

OOP_c++

RyanFcr (在RandomStar前辈的基础上做的笔记)

OOP_c++

0 前言

1 C++的新特性

1.0 输入输出流

1.1 变量和动态内存分配

1.2 引用 Reference

1.3 const类型

2 类 class

2.0 类的基本概念

2.1 构造函数和析构函数

2.2 static类型

2.3 Inline Function内联函数

2.4 继承Inheritance

2.5 友元 friend

2.6 多态和虚函数

2.7 虚函数使用总结

2.8 强制类型转换

2.8.1 static_cast

2.8.2 const_cast

2.8.3 reinterpret_cast

2.8.4 dynamic_cast

2.9 列表初始化

2.10 输入/输出

2.11 左值和右值

2.12 移动构造

3 重载

3.0 函数的重载 overload

3.1 运算符的重载

3.2 输入输出流的重载

4 模板

4.1 namespace 命名空间

4.2 template编程

4.3 STL和迭代器

5 Exceptions 异常处理

6 Smart Pointer 智能指针

7 奇奇怪怪的小知识点

7.1 格式

7.2 sort

7.3 cmath

[7.4 stable_sort](#)

[7.5 typeid](#)

[7.6 构造 析构](#)

[7.7 __gcd](#)

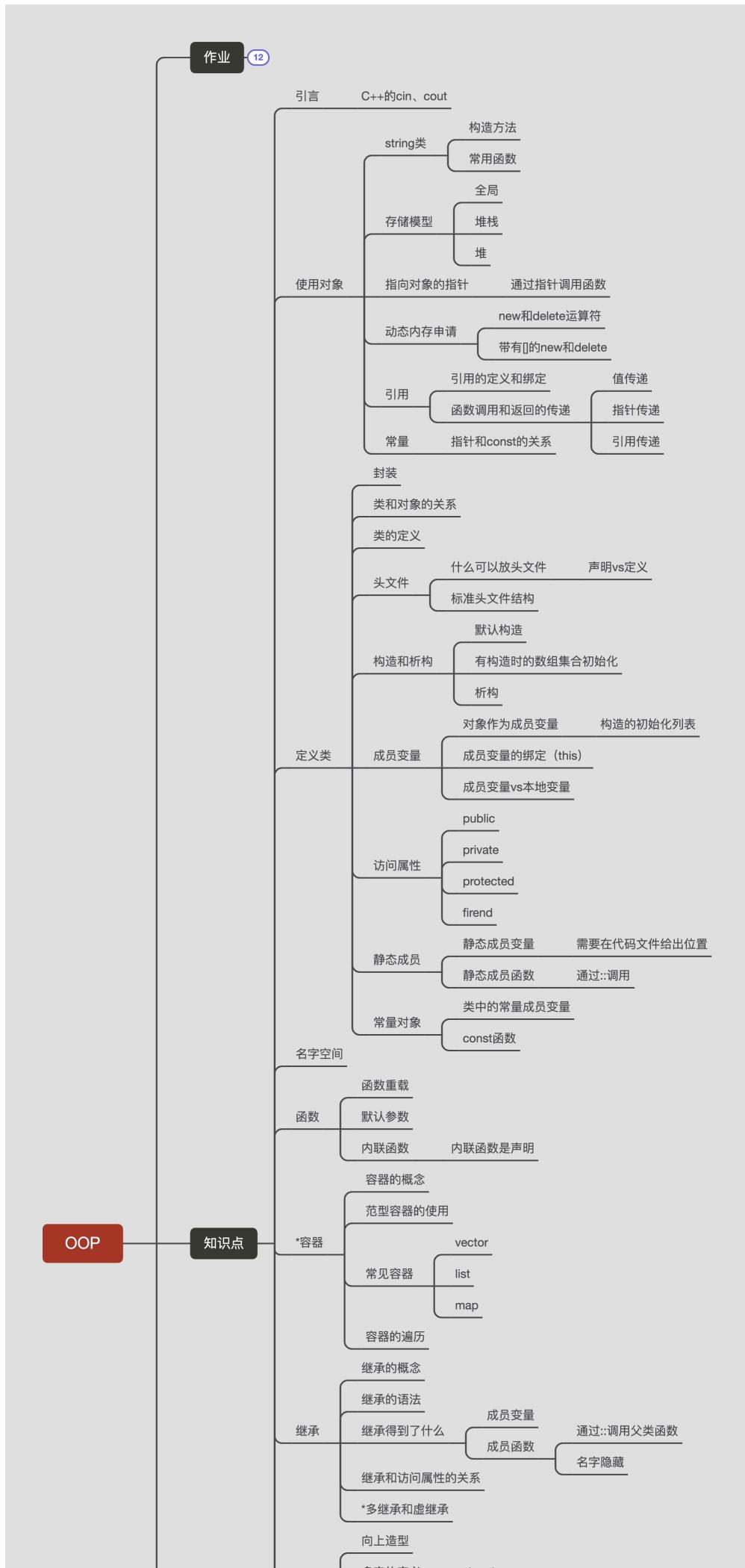
[7 错题集](#)

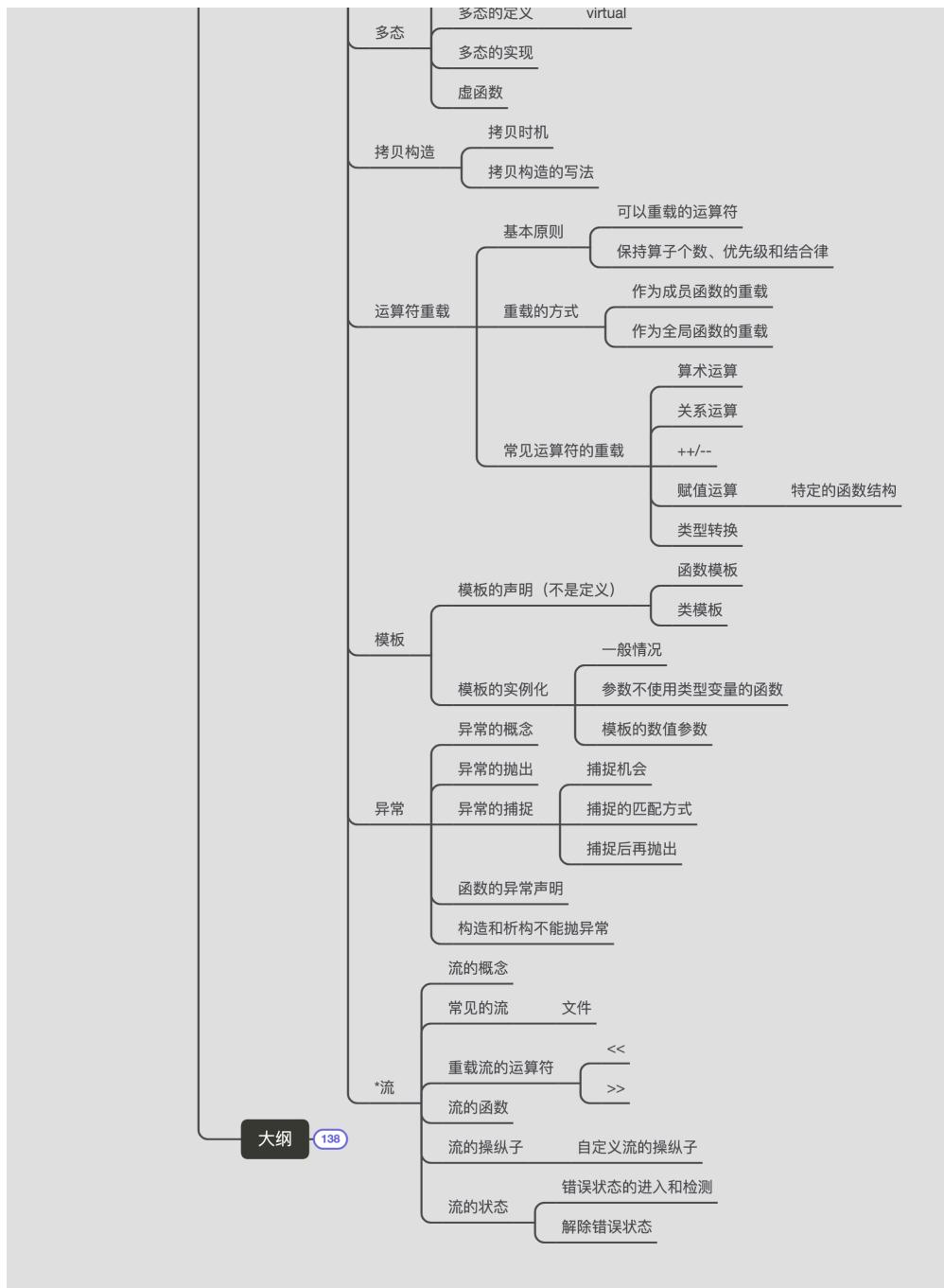
[7.1](#)

