cout << "代理请求" << endl;
29         realSubject->request();
30     }
31 };
32
33 // 客户端代码
34 int main() {
35     Proxy proxy;
36     proxy.request();
37     return 0;
38 }
```

我觉得这个虚拟代理就是前面老板，烤面包的代理方法

代理模式实际案例 3，保护代理

C++中的保护代理（Protective Proxy）是一种结构型设计模式，其目的是控制对对象的访问。它使用一个代理对象来控制原始对象的访问，代理对象通过限制或控制原始对象的访问来提供额外的安全性和保护。



```
1 #include <iostream>
2 #include <string>
3 #include <memory>
4
5 class Image {
6 public:
7     Image(std::string name) : name_(name) {}
8     virtual void Display() = 0;
9     virtual ~Image() {};
10
11 protected:
12     std::string name_;
13 };
14
15 class RealImage : public Image {
16 public:
17     RealImage(std::string name) : Image(name) {
18         LoadFromDisk();
19     }
20
21     void Display() override {
22         std::cout << "Displaying " << name_ << std::endl;
23     }
24
25 private:
26     void LoadFromDisk() {
27         std::cout << "Loading " << name_ << " from disk" << std::endl;
28     }
29 };
30
31 class ImageProxy : public Image {
32 public:
33     ImageProxy(std::string name) : Image(name) {}

34     void Display() override {
35         if (real_image_ == nullptr) {
36             real_image_ = std::make_unique<RealImage>(name_);
37         }
38         real_image_->Display();
39     }
40
41 private:
42     std::unique_ptr<RealImage> real_image_;
43 };
44
```

观察者模式



代码示例如下：

```
//定义显示器类
class Monitor{
public:
    void print(int v) 接收int类型股票价格
    {
        cout << "显示器内容: " << v << endl;
    }
};
```

```
//这个Stock类就是被所有观察者观察的对象
class Stock{
    int price = 20; 3. 修改当前股票价格
public:
    void setPrice(int v) //更改股票值
    {
        price = v;
    }
};
```

4. 现在有个疑问，stock既然是可以更改股票的价格。那我们是否可以调用显示器类和广告类的接口来同步更新股票价格呢？当然是可以的。修改如右边代码：

```
//这个Stock类就是被所有观察者观察的对象
class Stock{
    int price = 20;
    Monitor *monitor;
    Billboard *board;
public:
    Stock(Monitor *monitor,Billboard *board):monitor(monitor),board(board){}
    void setPrice(int v) //更改股票值
    {
        price = v;
        board->display(v);
        monitor->print(v);
    }
};
```

```
//定义广告牌类
class Billboard{
public:
    void display(int v)
    {
        cout << "电子广告牌内容: " << v << endl;
    }
};
```

```
//这个Stock类就是被所有观察者观察的对象
class Stock{
    int price = 20;
    Monitor *monitor; 5. 在修改股价的同时，我们定义显示器类和广告类指针，用于在stock中同步修改股价显示
    Billboard *board; 6. 股价修改的同时，修改广告牌和显示器。
public:
    void setPrice(int v) //更改股票值
    {
        price = v;
        board->display(v); 7. 注意*monitor和*board是指针，不能直接使用，必须要赋予地址
        monitor->print(v);
    }
};
```

8. 在初始化的时候将外部传入的显示器对象和广告牌对象的地址传入进来

记住，构造函数执行一定在public区域定义

```

int main()
{
    Monitor monitor; 观察者
    Billboard board; 观察者
    Stock stock(&monitor,&board); //C++11可以大括号初始化{&monitor,&board}
    stock.setPrice(50);           被观察者Stock
    return 0;                    更改股价，同时显示器和广告牌也会被更改
}

```

电子广告牌内容: 50
显示器内容: 50

现在这种方法存在一个问题，就是类与类之间的耦合度太深

The diagram illustrates the tight coupling between three classes: Stock, Monitor, and Billboard. A dashed line connects the Stock class to both the Monitor and Billboard classes. Annotations explain the coupling:

- 1. 当我显示器类的 print 函数改名字了
- 2. 当我广告类里面函数改名字了
- 3. 这时候我又必须去 Stock 类里面去修改显示器类和广告类的函数名。操作太麻烦。
- 4. 如果我又加入了新的显示类，这里又要增加代码。

类之间的紧耦合

- Stock 类会被频繁修改:
 - 当显示介质的接口变化时；
 - 当有新的显示媒介加入时。

```

struct Stock {
    int price = 20;
    Monitor* monitor;
    Billboard* board;
    Stock (Monitor* monitor, Billboard* board)
        : monitor(monitor), board(board) {}
    void setPrice(int v) {
        price = v;
        board->display(v);
        monitor->print(v);
    }
};

//定义显示器类
class Monitor{ 1. 当我显示器类的
public:          print函数改名字了
    void print(int v) ↵
    {
        cout << "显示器内容: " << v << endl;
    }
};

//定义广告牌类
class Billboard{ 2. 当我广告类里面函数
public:          改名字了
    void display(int v) ↵
    {
        cout << "电子广告牌内容: " << v << endl;
    }
};

```

下面我们用观察者模式来优化这种紧耦合的代码

The diagram shows the Observer pattern being applied to the code. The Stock class now holds a pointer to an Observer object, which can be either a Monitor or a Billboard. The Observer interface defines an update(int) method. The Monitor and Billboard classes implement their own update methods by calling the base class's update method.