[7.2](#)

[7.3](#)

[7.4](#)





0 前言

三大特点

- 封装
- 继承
- 多态

一些Buzzwords

- responsibility-driven design
 - 每部分只做他们自己的事情
- encapsulation 封装
- inheritance 继承

- iterators 迭代
- overriding 覆盖
- cohesion 凝聚
 - 内部的东西越紧越好
- template 模板
- coupling 耦合
 - 耦合度希望越低越好
- interface 接口
 - 把所需成员组合起来，用来装封一定功能的集合
- collection classes 容器 (放东西的东西)
- mutator methods 修改
- polymorphic method calls 多态性

Assessment

- quiz 5%
- Homework 10%
- 7 labs (一次上机期中考试) 15%
- 1 Project 15% team work
 - MUD游戏
 - 日记本/书架库
 - 也提出自己的题目
 - 每一个月都要上交报告
- 期中 5% 90-min week 9
- Final Exam 50%

| 如果想延长ddl，需要已经提交过一个版本，并只能延长24小时

| 语言的能力/使用领域主要是由

- 库
- 传统 (传统操作系统用c等)

| 建议

- 对计算机本身感兴趣 (体系、OS、编译) ——>C
- 想编程解决手头的问题 (统计、AI、桌面小程序) ——>Python
- 有明确的需求 (求职) ——>人家要什么学什么 (PHP、JavaScript、C++)
- 还没想好——>Java

学习建议

- 命令行
 - 编写脚本来完成工作
- git
- 使用第一手资料的意识和能力

Let's C it!

- 缺少type检查
- poor support对于大型编程
- 面向过程的编程

Goal For C++ (C++是C的超集)

- to combine
 - Flexibility and efficiency of C
 - Support for object oriented programming (from SmallTalk)

Alan Kay

C++改良了

- Data abstraction
- Access control
- Initialization & cleanup
- Function overloading
- Streams for I/O
- Constants (C99)
- Name Control
- Inline function (C99)
- References
- Operator overloading
- More safe and powerful memory management
- Support for OOP
- Templates
- Exception handling
- More extensive libraries, STL

1 C++的新特性

1.0 输入输出流

- C++可以使用输入输出流 (cin, cout) 进行输出，比如 `cout<<"Hello world";`
需要包含**头文件** `#include<iostream>`
- 提取符和插入符
- 输入多个变量可以写在一行，如: `cin>>x>>y>>z`

文件名就是iostream，没有.h

```
1 #include<iostream>
2 using namespace std;
3
4 int main()
5 {
6     int age;
7     cin >> age; //输入
8     cout << "Hello,world! I am" << age << "today" << endl;
9     // << 是插入
10    //cout: c是standard
11    // endl换行符
12
13 }
```

编译: gcc——>g++

- String类
 - `#include <string>`
 - 这样就可以定义字符串， string
 - 定义的时候用括号和等号都一样
 - `cin >> str cout<<str`
 - `cin`不读取空白值
 - `cin`不能和`getline`混用， 会出问题

```
■ 1 //Example: 可以用于分割被空格、制表符等符号分割的字符串
2 //`istringstream` 分词，需要`#include<sstream>`、
3
4 #include<iostream>
5 #include<sstream>           //istringstream 必须包含
6 #include<string>
7 using namespace std;
8 int main(){
9     string str="i am a boy";
```

```
10     istream is(str);
11     string s;
12     while(is>>s)  {
13         cout<<s<<endl;
14     }
15 }
```

- getline读到回车

- - 1 string s1,s2;
 - 2 getline(cin,s1);
 - 3 getline(cin,s2);

- cin不读回车，需要用getchar () 将回车读走

- 可以相加，可以赋值
- 不能相减，但是可以转换成字符再减
- `s.length()`
- ctors
 - `string(const char *cp , int len);`
 - `string(const string& s2 , int pos);`
 - `string(const string& s2, int pos ,int len);`
 - 初始化结构
- `substr(int pos,int len);`
- `assign();`
- `insert(const string&,int len);`
- `insert(int pos, const string& s);`
- `erase(int pos , int len);`
- `append();`
 - 在字符串的末尾添加str,
在字符串的末尾添加str的子串,子串以index索引开始，长度为len
在字符串的末尾添加str中的num个字符,
在字符串的末尾添加num个字符ch,
在字符串的末尾添加以迭代器start和end表示的字符序列.

- - 1 str1.append(str2);
 - 2 str3.append(str2, 11, 7);
 - 3 str4.append(5, '.'');

- `replace(int pos , int length , string);`

- `find(string ,int pos);`
 - 返回的是找到的第一个下标
 - 没找到返回-1
- `compare(string)`
 - 返回二者相减
- `to_string(int)`
 - 把数字转换成字符串

- ```

1 #include <string>
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7 int i = 10;
8 std::string s = std::to_string(i);
9 cout<< s << endl; //10
10
11 float f = 1.1;
12 std::string s1 = std::to_string(f);
13 cout<< s1 << endl; //1.100000
14
15 double d = 12.23;
16 std::string s2 = std::to_string(d);
17 cout<< s2 << endl; //12.230000
18
19 return 0;
20 }
```

- `stoi`
  - 可以把字符串转换成几进制的数.
  - `stoi` (字符串, 起始位置, n进制) , 将 n 进制的字符串转化为十进制
- 文件的输入输出, 使用`ofstream`和`ifstream`
- ([35条消息](#)) [fstream和ifstream详细用法ytffhew的博客-CSDN博客fstream和ifstream](#)

- ```

1 //made by Welles Sun
2
3 #include<fstream>
4 using std::ofstream;
5 using std::ifstream;
6 using std::ios;
```

```

7
8 ifstream key_ofs("keys.txt",ios::in);
9 ifstream value_ofs("values.txt",ios::in);
10 ifstream delete_ofs("delete_seq.txt",ios::in);
11
12 int key,value,de;
13 for(int i = 0 ; i < n ; i++)
14 {
15     key_ofs>>key;
16     value_ofs>>value;
17     delete_ofs>>de;
18     keys.push_back(key);
19     values.push_back(value);
20     delete_seq.push_back(de);
21 }
22
23
24 ofstream key_ofs("keys.txt",ios::out);
25 ofstream value_ofs("values.txt",ios::out);
26 ofstream delete_ofs("delete_seq.txt",ios::out);
27
28 for(int i = 0 ; i<n;i++){
29     key_ofs<<keys[i]<<std::endl;
30     value_ofs<<values[i]<<std::endl;
31     delete_ofs<<delete_seq[i]<<std::endl;
32 }
```

```

1 #include<iostream>
2 #include<fstream>
3 #include<iostream>
4 #include<string>
5
6 using namespace std;
7
8 int main()
9 {
10     string str="Hello world!";
11     ofstream fout("out.txt");
12     fout<<str<<endl;
13     ifstream fin("out.txt");
14     string str1,str2;
15     fin>>str1>>str2;
16     return 0;
17 }
```

1.1 变量和动态内存分配

- C++中的变量类型
 - global variable 全局变量，存储在**全局变量区**
 - 可以在不同的cpp文件之间共享，可以使用关键字 `extern` 来使用别的cpp文件中的全局变量
 - static global variable **静态全局变量**，不能在cpp文件之间共享
 - local variable 存储在**栈**区上
 - static local variable 静态局部变量
 - 存储在全局变量区
 - 在初次使用的时候**初始化**，keeps its value between visit to the function
 - allocated variable 动态分配的变量
 - 存储在内存的**堆**结构中
- objects的指针

```
1 string s="hello";
2 string *ps=&s;
3 (*ps).length();
4 ps->length();
```

- &:get address
- *:get the object
- ->:call the function
 - `ps->length()`
- 类和原始类型不同
 - 本地变量的情况下
 - `int i`: 没有确定的值
 - `string s`: 有确定的值
 - `string *ps` 是没有确定的值的，没有指定指向谁
- C++动态内存分配
 - 还可以用 `malloc` 和 `free`，但是不能混用
 - `new` 和 `delete` 是运算符，需要考虑优先级，不是函数

- new用于动态分配内存给变量，如 `new int`, `new double[1000]`, `new double [1000]`◦ 是顺便初始化，初始化为0；不初始化为不为0得看编译器
 - 与malloc的区别：**malloc不执行类的构造函数**，而new出新的对象的时候会执行对象的**构造函数** `new Class_Name[x]` 会执行x次Class_Name的构造函数
 - 构造函数是初始化的作用。除了申请空间，还执行**构造函数赋值**
 - 返回申请出来空间的第一个指针
 - 底层为malloc
 - 主要是为指针开辟空间
 - 在 C++ 中，通过 `new` 运算符来实现动态内存分配。`new` 运算符的第一种用法如下：`T *p = new T;` 其中，T 是任意类型名，p 是类型为 `T*` 的指针
 - `new` 运算符还有第二种用法，用来动态分配一个任意大小的数组：
`T *p = new T[N];`
 - 没有空间的时候会抛异常
 - **内存泄漏 Memory Leak**

```

1 | int* p = new int;
2 | *p = 123;
3 | p = new int;

```

- `new`和`delete`必须搭配使用，不然会内存泄露
- 上面这段代码中一开始为指针p分配了一段内存空间并赋值了123，但是第三行代码又为p赋值了一段新的内存空间，原来的内存空间存储了123，但是这一段内存空间已经没有指针指向，因此**不能访问，也不能删除**，造成了内存泄漏
 - delete用于删除动态分配的内存
 - 用法 `delete p; delete[] p;`
 - `delete[] p` 告诉程序需要**free整个数组**
 - 和new类似，`delete`会**执行所删除对象的构造函数**，不能`delete`没有定义过的变量，同一个变量不能`delete`两次；**可以`delete`空指针**
 - 使用`delete`删除对象时要调用**析构函数**，再回收空间
 - 两个指针p1,p2指向同一个数据，如果p1被`delete`了，p2也不能访问原本p1指向的变量的值，因为`delete`删除的是内存里的数据

```
1 int *p1 = new int;
2 int *p2 = new p1;
3 *p2 = 1;
4 delete p1;
5 cout<<*p2<<endl; // error!!!
```

- 指向同一个数据的两个指针实际上只是两个不同的变量名而已，
delete删除的不是变量名而是数据

1.2 引用 Reference

引用的作用：让程序没有那么多的*和&

三种引用方式：

- 直接访问
 - 指针
 - 引用
- a new type in C++，相当于给变量取了一个别名，使用的还是同一块内存，使用方法为 `type &refname = name;` 引用的对象**不能**是表达式

```
1 char c;
2 char* p = &c;
3 char& r = c; //a reference to a character
```

- r只能是c的别名。不能是其他的别名；c可以有另外的别名
- 定义的时候需要绑定；但是声明的时候不需要绑定
- 引用只能和一个**左值**绑定在一起，可以放在赋值号的左边就可以
- 就是大部分情况下**引用都要绑定一个变量**，除了在**函数参数列表**或者**结构成员变量**里
 - 被caller或者constructor调用的时候才绑定
 - 不能用引用去引用引用
 - **没有指向引用的指针**，但是指针的引用是可以的
 - **没有引用的数组**（一大堆引用），但是可以数组的引用
- 引用和指针的区别
 - 不能**定义空引用**，引用必须连接到一块合法的内存
 - 一旦引用被初始化为一个对象**就不能更改**
 - 引用在创建的时候必须要**初始化**
 - 指针可计算，很灵活（是一种问题）；**引用不可计算**

```

1 int *f(int *x)
2 {
3     (*x)++;
4     return x;
5 }
6
7 int &g(int &x)
8 {
9     x++;
10    return x;
11 }
12
13 int x;
14 int &h()
15 {
16     return x;
17 }
18
19 int main()
20 {
21     int a=0;
22     f(&a);
23     g(a);
24     h()=16; //这里全局变量x被赋值为16
25 }
```

1.3 const类型

常变量 Constant

- 用于定义常量类型，如 `const int x=12345;`, const类型的变量在初始化之后就不能改变其值，编译器确保他不被修改，const型变量不能在连接单元外使用
- 只有函数的参数和成员变量才可以不赋初值
 - 函数的参数是**调用**的时候才初始化
 - 成员变量在**构造成员**的时候才初始化
- 只有全局变量，函数的参数，成员变量才不会被优化成const
- Run-time constants 运行时常量
 - 数组的定义时的长度值必须在**编译期**就已知，所以宏定义中的常数可以作为数组长度，而 `int n; int a[n]` 这样的语法就是错误的，不过现在似乎有了编译器优化，在Dev-cpp中这样写也可以通过编译

```

1 const int size=100;
2 int a[size];//OK
3
4 int x;
5 cin>>x;
6 const int size=x;
7 int a[size];//Error!!!