```

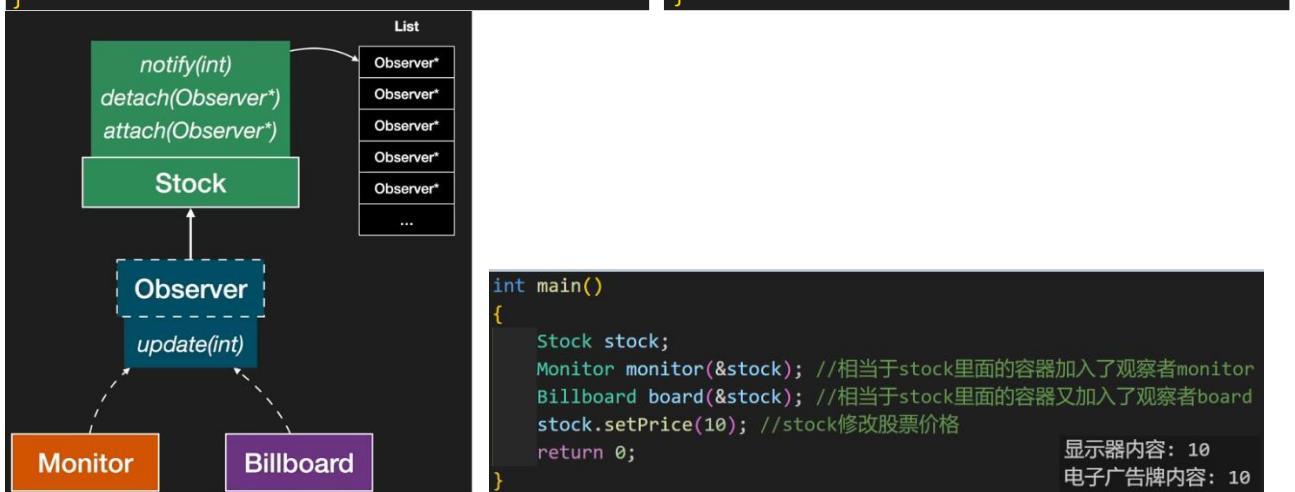
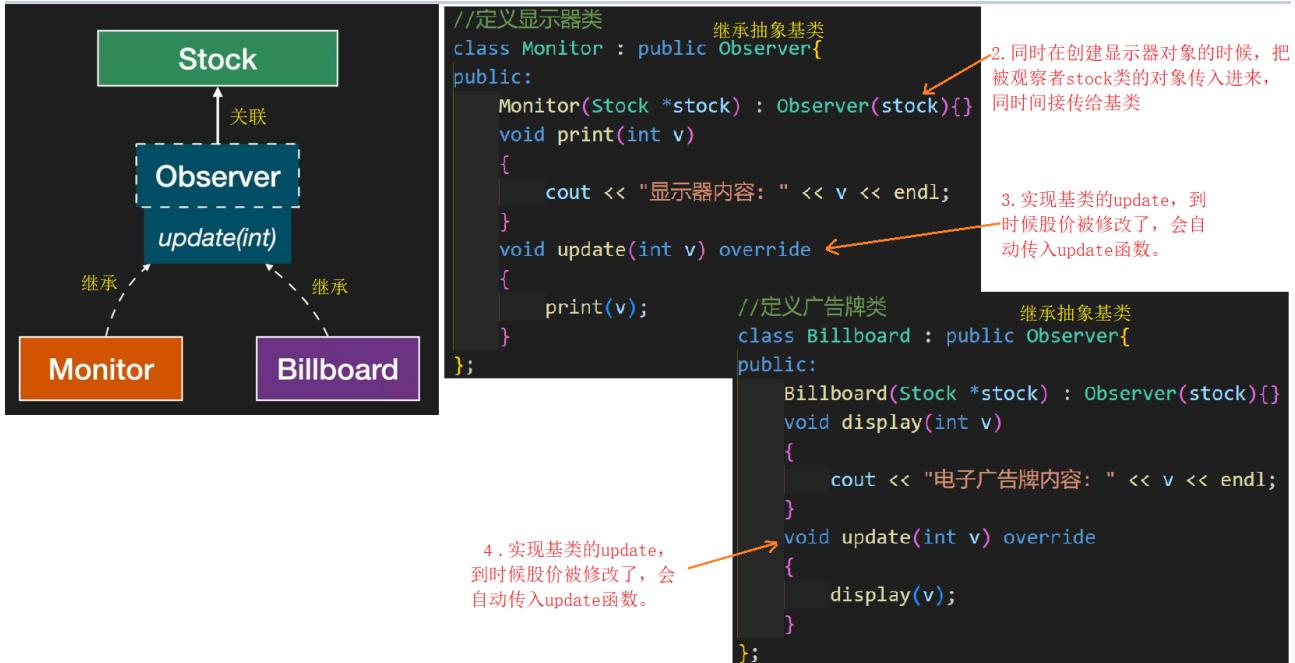
class Observer { 1. 我们定义个抽象类，作为显示器类和广告类的基类
public:
    Stock *stock;
    Observer(Stock *stock); //修改抽象基类构造函数，实现指针初始化
    virtual ~Observer();
    virtual void update(int) = 0;
};

//定义显示器类
class Monitor{
public:
    void print(int v)
    {
        cout << "显示器内容: " << v << endl;
    }
};

//定义广告牌类
class Billboard{
public:
    void display(int v)
    {
        cout << "电子广告牌内容: " << v << endl;
    }
};

class Stock {
    Monitor monitor; 观察者
    Billboard board; 观察者
    Observer observer; 观察者
    Stock (Monitor* monitor, Billboard* board)
        : monitor(monitor), board(board) {}
    void setPrice(int v) {
        price = v;
        observer->update(v);
    }
};

```



```

int main()
{
    Stock stock;
    Monitor monitor(&stock);
    Billboard board(&stock);
    stock.setPrice(10); //stock
    return 0;
}

```

从实现代码可以看出，Stock在创建的时候，不依赖其它观察者对象(Monitor)，(Billboard)。这样Stock代码就可以稳定不变。