```

- const和指针pointer

- 常量指针：`char * const p = "abc";` **不能赋予这个指针新的地址**，相当于地址是const类型，但是p指向的值可以改变
- `const char* p = "abc";` 这种情况下p指向的是一个const char类型的值，因此**指向的值不能改变**，而指针指向的对象可以改变
 - 也就是说不能通过这个指针改变这个值
- 左右都有，就都不能改

```

1 //第一种情况是q是一个const指针，但是q指向的东西可以变
2 char * const q ="abc";
3 *q='c'; // OK
4 q++; // Error!
5
6 //第二种情况下，(*q)是一个const的值，此时q所指向的值不能变
7 const char *p = "abc";
8 *p='c';//error!
9
10 //区别这三种东西
11 string p1="zyc";
12
13 const string *p=&p1;
14 string const* p=&p1;//1和2等价
15 string *const p=&p1;

```

- 不能将const类型的变量赋值给对应的指针，因为可能会带来const变量的改变，这是const类型的变量不允许的
- 可以把非const类型的值赋给对应的const型变量，函数中可以将参数设置为const类型表明**这些参数在函数中不能被修改原本的值，也可以将返回值类型设置为const表示返回值不能被修改**
- **const修饰成员函数**，在定义成员函数，在成员函数后面加**const限定**，创建**常成员函数**，表示**只能读取成员变量的值，而不能修改成员变量的值。**（注意区分在返回值前面加**const限定的成员函数**，这种成员函数是表示返回值是一个定值，传回去的东西没法修改）

- 除了成员函数外其他不能加const
- 函数原型和函数定义都要有,写两遍
- const也可以用于构造overload函数
 - 因为其实存在隐藏的参数

```

1 void f(A *this)
2 {
3     cout<<"a";
4 }
5 void f(const A*this) const{
6     cout<<"const a";
7 }
```

如果函数不修改成员变量，就加上const

- const类型的成员变量初始化只能用初始化列表 或者c++11的直接赋值

```

○ //常成员函数
1 class Student{
2 public:
3     Student(char *name, int age, float score);
4     void show();
5     //声明常成员函数
6     char *getname() const;
7     int getage() const;
8     float getscore() const;
9 private:
10    char *m_name;
11    int m_age;
12    float m_score;
13 };
14
15
16 Student::Student(char *name, int age, float score):
17     m_name(name), m_age(age), m_score(score){ }
18 void Student::show(){
19     cout<<m_name<<"的年龄是"<<m_age<<", 成绩是"
20     <<m_score<<endl;
21 }
22 //定义常成员函数
23 char * Student::getname() const{
24     return m_name;
25 }
26 int Student::getage() const{
27     return m_age;
28 }
```

```
26 }
27 float Student::getscore() const{
28     return m_score;
29 }
```

- 在c++ 中，经常用对象的**常指针**和**常引用**作为函数参数，这样既能保证数据的安全，使得数据在函数中不能被任意修改，能再点用函数不必传递实参对象的副本，大幅度减少函数调用的空间和时间的开销。

```
1 #include<iostream>
2 using namespace std;
3
4 struct student{
5     int id;
6 };
7
8 void foo(const student *ps)
9 {
10     /*ps could not be changed in the function
11     cout<<ps->id<<endl;
12     cout<<(*ps).id<<endl;
13 }
14
15 void bar(const student &s)
16 {
17     //s could not be changed in the function
18     cout<<s.id<<endl;
19 }
20
21 int main()
22 {
23     student s;
24     s.id=2;
25     foo(&s);
26     return 0;
27 }
28 const char* v()
29 {
30     return "result of function()"; //传回去的东西无法修改
31 }
32 const int* const w()
33 {
34     static int i;
35     return &i;
36 }
```

- `char * s="Hello world!";` 实质上是 `const char *` 类型，不要去修改 `s` 中的内容，这是一种 **未定义的行为**(undefined behavior)，应该写成 `char s[]="Hello world!";`
 - 只是为了支持老代码

```

1 #include<iostream>
2 using namespace std;
3
4 int main()
5 {
6     const char *s1="Hello World";
7     const char *s2="Hello World";
8
9     cout<<(void*)s1<<endl;
10    cout<<(void*)s2<<endl;
11    return 0;
12    //输出的结果是s1和s2的地址，他们的结果是一样的
13 }
```

- 比较重要的当我们传一个结构的时候，我们常传一个指针
 - 安全

2 类 class

面向过程关注顺序，面向对象关注**对象**，每个对象有什么样的**属性**

- 每个对象都有属性
- 属性是改变的
- 对象间有关系

OOP Characteristics

- Everything is an object
- A program is a bunch of objects telling each other **what to do** by sending messages
 - 不是 How to do, what to do 函数
- Each object has its own **memory** made up of other objects
- Every object has a type
- All objects of a particular type can receive the same messages

面向对象的代码可以一直用，可以重复，可以利用已有的劳动；工业化

抽象和模块化

2.0 类的基本概念

Class 和 Struct 大致是一样的

class 和 struct 的区别： class 中的变量和函数默认为 private， struct 中的函数默认为 public

Class 可以保证包裹，让外面的不能访问；默认声明都不可以访问，只有 public 才可以对外开放

类是用户定义的一种数据类型，可以使用这个类型来说明一个或多个变量，即对象。

- C++ 中的对象 = 属性 + 操作

Class defines Object

Object is a Class

objects = attributes (Data) + operations，对于面向对象的编程（从对象出发）而言，任何变量都是一个对象，任何对象都有对应的类型，程序就是一系列对象互相之间传递信息来完成功能

- 通过 operations 把 Data (attribute) 包裹起来
- C++ 的类包含 **成员变量** 和 **成员函数**
- 类里可以区分 private 和 public
- C++ 中 class 的定义，具体实现和调用可以分成 **三个文件**
 - 头文件 header 是具体实现和调用之间的接口，每个类的定义需要用 1 个头文件 .h
 - 函数声明、类的声明和定义、外部变量、**内联函数声明** 需要在头文件里，而函数的 body 在代码文件 .cpp 里
 - 一个 .cpp 文件是一个编译单元，之后再把所有的 .obj 链接起来成为一个 exe 文件
 - 所以才有了头文件，告诉编译器，在别的编译单元里的东西长什么样
 - 只有 C 和 C++ 才有 .h，C 只会做一遍扫描，从上到下，包括行末的分号是为了简化编译器的编译；其实两遍扫描可以解决这个问题

```
1 #ifndef HEADER_FLAG  
2 #define HEADER_FLAG  
3 //Type declaration here.....  
4 #endif //HEADER_FLAG
```