```

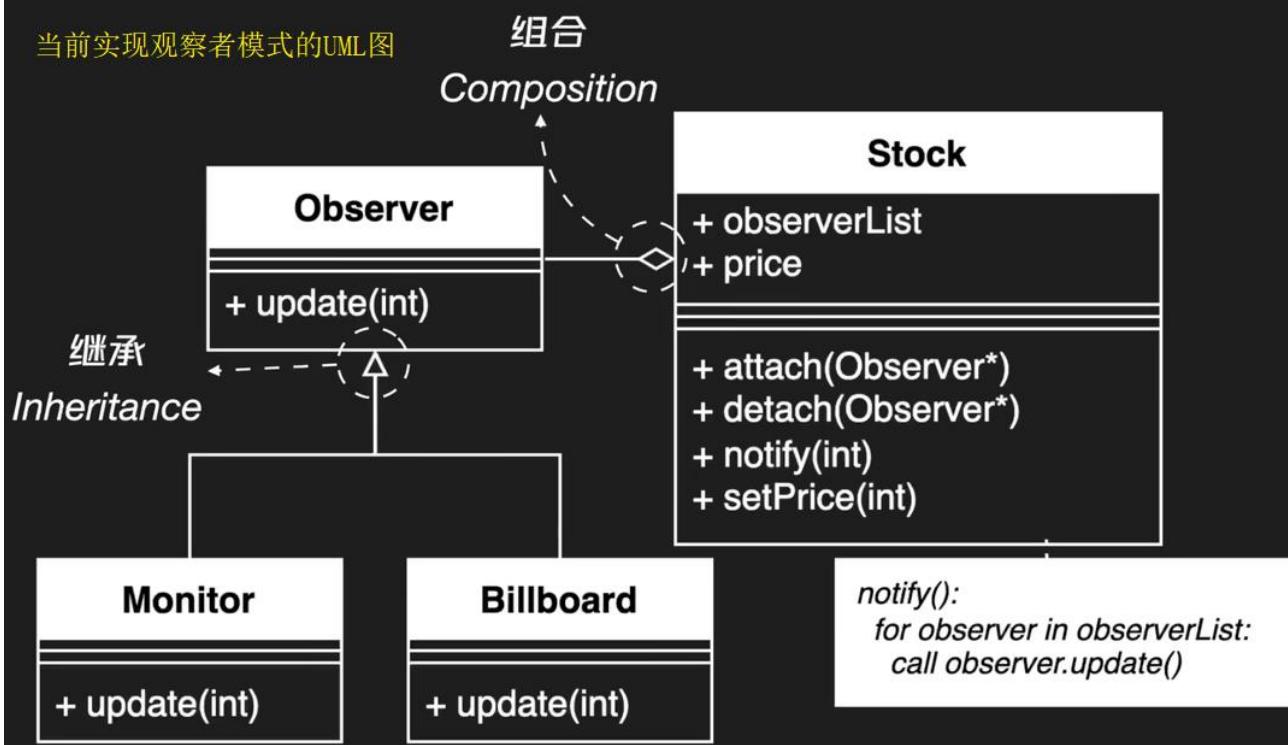
//定义广告牌类
class Billboard : public Observer{
public:
    Billboard(Stock *stock) : Observer(stock){}
    void display(int v)
    {
        cout << "电子广告牌内容: " << v << endl;
    }
    void update(int v) override
    {
        display(v);
    }
};

```

这时候不管怎么去修改广告牌显示名称，都不用去动stock里面的函数。

只需要把修改后的函数名称，放入基类抽象的统一接口函数update就行。

当前实现观察者模式的UML图



观察者模式实际案例 1，信号与槽基本设计思路

信号与槽框架基本编程思路

设定两者类，一个为观察者类，一个为被观察者类

观察者类中，定义一个对某个事件感兴趣的处理函数，一般也叫做槽函数

被观察者类中，定义一个数据结构，用来保存观察者对某一个事件id（信号）感兴趣，使用数据结构建立信号与对象之间的映射关系

被观察者类中，定义两个方法函数：

一个方法为：添加观察者与其感兴趣的事件id（信号）加入到容器中

另一个方法为：信号函数：通知事件函数执行逻辑：首先遍历容器中，有没有感兴趣的事件ID，如果有，则代表一系列的观察者，对这个事件感兴趣，那么再次遍历观察者列表，让每一个观察者执行相应的槽函数

```
class RecvBase
{
private:
    /* data */
public:
    RecvBase()
    {
        cout << "RecvBase 初始化" << endl;
    }
    virtual ~RecvBase()
    {
        cout << "RecvBase 析构执行" << endl;
    }
    virtual void slotFunctions(int msgid) = 0; // 定义虚槽函数
};
```

```
class Recv : public RecvBase
{
private:
    /* data */
public:
    Recv(/* args */){}
    ~Recv(){}
    void slotFunctions(int msgid) override
    {
        switch (msgid)
        {
            case 1:
                cout << this << "接收到1111信号，执行1信号对应槽函数" << endl;
                break;
            case 2:
                cout << this << "接收到2222信号，执行2信号对应槽函数" << endl;
                break;
            case 3:
                cout << this << "接收到3333信号，执行3信号对应槽函数" << endl;
                break;
            case 4:
                cout << this << "接收到4444信号，执行4信号对应槽函数" << endl;
                break;
            default:
                break;
        }
    }
};
```

```
class Sender
{
private:
    /* data */
public:
    Sender(){}
    ~Sender{} 信号ID 对应的槽函数指针
    map<int, list<RecvBase *>> RecvMap;
    void observerToRecvMap(int msgid, RecvBase *recv)
    {
        this->RecvMap[msgid].push_back(recv); 将元素地址存入map容器的末尾
    }
};
```

```

void senderSignals(int msgid) 信号发送函数, 发送ID
{
    auto it = RecvMap.find(msgid); 找到ID对应的槽函数
    if (it != RecvMap.end())
    {
        for(RecvBase *p : it->second)
        {
            p->slotFunctions(msgid); 执行槽函数
        }
    }
    else
    {
        cout << "接收到未知信号, 没有可执行的槽函数" << endl;
    }
}

int main()
{
    Sender sender; //信号类
    RecvBase *r1 = new Recv(); //槽接收类
    RecvBase *r2 = new Recv();
    RecvBase *r3 = new Recv();
    RecvBase *r4 = new Recv();

    sender.observerToRecvMap(1,r1); //信号为1, 槽函数为r1, 信号与槽相互绑定
    sender.observerToRecvMap(2,r2);
    sender.observerToRecvMap(3,r3);
    sender.observerToRecvMap(4,r4);

    for (size_t i = 0; i < 5; i++)
    {
        sender.senderSignals(i); //循环发送信号, 执行对应的槽函数
    }