- `::` 操作符（不是运算符）可以用来访问类中的内容
- 在类里声明，在外实现，用 `::` 串起来，告诉编译器外面的实现是属于哪个类的

```

1 <Class Name>::<function name>
2 ::<function name> //什么都不加的意思是调用哪个free的函数，不属于任何人
3
4
5 class Point {
6 public:
7     void init(int x,int y);
8     void move(int dx,int dy);
9     void print() const;
10
11 private:
12     int x;
13     int y;
14 } ;
15
16 void Point::init(int ix, int iy) {
17     x = ix; y = iy;
18 }
19 void Point::move(int dx,int dy) {
20     x+= dx; y+= dy;
21 }
22 void Point::print() const {
23     cout << x << ' ' << y << endl;
24 }
25
26 void S::f() {
27     ::f(); // would be recursive otherwise!如果不加::，就是调用自己了
28     ::a++; // Select the global a
29     a--; // The a at class scope
30 }
```

- this: this 指针是一个**系统预定义的特殊指针**，指向当前正在操作的对象。
 - 成员函数在类里面，就可以**忽略掉第一个变量**；但实际上也是知道的
 - 类的**非静态成员函数**才会有this指针
 - 任何成员函数在调用的时候都得通过对象来调用，需要**拿到对象的地址**，会赋给this
 - ***this**可以拿到那个对象

| 声明和定义：

- 声明的东西是**不存在的**，只有定义的时候才**存在**。结构都是声明。
 - 不需要`::`
 - 声明是向编译器介绍名字--标识符。它告诉编译器“这个函数或变量在某处可找到，它的模样像什么”。
 - 而定义是说：“在这里建立变量”或“在这里建立函数”。它为名字分配**存储空间**。无论定义的是函数还是变量，编译器都要为它们在定义点分配存储空间。
 - 对于变量，编译器确定变量的大小，然后在内存中开辟空间来保存其数据，对于函数，编译器会生成代码，这些代码最终也要占用一定的内存。
- 定义：
 - 需要`::`，告诉编译器这个是属于哪的
- 总之就是：把建立空间的声明成为“定义”，把不需要建立存储空间的成为“声明”。
 - 基本类型变量的声明和定义(初始化)是同时产生的；而对于对象来说，声明和定义是分开的。

2.1 构造函数和析构函数

- constructor 构造函数
 - 本质也是**成员函数**
 - 构造函数的函数名和类的名字**相同**，可以传入一些参数用来**初始化**一个对象，在类的对象被定义的时候会自动调用构造函数
 - 赋值用`()`赋值，也可以用`=`
 - 当没有构造函数的时候，可以采用**结构类型的赋值方法**；但当你有了构造函数的时候，**你就必须用构造函数来赋值**
 - 如果你有多个构造函数，则会根据不同的构造函数构造
 - 利用`goto`可以跳过构造函数，但是会出错
 - `default constructor` 默认构造函数，**不需要参数**也可以使用的构造函数
 - 你可以给一个默认构造函数
 - 你也可以一个默认构造函数都不给，系统会给你一个，但是什么都不会做

若类A的构造函数定义为

A(int aa=1, int bb=0) { a = aa; b = bb; }

则执行： A x(4);后， x.a和x.b的值分别是（）

A. 1,0

B. 1,4

C. 4,0



D. 4,1

构造函数传参的时候是从左到右

默认值需要从右到左

- 成员变量 (Fields)

- 作用域：类的内部

- 生成期：跟着对象

```
1 #include <iostream>
2 using namespace std;
3 class Y
4 {
5     public:
6     float f;
7     int i;
8     Y(int a);
9     //Y()
10};
11 int main()
12 {
13     Y y1[] = {Y(1), Y(2), Y(3)}; //可以
14     Y y2[2] = {Y(1)};           //不可以，没有给y2[1]赋值
15     Y y3[7];                  //不可以，如果有默认构造函数可以
16     Y y4;                     //不可以，如果有默认构造函数可以
17 }
```

- 初始化列表：在函数签名后面，大括号之前直接对类中定义的变量进行赋值

```

1 class Point {
2     private:
3         const float x, y; //也可以在这里赋值，但是会被初始化列表覆盖
4         Point(float xa = 0.0, float ya = 0.0)
5         :
6             y(ya), x(xa) {} //这个顺序会报warning，需要是声明的顺序
7     };
8     //相当于给y赋了ya，x赋了xa
9     //不需要在ctor的body里赋值

```

- 初始化的顺序是**声明的顺序** `const float x, y;`，**不是在初始化列表里list的顺序**，`destroy`是以一个反顺序的
- `const`类型的成员变量初始化**只能用初始化列表**
- 构造函数的执行分为两个阶段：**初始化阶段和函数执行阶段**，会先执行初始化列表里的赋值，在进入函数主体进行对应的操作

Initialization vs. assignment

```

Student::Student(string s):name(s) {}

initialization
before constructor

Student::Student(string s) {name=s;}

assignment
inside constructor

string must have a default constructor

```

正常情况都要用第一种

- Destructor 析构函数
 - 析构函数的函数名是**类名前面加一个~**，析构函数**不需要参数，不可能被重载overload**，在类的生命周期结束的时候**会被编译器自动调用**

```

1 class Y{
2     public:
3         ~Y() {};
4 }

```

- `delete`会**提前引起析构**，用`delete`释放内存时会调用析构函数；如果他没有出现就不会调用析构
- 除此之外，类结束之后以构造的**反顺序**调用析构
- 全局变量在`main`之前构造，在`main`结束之后再析构

- function overloading 函数重载:
 - 函数名相同而参数的个数和类型不同的几个函数构成重载关系，一个类可以有多个不同的构造函数来解决不同情况下的构造
 - default value: 缺省值，可以在函数参数表中直接声明一些参数的值，但是必须要从右往左，先给最右边的赋值，当传入的参数缺省时函数默认将已经声明的值作为参数的值
 - `stash(int size, int initQuantity = 0)` 类似于这样的语句，赋值需要在**声明语句**里做，而不是在**定义语句**里做；函数不应该知道默认值
 - 在实际调用函数的时候，比如 `stash(1)` 的顺序是从左到右，将size赋成1
- constant object 常量对象
 - 需要加**const** 声明，在声明之后就不能改变这个对象内部变量的值
 - 会有一些成员函数不能正常使用
 - 在成员函数参数表后面加**const** 可以成为**const型成员函数**，**const类型的成员函数不能修改成员变量的值**，称为常成员函数
 - 但是如果成员变量中有指针，并不能保证指针指向的内容不被修改
 - **const** 声明写在函数的开头表示函数的**返回值类型**是**const**
 - **const**类型的函数和非**const**类型的函数也可以构成**重载**关系，比如：

```

1  class A {
2  public:
3      void foo() {
4          cout << "A::foo()" << endl;
5      }
6      void foo() const {
7          cout << "A::foo() const;" << endl;
8      }
9  };
10
11 int main()
12 {
13     A a;
14     a.foo(); //访问的是非const类型的foo
15     const A aa;
16     aa.foo(); //访问的是const类型的foo
17     return 0;
18 }
```