    delete r1;
    delete r2;
    delete r3;
    delete r4;
}

```

从1还是绑定的

RecvBase 初始化
RecvBase 初始化
RecvBase 初始化
RecvBase 初始化
接收到未知信号, 没有可执行的槽函数
0x1f12e0接收到1111信号, 执行1信号对应槽函数
0x1f1300接收到2222信号, 执行2信号对应槽函数
0x6334a0接收到3333信号, 执行3信号对应槽函数
0x6334c0接收到4444信号, 执行4信号对应槽函数
RecvBase 析构执行
RecvBase 析构执行
RecvBase 析构执行

因为for循环的i=0, 是信号0, 信号0没有绑定

QT 中信号与槽，就是用观察者实现的，下面用模板来实现信号与槽函数

1.3 观察者设计模式的编程套路是什么？

套路：

1. 设定两者类，一个为观察者类，一个为被观察者类。

2. 观察者类中，定义一个对某个事件感兴趣的处理函数，一般也叫槽函数。所以QT中接收数据的槽函数在观察者类中

3. 被观察者类中，定义一个数据结构，用来保存观察者对哪一个事件id感兴趣，使用map建立对应关系。 I

也就是对哪一个信号感兴趣

4. 被观察者类中，定义两个方法函数：

可以用一个类来封装，这个类既有观察者，又有观察者对应的函数指针。用map也可以，一个键，一个值，一一对应。

一个方法为：添加观察者与其感兴趣的事件id加入到容器之中。

定义一个发送信号的函数

另一个方法为：通知事件函数执行逻辑：首先遍历容器之中，有没有感兴趣的事件ID。如果有，则代表一系列的观察者，对这个事件感兴趣，那么再次遍历观察者列表，让每一个观察者执行相应的槽函数。

这就是QT的信号与槽链接函数

```
connect(btnRegis , &QPushButton::clicked , this , &Widget::openRegisDialogWidget)
```

信号类 (被观察者)

槽类 (观察者)



注意：我们信号的类型是不统一的，有时候用的是 QPushButton，有时候又是用其它的类。

槽函数类型也是不统一的，一会传入Widget，又可能会传入其它的类。

所以信号的参数传入和槽的参数传入不能定死数据类型，只有用类模板做形参比较合适。

```
template <class TParam> ← class与 typename 的区别
class SlotBase
{
private:
    /* data */
public:
    SlotBase(){}; 1. 先定义一个抽象接口类
    ~SlotBase(){}; ←
public:
    virtual void slotFunction(TParam param) = 0;
};
```

// 定义接收者类型，和参数类型，槽类

```
template <class TRecver, class TParam>
class Slot : public SlotBase<TParam>
{
private: 2. 在构造函数中对它进行初始化
    TRecver *m_pRecver; // 定义一个接收者的指针，在构造中对其进行初始化
    void (TRecver::*m_func)(TParam param); // 定义一个接收者类中的成员函数指针
public: 3. 定义的这个函数指针，是m_pRecver这个类中的函数
    Slot(TRecver *pObj, void (TRecver::*func)(TParam)) 定义对象的时候传入两个参数
    {
        this->m_pRecver = pObj; ← 4. 对接收者对象赋值
        this->m_func = func; // 使用类外部的接收者类的对象
        // 与接收者类中的成员函数指针进行初始化
    }

    void slotFunction(TParam param) override
    {
        5. Slot构造函数构造完之后，就要用接收者类的对象指针调用类中的成员函数
        (m_pRecver->*m_func)(param); // 成员对象指针调用类内部的成员函数
    }
};
```

尽管class和typename在模板参数列表中大多数情况下可以互换使用，但它们之间存在一些关键区别：

1. 依赖类型：typename用于指定依赖类型，而class不能用于这种情况。依赖类型是指依赖于模板参数的类型，如T::iterator。
2. 关键字用途：class在模板参数中的用途仅限于声明类型参数，而typename除了这个用途外，还用于其他上下文，如指定依赖类型或模板模板参数中的类型。
3. 语义清晰性：虽然二者可互换，但在某些情况下使用typename可以提高代码的语义清晰性，尤其是在处理依赖类型时。

```

//定义一个信号类
template <typename TParam>
class Signal
{
private:
    vector< SlotBase<TParam>*> signal_vector; // 6. vector里面就放连续的槽函数
public:
    Signal(){}
    ~Signal(){}
public:
    template<class TRecver> //定义函数模板 // 7. 添加接收对象，和对应的槽函数
    void addslot(TRecver *pObj, void (TRecver::*func)(TParam))
    //加入对我主体感兴趣的观察者
    {
        signal_vector.push_back(new Slot<TRecver,TParam>(pObj,func)); // 8. 将对主体感兴趣的观察者(也就是接收对象和槽函数)放入容器
    }
    void operator()(TParam param) //仿函数(发送一个信号)
    { // 9. 同时用一个仿函数发信号
        for(SlotBase<TParam> *p : signal_vector) //枚举for循环
        {
            p->slotFunction(param); // 10. for循环寻找有没有观察者对我这个信号感兴趣的，有兴趣的，我就一一通知每个观察者。
        }
    }
};

//定义接收类1，里面func是槽函数
class Recver1
{
public:
    void func1(int param)
    {
        cout << "这是Recver1中的函数，参数为：" << param << endl;
    }
};

//定义接收类2，里面func是槽函数
class Recver2
{
public:
    void func2(int param)
    {
        cout << "这是Recver2中的函数，参数为：" << param << endl;
    }
};

```

我到时候直接用取类里面函数
&Recver1::func1的方式当作槽函数

```

class sendObj
{
    // 这是定义用来发送信号的类
public:
    Signal<int> valueChanged; // 定义signal对象
    void testSignal(int value)
    {
        valueChanged(value); ← 这个对象就是信号
    }
};

#define connect(sender , signal , recver , method) (sender)->signal.addSlot(recver,method)

使用宏定义来做信号与槽的链接。
int main(void)
{
    Recver1 *r1 = new Recver1; // 定义接收者(观察者)
    Recver2 *r2 = new Recver2; // 定义接收者(观察者)
    sendObj *sd = new sendObj; // 定义发送者

    connect(sd , valueChanged, r1, &Recver1::func1); // 信号与槽连接
    connect(sd , valueChanged, r2, &Recver2::func2);