- **const**类型的成员函数的使用规则如下：

- non-const成员函数不能调用const类型对象的成员变量，而const类型可以访问
- const类型函数不会改变任何成员变量的值
- 构成重载关系的时候，const类型的对象只能调用const类型的成员函数，不能调用non-const，而非const类型的对象**优先调用non-const的成员函数**，如果没有non-const再调用const类型的
- copy constructor拷贝构造函数

```

1 #include <iostream>
2 using namespace std;
3 class A
4 {
5     int i;
6
7 public:
8     A(int ii) : i(ii) { cout << "A()" << endl; }
9     A(const A &r) : i(r.i) { cout << "A(A&)" << endl; }
10    A() { cout << "A()" << endl; }
11    virtual ~A() { cout << "~A()" << endl; }
12    int getValue() const { return i; }
13    void setValue(int i) { this->i = i; }
14 };
15
16 void f(A aa)
17 {
18     cout << &aa << endl;
19     cout << aa.getValue() << endl;
20 }
21 int main()
22 {
23     A a;
24     a.setValue(10);
25     cout << &a << endl;
26     f(a);
27 }
28 A()
29 0x7ffe2b4f4b00
30 A(A&)
31 0x7ffe2b4f4b10
32 10
33 ~A()
34 ~A()
```

- 我们发现地址不一样，进入函数内部重新生成了一个本地变量，但是构造函数只调用了一次，析构函数调用了两次；我们发现 `A(const A&r):i(r.i){cout<<"A(A&)"<<endl;}` 这个构造函数被调用了，在拷贝的时候被调用的构造函数叫做**拷贝构造**
- 把一个对象直接赋值给另一个对象，一般的形式为 `class_name(const class_name & copy_class_var)`，通过拷贝构造函数可以实现对象之间的互相赋值
 - 理论上 `const` 也可以不加
- 如果定义变量时直接给变量用同类型的变量赋值，调用的就是拷贝构造函数，如果是**定义之后再赋值，就是调用了重载之后的等号**，比如下面这一段代码

```

1 | class A {
2 |     A() {}
3 |     A(const A& a) {}
4 |     A& operator=(const A& a) {}
5 | };
6 |
7 | int main()
8 | {
9 |     A a;
10 |    A b = a; //调用拷贝构造函数
11 |    A c;
12 |    c = a; //调用重载之后的等号，如果没有重载的话，就是赋值
13 | }

```

- 自己写拷贝构造函数的原因
 - 存在指针，比较复杂
 - 不想全盘拷贝

如果我自己不写拷贝构造函数，其实默认的拷贝构造函数是能work的很好的，但是如果存在上述情况，需要自己写。

○ 什么时候拷贝构造会发生

- 构造的时候
 - 主动构造
 - 默认构造（隐式）
 - 函数传参
 - 函数返回结果
 - 会有优化，比较复杂；优化了就不会有拷贝构造和析构

■ 放进stl容器的时候

用一个对象去初始化同类的另一个对象时，要调用拷贝构造函数，而对象间赋值不会调用拷贝构造函数，因为不是构造

- 如果你自己写了拷贝构造，那么**所有元素都要拷贝**，不然如果有一个没有写拷贝构造函数，就会变成空
 - 任何对象只能被创建和删除一次，**一旦创建，就会成为任何赋值操作的目标**
- C++中的拷贝：**浅拷贝和深拷贝**
 - 浅拷贝：在原来已有的内存中增加一个新的指针指向这一段内存
 - 比如 `string s1="zyc"; string s2(s1);` 就是一种浅拷贝
 - 深拷贝：分配一块新的内存，复制对应的值，并定义一个新的指针指向这一块内存
 - 缺省的拷贝构造函数和赋值运算符进行的都是浅拷贝
 - 拷贝构造函数和赋值运算符的区别
 - 拷贝构造函数是在对象被**创建**的时候调用的
 - 赋值运算符只能使用于已经存在的对象，也就是进行赋值之前，这个对象已经被某个构造函数**构造**出来了
 - 例如

```
1 #include<cstring>
2 #include<iostream>
3 using namespace std;
4
5 struct Person{
6     char *name;
7     Person(const char *s){
8         name=new char[strlen(s)+1];
9         strcpy(name,s);
10    }
11    ~Person(){
12        delete[] name;
13    }
14 }
15
16 //不需要拷贝构造函数的一种方式
17 Person bar(const char *s)
18 {
19     cout<<"in bar()"<<endl;
20     return Person(s);
21 }
```

```

22
23 int main()
24 {
25     Person p1("Trump");
26     Person p2=p1;
27     cout<<(void *)p1.name<<endl;
28     cout<<(void *)p2.name<<endl; //会发现输出的地址是一样的，说
29     明指针在默认情况下也进行了copy，若要避免则应该自己编写拷贝构造函数
30
31 }
```

2.2 static类型

- basic meaning
 - c中静态本地变量：**持久存储**；静态全局变量：**访问受限**
- 类中的成员(变量和函数)分为两种
 - 静态成员：**在类内所有对象之间共享**
 - 实例成员：只能在某个具体的对象中调用
 - **静态函数不能访问实例成员**；访问类的非静态成员，必须通过对对象名，比如实例化一个对象，然后进行访问；对于静态成员不需要通过对对象名也可以，可以直接类名`::`
- static类型的全局变量只在**当前文件**有效，不能通过extern跨文件调用，而函数中的static类型的变量在**第一次调用**的时候会被初始化，之后再调用该函数这个static类型的变量**保持上一次函数调用结束时的值**
- 全局变量的构造在main之前

```

1 #include<iostream>
2 using namespace std;
3
4 class A{
5     public:
6         A() {cout<<"A::A()"<<endl;}
7         ~A() {cout<<"A::~A()"<<endl;}
8     };
9
10 void f(int n)
11 {
12     if(n>10)
13         static A a;
14     cout<<"f()"<<endl;
15 }
16 int main()
```

```

17 {
18     cout<<"start"<<endl;
19     f(1);
20     f(11);
21     f(12);
22     return 0;
23 }
24 //此时只有在第二次调用时a才会被构造
25 //无论什么情况，A的构造和析构函数只会被执行一次
26 输出：
27 start
28 f()
29 A::A()
30 f()
31 f()
32 A::~A()

```

- 类中的static

- 类中定义的static类型的变量是**静态成员变量**（实际上是全局变量（生命来说），但是**访问限制与类的内部**），其值会在这个类的**所有成员之间共享**
 - 要在对应的 .cpp 里面定义

■ 1 | int StatMem::m_h;

- non-const**类型的静态成员变量需要在类的外面进行定义赋值，比如：

```

1 #include <iostream>
2 using namespace std;
3
4 class A {
5 public:
6     static int count;
7     A() {
8         A::count++;
9     }
10 };
11
12 int A::count = 0; // 在类的外部赋值的时候不需要说明static，但是需要注明A::，否则就是一个新的变量
13
14 int main()
15 {
16     A* array = new A[100];
17     cout<<A::count<<endl;

```

```
18 }
19 输出100
```

- 静态成员函数:

- 可以访问类定义中的静态成员变量，但是不能直接访问普通的成员变量
- 可以直接类名::函数名或者对象名::函数名调用
- 需要在函数定义之前加static关键字
- 没有this指针

```
1 /* 请在这里填写答案 */
2 #include <iostream>
3 #include <cmath>
4 using namespace std;
5 class CPoint
6 {
7 private:
8     int x, y;
9
10 public:
11     CPoint(int x, int y)
12     {
13         this->x = x;
14         this->y = y;
15     }
16     CPoint(CPoint &a) : x(a.x), y(a.y) {}
17     int getX()
18     {
19         return x;
20     }
21     int getY()
22     {
23         return y;
24     }
25     void print()
26     {
27         cout<<x<<" , " <<y<<endl;
28     }
29 };
30 class CLine
31 {
32 public:
33     CPoint x, y;
34     static int count;
```

```

35     CLine(CPoint _x, CPoint _y) : x(_x), y(_y) {
36         CLine::count++;
37         CLLine(a) : x(a.x), y(a.y) { CLine::count++; }
38         int GetLen()
39         {
40             int l;
41             l = sqrt((x.getX() - y.getX()) * (x.getX() - y.getX())
42             + (x.getY() - y.getY()) * (x.getY() - y.getY()));
43             return l;
44         }
45         static int ShowCount()
46         {
47             return CLine::count;
48         }
49     };
50     int CLine::count = 0;
51     int main()
52     {
53         int x, y;
54         cin >> x >> y;
55         CPoint p1(x, y);
56         cin >> x >> y;
57         CPoint p2(x, y);
58         CLine line1(p1, p2);
59         cout << "the length of line1 is:" << line1.GetLen() <<
60         endl;
61         CLine line2(line1);
62         cout << "the length of line2 is:" << line2.GetLen() <<
63         endl;
64         cout << "the count of CLine is:" << CLine::ShowCount() <<
65         endl;
66         return 0;
67     }

```

2.3 Inline Function内联函数

执行function call需要额外开销

在编译的时候插入代码的地址

inline是由编译器决定的，编译器可能帮你优化

```

1 //内联函数，交换两个数的值
2 inline void swap(int *a, int *b){
3     int temp;
4     temp = *a;
5     *a = *b;
6     *b = temp;
7 }
```

- 需要在函数名、static和返回类型前面加关键字 `inline`
 - 内联函数在编译期会被编译器在调用处直接扩展为一个完整的函数，因此可以减少运行时调用函数的cost
 - 内联函数的**定义**和**函数主体**部分都应该写在**头文件**中
 - `inline`函数的声明是没有意义的
 - **函数的定义其实就是声明**
 - 如果放在 `.cpp` 里，就默认这个函数只能在这个**本地**使用
 - 本质是**空间换时间** tradeoff
 - 比c的宏要好，编译的时候会检查类型
 - 生成的文件会很大，但是节省了调用的时间
 - class中的函数都是**默认**`inline`的
 - 只在class里写的都是`inline`的，如果在class外利用 `class::f()` 写的话就不是`inline`的
 - Access函数，取class里的一个值的这类函数适合做成`inline`
 - 小的，重复调用的函数适合`inline`
 - 大的，递归的不适合`inline`
- an inline function is expanded in place, like a preprocessormacro, so the overhead of the function call is eliminated, 不需要函数调用产生的开销，由编译器**直接优化**，但是可能会使得需要编译的代码量增大(虽然写的人是看不出来的)，主要作用是减小函数调用时的开销，一般在函数比较小的时候才会使用

2.4 继承Inheritance

可扩展性

- composition 组合：把其他的类作为自己的成员变量
- Inheritance：从基类 (**superclass**) 中继承生成派生类 (**subclass**)
 - 派生类**继承了基类的所有变量和成员函数**
 - 在派生类的里面先有基类的变量和成员函数
 - 指向派生类的指针可以看作指向基类的指针
 - 共享设计

- 派生类中不能直接访问基类的private的变量和成员函数，但是可以通过基类的成员函数来访问这些成员函数和变量
 - 尽管不能访问基类的变量，但是仍然基类的变量是存在于派生类当中的；只能通过基类的成员函数访问
- 派生类的构造函数
 - 可以在派生类的构造函数中调用基类的构造函数
 - 派生类在被构造的时候会先调用基类的构造函数，再调用成员对象的构造函数，再调用派生类的构造函数，析构的时候先调用派生类的析构函数，再调用基类的析构函数
 - 构造顺序：静态成员--虚基类---（抽象类、基类）--成员变量--类自身的构造函数
 - 如果派生类没有定义构造函数，则直接调用基类的构造函数
 - 如果派生类定义了构造函数，在执行之前会先调用基类的构造函数，如果派生类的构造函数中没有显式调用基类的构造函数，则会选择调用基类的无参构造函数
 - 一定会调用父类的构造和析构；符合继承的设计；如果我们想只调用子类，不调用父类，其实是不满足设计理念的，只需要让父类啥都不干就好；如果我们想只调用父类，不调用子类，就子类里不要干事情，都交给父类
 - 如果父类的构造函数需要参数，我们需要在子类的初始化列表里面把参数给父类

1. 公有继承(public)

公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，它们都保持原有的状态，而基类的私有成员仍然是私有的，不能被这个派生类的子类所访问。

2. 私有继承(private)

私有继承的特点是基类的公有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问。

3. 保护继承(protected)

保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成员函数或友元访问，基类的私有成员仍然是私有的。

```
1 class A {  
2 public:  
3     int i;
```

```
4     A(int ii = 0): i(ii) {
5         cout<<"A(): "<<i<<endl;
6     }
7 }
8 //public可以写做private或者protected, 没有写默认为
9 //private, 但是不是oop里正常的继承, private悄悄的继承,
10 //protected你的子类知道的继承
11 class B: public A {
12 public:
13     int i;//只是B的i, 实际上B有两个i, 但是B只能通过A的函数
14     //来拿到A的i
15     A a;
16     B(int ii = 0): i(ii) {
17         cout<<"B(): "<<i<<endl;
18     }
19 };
20
21 int main()
22 {
23     B b(100);
24     return 0;
25 }
26 //输出
27 //A(): 0
28 //A(): 0
29 //B(): 100
```

```
1 #include<iostream>
2 #include<string>
3 using namespace std;
4 class Employee{
5 public:
6     Employee(const string& _name,const string&
7 _ssn):
8         name(_name),ssn(_ssn){}
9     const string& getName() const {return name;}
10    const string& getSSN() const {return ssn;}
11 //const string& 传递开销最小，同时不会改变原值
12    void print()const
13    {
14        cout<<name<<endl;
15        cout<<ssn<<endl;
16    }
17    void print(const string& msg)const {
18        cout<<msg<<endl;
```

```

18     print();
19 }
20 protected:
21     string name;
22     string ssn;
23     //为了避免引用风暴，可以用std::string
24 };
25 class Manager:public Employee{
26 public:
27     Manager(const string& _name,const string&
28     _ssn,const string& _title):
29         //name(_name),ssn(_ssn),title(_title){}
30         Employee(_name,_ssn),title(_title){}
31     //直接利用父类的构造函数
32     const string& getTitle()const { return title; }
33     void print()const{
34         Employee::print();//如果想调用父类的print，只能
35         通过这样的方式，因为已经把父类的print隐藏了
36         cout<<title<<endl;
37     }
38 protected:
39     string title;
40 };
41
42 int main()
43 {
44     Employee p1("AA","123");
45     Manager p2("Lu","3233","Miss");
46     p1.print();
47     p2.print();
48     p1.print("welcome:");
49     // p2.print("welcome:");
50     //会出错，因为一个被函数重载了，就都被重载了name hide,
51     子类只有一个函数，父类其他同名的也会被隐藏
52 }
```

```

1 //构造函数的继承
2 #include <iostream>
3 #include <string>
4 using namespace std;
5 class Student
6 {
7 public:
8     Student(int n, string nam)
9     {
```

```
10     num = n;
11     name = nam;
12 }
13 void display()
14 {
15     cout << "num:" << num << endl;
16     cout << "name:" << name << endl;
17 }
18
19 protected:
20     int num;
21     string name;
22 };
23
24 class Student1 : public Student
25 {
26 public:
27     student1(int n, string nam, int a) : Student(n,
nam)
28     {
29         age = a;
30     }
31     void show()
32     {
33         display();
34         cout << "age: " << age << endl;
35     }
36
37 private:
38     int age;
39 };
40
41 class Student2 : public Student1
42 {
43 public:
44     student2(int n, string nam, int s, int a) :
Student1(n, nam, a)
45     {
46         score = s;
47     }
48     void show_all()
49     {
50         show();
51         cout << "score:" << score << endl;
52     }
```

```

53
54     private:
55         int score;
56     };
57
58     int main()
59     {
60         student2 stud(10010, "Li", 17, 89);
61         stud.show_all();
62     return 0;
63 }
```