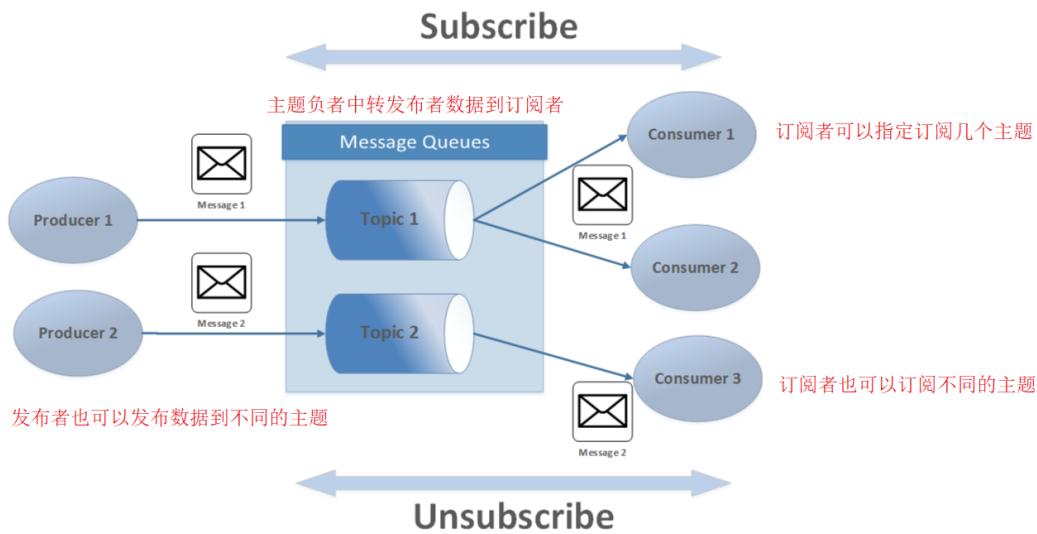
    sd->testSignal(10000); // 发送信号给10000的值到信号
    return 0;
}

```

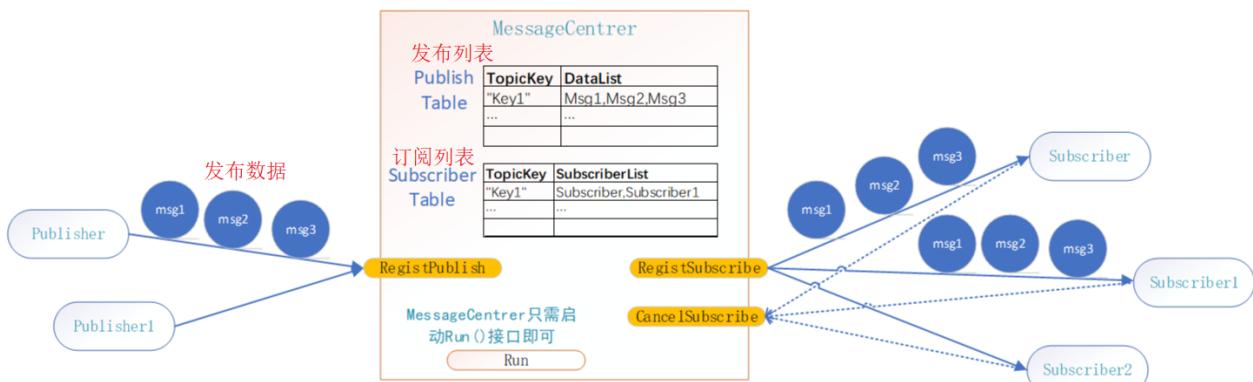
这是Recver1中的函数，参数为：10000
这是Recver2中的函数，参数为：10000

发布订阅模式

发布-订阅（Publish-Subscribe）模式是一种消息传递模式，用于构建分布式系统中的通信机制。在这种模式下，消息的发送者（发布者）和消息的接收者（订阅者）之间通过一个称为“主题（Topic）”的中介进行通信。发布者将消息发布到特定的主题上，而订阅者可以选择订阅感兴趣的主题，并在消息发布到该主题时接收消息。



本文实现主要包括3个部分：Publisher、Subscriber和MessageCenterer



代码示例如下：

```
#include <iostream>
#include <string>
#include <vector>
#include <thread>
#include <mutex>
#include <map>
#include <list>
using namespace std;

//抽象发布类
class Publisher
{
public:
    virtual void Publish(std::string Topic, void* msgdata, unsigned int datasize) = 0;
};

//抽象订阅类
class Subscriber
{
public:
    virtual void Subscribe(std::string Topic) = 0;
    virtual void UnSubscribe(std::string Topic) = 0;
    virtual void HandeEvent(std::string Topic, void * msgdata)=0;
};

#define MAX_PUBLISHES 10000

/******************
* 消息中心
*****************/
class MessageCenter
{
public:
    static MessageCenter* GetMC();
    void Run();
    void RegistPublish(std::string tpcKey, void* msgdata, unsigned int datasize);
    void RegistSubscribe(std::string tpcKey, class Subscriber* subscriber);
    void CancelSubscribe(std::string tpcKey, class Subscriber* subscriber);

private:
    MessageCenter();
    virtual ~MessageCenter();
    void CoreProcess();

private:
    std::map<std::string, std::list<void*>> mPublisher; //topickey:PublishDatas,只关心数据, 不关心是谁发布的
    std::map<std::string, std::list<class Subscriber*>> mSubscriber; //topickey:Subscribers 只关心订阅者(其后续会处理订阅的消息)

    std::unique_ptr<std::thread> mCoreProcess; //核心线程, 维护发布数据队列 + 订阅触发处理
    std::mutex mPublishMutex; //发布数据队列修改时的保护锁
    std::mutex mSubscribeMutex; //订阅者注册/取消订阅时的保护锁

    static MessageCenter* mSgMC; //消息中心单例对象
    static std::mutex mMCMutex; //线程安全单例保护锁
};

MessageCenter* MessageCenter::mSgMC = nullptr;
std::mutex MessageCenter::mMCMutex;

MessageCenter* MessageCenter::GetMC()
{
    if (mSgMC == nullptr)
    {
        std::unique_lock<std::mutex> lock(mMCMutex);
        if (mSgMC == nullptr)
        {
            volatile auto temp = new (std::nothrow) MessageCenter();
            mSgMC = temp;
        }
    }
    return mSgMC;
}
```

```

void MessageCentre::Run()
{
    mCoreProcss.reset(new std::thread(&MessageCentre::CoreProcss,this));
}
void MessageCentre::RegistPublish(std::string tpcKey, void* msgdata, unsigned int datasize)
{
    if ((this->mSubscriber.find(tpcKey) != this->mSubscriber.end()) &&
        (this->mPublisher[tpcKey].size() > MAX_PUBLISHES)) return;
    mPublishMutex.lock();
    void* tmpdata = new char[datasize];
    memcpy(tmpdata, msgdata, datasize);
    this->mPublisher[tpcKey].push_back(tmpdata);
    mPublishMutex.unlock();
}

void MessageCentre::RegistSubscribe(std::string tpcKey, Subscriber* subscriber)
{
    mSubscribeMutex.lock();
    this->mSubscriber[tpcKey].remove(subscriber);
    this->mSubscriber[tpcKey].push_back(subscriber);
    mSubscribeMutex.unlock();
}

void MessageCentre::CancelSubscribe(std::string tpcKey, Subscriber* subscriber)
{
    mSubscribeMutex.lock();
    if (this->mSubscriber.find(tpcKey) != this->mSubscriber.end())
        this->mSubscriber.find(tpcKey)->second.remove(subscriber);
    mSubscribeMutex.unlock();
}

MessageCentre::MessageCentre()
{
    this->mPublisher.clear();
    this->mSubscriber.clear();
}

MessageCentre::~MessageCentre()
{
}

void MessageCentre::CoreProcss()
{
    while (true)
    {
        auto it = this->mSubscriber.begin();
        while (it != this->mSubscriber.end())
        {
            if (this->mPublisher.find(it->first) != this->mPublisher.end())
            {
                auto itt = it->second.begin();
                while (itt != it->second.end())
                {
                    auto mpitr = this->mPublisher.find(it->first)->second.begin();
                    auto mpitrend = this->mPublisher.find(it->first)->second.end();
                    while (mpitr != mpitrend)
                    {
                        (*itt)->HandleEvent(it->first,*mpitr);
                        ++mpitr;
                    }
                    ++itt;
                }
                mPublishMutex.lock();
                auto mpitr = this->mPublisher.find(it->first)->second.begin();
                auto mpitrend = this->mPublisher.find(it->first)->second.end();
                while (mpitr != mpitrend)
                {
                    delete[](*mpitr);
                    ++mpitr;
                }
                this->mPublisher.find(it->first)->second.clear();
                this->mPublisher.erase(it->first);
                mPublishMutex.unlock();
            }
            ++it;
        }
    }
}

```

```

/******************应用测试***** */
struct Person
{
    std::string name;
    int age;
};

//发布者接口
class AppPublisher : public Publisher
{
public:
    void Publish(std::string Topic, void* msgdata, unsigned int datasize) override
    {
        MessageCentre::GetMC()->RegistPublish(Topic, msgdata, datasize);
    }
};

//订阅者接口
class AppSubscriber : public Subscriber
{
    typedef void (*HandlerFun)(void*);

public:
    AppSubscriber()
    {
        HandlerMap.clear();
        HandlerMap["Person"] = HandeEvent_Person;
        HandlerMap["Other"] = HandeEvent_Other;
    }

public:
    void Subscribe(std::string Topic) override
    {
        MessageCentre::GetMC()->RegistSubscribe(Topic, this);
    }

    void UnSubscribe(std::string Topic) override
    {
        MessageCentre::GetMC()->CancelSubscribe(Topic, this);
    }

    void HandeEvent(std::string Topic, void* msgdata) override
    {
        if (HandlerMap.find(Topic) != HandlerMap.end()) HandlerMap[Topic](msgdata);
    }

private:
    static void HandeEvent_Person(void* msgdata)
    {
        struct Person* dt = (struct Person*)msgdata;
        // 获取当前系统时间
        auto now = std::chrono::system_clock::now();
        // 转换为 time_t
        std::time_t now_time = std::chrono::system_clock::to_time_t(now);
        std::cout << dt->name << dt->age << std::ctime(&now_time) << std::endl;
    }

    static void HandeEvent_Other(void* msgdata)
    {
        struct Other* dt = (struct Other*)msgdata;
        //do something
    }
};

private:
    //TopicKey:HandlerFun
    std::map<std::string, HandlerFun> HandlerMap;
};

int main(void)
{
    MessageCentre::GetMC()->Run();
    AppSubscriber appSub;
    AppPublisher appPub;

    appSub.Subscribe("Person");
    appSub.Subscribe("Person");
    //appSub.UnSubscribe("Person");

    struct Person ps { "sma", 18 };
    struct Person ps1 { "wxq", 17 };
    appPub.Publish("Person", &ps, sizeof(ps));
    for (size_t i = 0; i < 10; i++)
    {
        //appPub.Publish("Person", &ps, sizeof(ps));
        appPub.Publish("Person", &ps1, sizeof(ps1));
    }
    return 0;
}

```

sma18 Thu May 9 14:04:34 2024
wxq17 Thu May 9 14:04:34 2024
sma18 Thu May 9 14:04:34 2024
wxq17 Thu May 9 14:04:34 2024

测试结果

命令模式



```

class Command
{
    virtual void execute() = 0;
    virtual ~Command() {}

};

//快捷指令类
class Shortcut{
private:
    vector<Command *> commands;
public:
    Shortcut(initializer_list<Command *> commands) : commands(commands){}
    //初始化commands成员，这里是用initializer_list初始化，可以简化创建对象使用语法的形式
    void addCommand(Command *cmd) //也可以在程序运行时添加任务
    {
        commands.push_back(cmd);
    }
    void run()
    {
        for (const auto & cmd : commands)
        {
            cmd->execute();
        }
    }
}

```

6. 创建Shortcut对象时初始化command私有成员
7. 程序运行动态添加Command任务
8. 当用户点击快捷命令之后，Shortcut对象内所包含的任务要被顺序执行。
9. Command需要实现具体任务对象的函数



11. 我们现在发现要获取天气，需要链接网络，和调用天气服务的程序，这时候我们需要在WeatherForecastCommand类中实现这两个功能。

```

//实现获取天气预报的功能，这个功能属于Command命令，所以继承至Command
class WeatherForecastCommand : public Command
{
public:
    void execute() override
    {
    }
}

10. 天气预报的功能我们用继承方式实现

```

11. 为了代码分离好维护，我没有在天气预报类里面实现网络链接和天气服务，我另外写类来实现。

```

//网络链接类 12. 单独实现
class Network
{
private:
    bool state = false;
public:
    void on()
    {
        if (!state)
        {
            state = true;
            cout << "网络链接" << endl;
        }
    }
};

//天气服务类 13. 单独实现
class WeatherService
{
public:
    void forecast()
    {
        cout << "获取天气信息" << endl;
    }
};

```



```

//实现获取天气预报的功能，这个功能属于Command命令，所以继承至Command
class WeatherForecastCommand : public Command
{
private:
    Network &network;           17. 所以在天气预报功能中根据自己需要，单独加入网络类和天气服务类
    WeatherService &weather;
public:
    WeatherForecastCommand(Network &network, WeatherService &weather) : network(network), weather(weather)
    {
    }

    void execute() override
    {
        network.on();
        weather.forecast();      18. 创建天气预报对象的时候，传入网络和天气服务，初始化引用
    }
};

```

19. 运行程序的时候，自动调用这些逻辑，可以快速完成天气预报这个任务。

```

//发送微信消息服务
class WeChartServiceCommand : public Command
{
private:
    Network &network;           ← 20. 发送微信服务和获取天气服务一样，需要加入网络功能
    WeChatService &wechat;      ← 21. 另外天气服务功能变成微信服务功能
public:
    WeChartServiceCommand(Network &network , WeChatService &wechat) :
        network(network), wechat(wechat)           ← 22. 同样的初始化
    {
    }
    ~WeChartServiceCommand(){}
    void execute() override
    {
        network.on();
        wechat.sendMessage();   23. 发送微信消息
    }
};

```

```

int main(void)
{
    Network network; //单独创建网络对象
    WeChatService wechat; //单独创建微信对象
    WeatherService weatherService; //单独创建天气功能对象

    WeChartServiceCommand weChartServiceCommand {
        network,wechat           ← 我们发送微信信息需要用到，网络和微信的具体发送函数，所以加入这两个
    };
    WeatherForecastCommand weatherForecastCommand{
        network,weatherService
    };           ↑
    这其实就每个服务功能，组合其它几个对象实现
    Shortcut shortcut {
        &weChartServiceCommand,
        &weatherForecastCommand           ← 我们获取天气信息也需要网络和天气执行函数
    };
    shortcut.run();           ← 将服务的不同功能加入快捷对象
    return 0;                ← 快捷对象依次去执行不同的服务
}

```

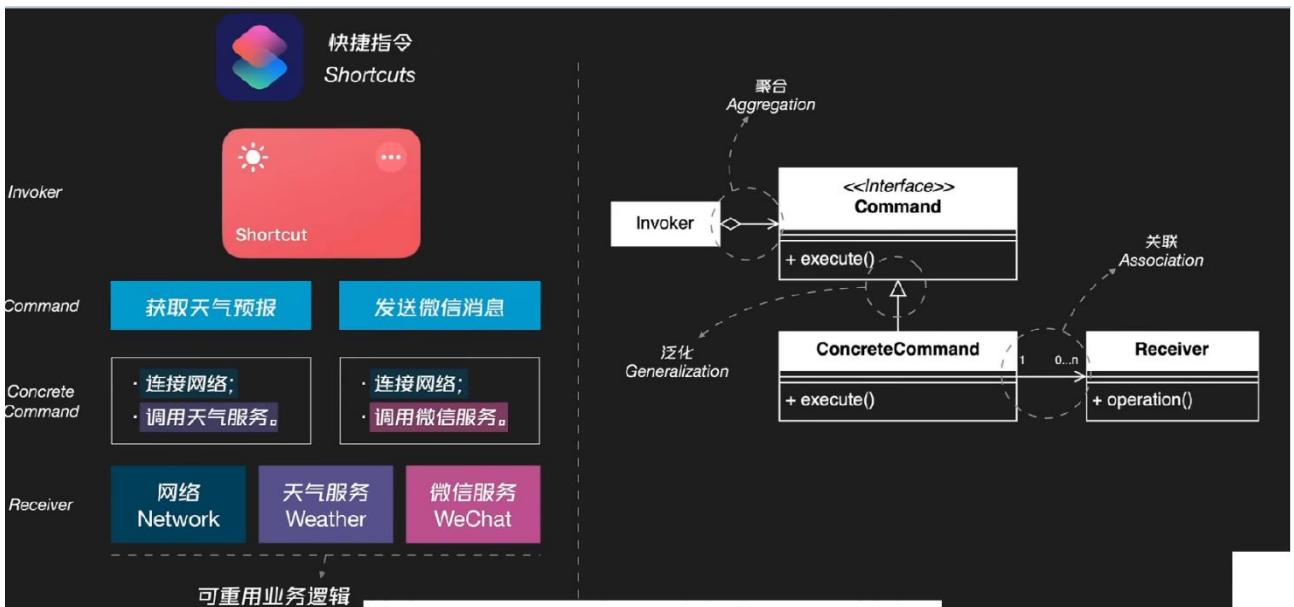
网络链接
微信发送信息
获取天气信息
执行结果

```

class Command
{
public: 之前忘记加public了
    virtual void execute() = 0;
};

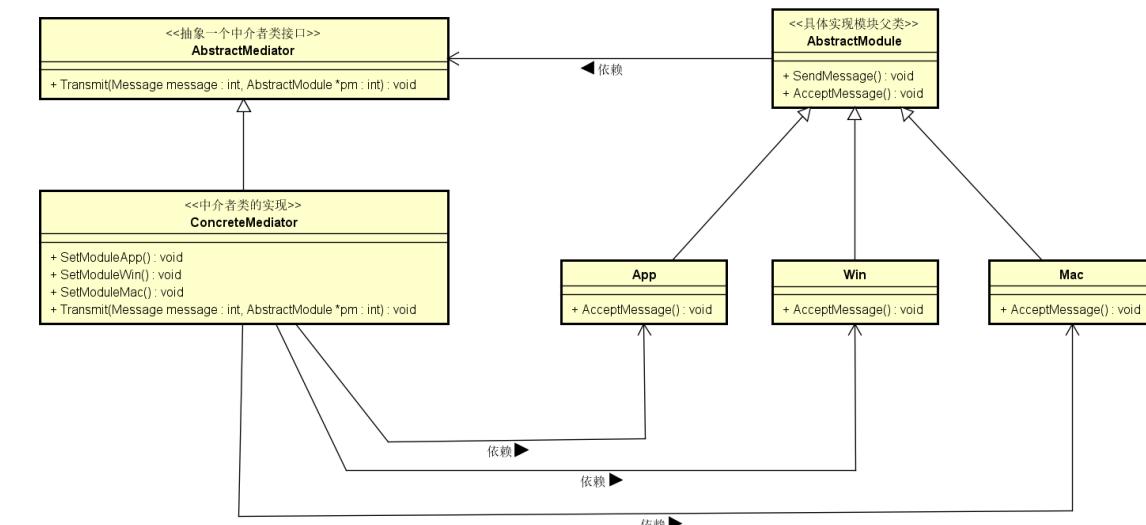
```

下面给出整个命令模式 UML 图



中介者模式

我们实现一个从手机APP到Mac电脑或者到Windows电脑之间相互转发数据的过程



```
enum class Message
{
    ATW_MESSAGE, //app --> win
    ATM_MESSAGE, //app --> mac      1. 定义转发的事件变量
    WTM_MESSAGE, //win --> mac
};

class AbstractModule; //模块父类提前定义
```

```
//中介者转发接口抽象父类
class AbstractMediator 2. 抽象中介者类
{
private:
    /* data */
public:
    AbstractMediator(){}
    ~AbstractMediator(){}
    virtual void Transmit(Message message , AbstractModule *pm) = 0;
};
```

```

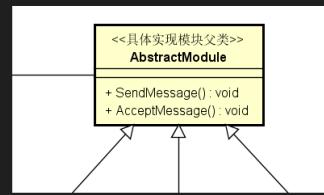
//模块父类
class AbstractModule
{
protected:
    AbstractMediator *pm;
public:
    AbstractModule(AbstractMediator *pm): pm(pm)
    {
    }
    void SendMessage(Message message) //发送消息
    {
        pm->Transmit(message, this);
    }
    virtual void AcceptMessage(Message message) = 0; //需要子类实现的接收过程
};

class App : public AbstractModule
{
    3. 传入中介者对象
public:
    App(AbstractMediator *pm) : AbstractModule(pm){}
    void AcceptMessage(Message message)
    {
        switch(message)
        {
            4. 实现本app类的接收函数
            case Message::ATW_MESSAGE: cout << "app --> win" << endl; break;
            case Message::ATM_MESSAGE: cout << "app --> mac" << endl; break;
            case Message::WTM_MESSAGE: cout << "win --> mac" << endl; break;
        }
    }
};

class Win : public AbstractModule
{
public:
    Win(AbstractMediator *pm) : AbstractModule(pm){}
    void AcceptMessage(Message message) 同理，这不过是windows类
    {
        switch(message)
        {
            case Message::ATW_MESSAGE: cout << "app --> win" << endl; break;
            case Message::WTM_MESSAGE: cout << "win --> mac" << endl; break;
            default:break;
        }
    }
};

class Mac : public AbstractModule mac类，同理
{
public:
    Mac(AbstractMediator *pm) : AbstractModule(pm){}
    void AcceptMessage(Message message)
    {
        switch(message)
        {
            case Message::ATM_MESSAGE: cout << "app --> mac" << endl; break;
            case Message::WTM_MESSAGE: cout << "win --> mac" << endl; break;
        }
    }
};

```

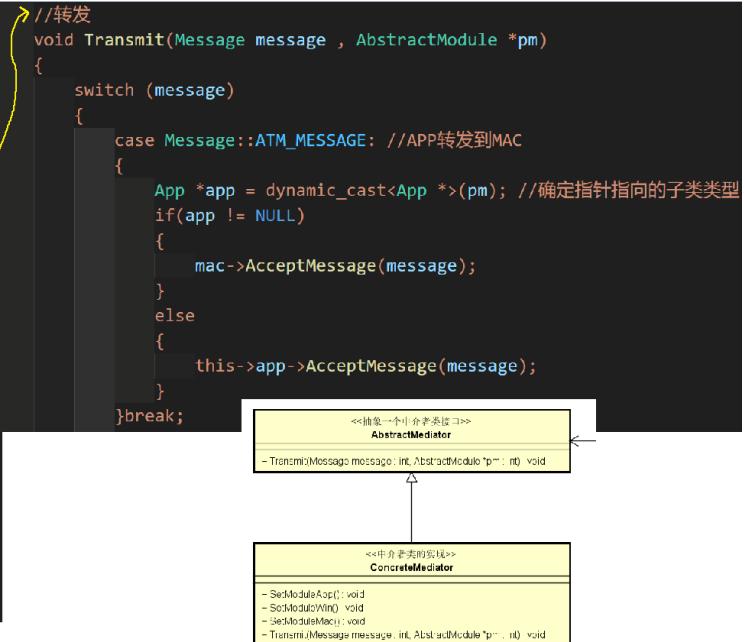


```

//中介者类实现
class ConcreteMediator: public AbstractMediator
{
protected:
    AbstractModule* app = nullptr;
    AbstractModule* win = nullptr;
    AbstractModule* mac = nullptr;

public:          5. 设置当前中介者要转发的模块
    //为中介者设定模块
    void SetModuleApp(AbstractModule *app)
    {
        this->app = app;
    }
    void SetModuleWin(AbstractModule *win)
    {
        this->win = win;
    }
    void SetModuleMac(AbstractModule *mac)
    {
        this->mac = mac;
    }
}

```



```

case Message::ATW_MESSAGE: //APP转发WIN
{
    App *app = dynamic_cast<App *>(pm); //确定指针指向的子类类型
    if(win != NULL)
    {
        win->AcceptMessage(message); //转换成功用windows去做接收消息
    }
    else
    {
        this->app->AcceptMessage(message); //转换不成功用自身去接收消息
    }
} break;
case Message::WTM_MESSAGE: //win转发到mac
{
    Win *win = dynamic_cast<Win *>(pm); //确定指针指向的子类类型
    if(win != NULL)
    {
        mac->AcceptMessage(message);
    }
    else
    {
        this->win->AcceptMessage(message);
    }
} break;

```

```
int main(void)
{
    AbstractMediator *pm = new ConcreteMediator();

    AbstractModule *app = new App(pm); //指定app做中介者
    AbstractModule *win = new Win(pm); //指定win做中介者
    AbstractModule *mac = new Mac(pm); //指定mac做中介者

    //为中介者设定模块
    ConcreteMediator *pc = dynamic_cast<ConcreteMediator *>(pm); //父类到子类转换
    pc->SetModuleApp(app);
    pc->SetModuleWin(win);
    pc->SetModuleMac(mac);

    //不同平台的模块通信
    app->SendMessage(Message::ATM_MESSAGE); //app --> mac的发送
    app->SendMessage(Message::ATW_MESSAGE); //app --> win的发送
    win->SendMessage(Message::ATW_MESSAGE);
    mac->SendMessage(Message::WTM_MESSAGE);
    return 0;
}
```

app --> mac
app --> win
app --> win
win --> mac