- 当继承和组合两种情况同时出现时，先构造基类，再构造派生类中**组合的其他类**，再构造派生类，析构的时候类似
 - **Base class is always constructed first**
 - 就算组合的类在**初始化列表或者构造函数中没有调用**构造函数，C++编译器也会自动调用这个类默认的构造函数
- 访问控制
 - **public**: 所有情况下可见
 - **protected**: 可以被**自己/派生类**和友元函数访问
 - 比如在main里面，生成一个父类/子类，是没有办法访问**protected**的；但自己可以在自己的class里面访问自己
 - **private**: 对于自己和友元函数可见，派生类的只能通过父类的函数去访问
- 继承的种类：public, private, protected继承，继承之后的基类变量的访问控制取原本类型和继承类型中较严格的
- 可以通过**强制类型转换**把派生类的对象转化为基类的对象
- 可以有多个父类，多继承

2.5 友元 friend

- 友元函数
 - 在类中**声明**一个全局函数或者其他类的成员函数为 **friend**
 - 可以使这些函数拥有访问类内**private**和**protected**类型的变量和函数的权限
 - 友元函数也可以是一个类，这种情况下被称为是友元类，整个类和所有的成员都是友元
 - **友元函数本身不是那个类的成员函数**，函数签名里不需要 `className::` 来表示是这个类的成员函数，直接作为普通函数即可

```

1 class A {
2     private:
3         int val;
4     public:
5         A(int value): val(value) {
6             cout<<"A()"<<endl;
7         }
8         friend void showValue(A a);
9     };
10
11 void showValue(A a)
12 {
13     cout<<a.val<<endl;
14 }
```

- 类与类之间的友元关系**不可以继承**

2.6 多态和虚函数

- 多态 Polymorphism

Poly: 多

morph: 形态

ism: 性质

- Up-casting 造型|向上造型
 - 改变“眼光”
 - 将派生类的成员看作是基类的成员
 - 指针和引用
 - 不同于cast (强制类型转换)，不改变内容
- 同一段代码可以产生不同效果
- 对于继承体系中的某一系列**同名函数**，不同的类型会**调用不用的函数**
- 一般情况下，有继承关系的类之间有函数构成**重载关系**，依然会根据**变量类型**来调用对应的函数，比如：
 - virtual: 虚，说明子类可能覆盖——**override** (只需父类注明virtual)

```

1 #include <iostream>
2 using namespace std;
3
4 class A{
5     public:
6         virtual void foo() {
```

```

7         cout<<1<<endl;
8     }
9 };
10
11 class B: public A{
12 public:
13     virtual void foo() {
14         cout<<2<<endl;
15     }
16 };
17
18 int main()
19 {
20     A a;
21     B b;
22     a.foo();
23     b.foo();
24     return 0;
25 }
26 输出
27     1
28     2

```

- 此时运行的结果是1和2，即A型的变量的foo函数是基类中的，B类型的变量的foo函数是派生类中的

```

1 #include <iostream>
2 using namespace std;
3 class A
4 {
5 public:
6     int i;
7     A()
8     {
9         i = 10;
10        cout << "A()" << i << endl;
11    }
12    void f() { cout << "A::f()" << endl; }
13 };
14 class B : public A
15 {
16 public:
17     int i;
18     B()
19     {

```

```

20         i = 20;
21         cout << "B()" << i << endl;
22     }
23     void f() { cout << "B::f()" << endl; }
24 };
25 int main()
26 {
27     B b;
28     A *p = &b; // up-cast
29     b.f();      //函数和变量都同名，用b的时候调用b的，但是用指针
                  的时候，是调用父类的，加'virtual'就是让指针知道，调用的是子类的
30     p->f();
31     // A::f()
32     cout << sizeof(*p) << endl;
33     //输出4, sizeof编译时刻计算，编译的时候，*p类型是A，静态类型
34     cout << sizeof(b) << endl;
35     //输出8
36     int *pi = (int *)p;
37     cout << pi[0] << "," << pi[1] << endl;
38     // 10,20
39     cout << p->i << endl;
40     // 10
41     cout << b.i << endl;
42     // 20
43 }

```

- 静态绑定

- 函数的调用在**程序开始运行之前**就已经确定了
 - 编译的时候就绑定了，用`.`或者`->`
 - 默认静态
- 对于像下面这样的情况，基类的指针(引用)指向派生类，并且调用了基类中也存在的同名函数，**最终调用的都是基类的同名函数**。所以需要使用多态，虚函数

```

1 class Shape {
2     protected:
3         int width, height;
4     public:
5         Shape( int a=0, int b=0)
6     {
7         width = a;
8         height = b;
9     }

```

```

10     int area()
11     {
12         cout << "Parent class area :" << endl;
13         return 0;
14     }
15 };
16 class Rectangle: public Shape{
17     public:
18         Rectangle( int a=0, int b=0):Shape(a, b) { }
19         int area ()
20         {
21             cout << "Rectangle class area :" << endl;
22             return (width * height);
23         }
24 };
25 class Triangle: public Shape{
26     public:
27         Triangle( int a=0, int b=0):shape(a, b) { }
28         int area ()
29         {
30             cout << "Triangle class area :" << endl;
31             return (width * height / 2);
32         }
33 };
34
35 int main( )
36 {
37     Shape *shape;
38     Rectangle rec(10,7);
39     Triangle tri(10,5);
40     shape = &rec;
41     shape->area();
42     shape = &tri;
43     shape->area();
44     return 0;
45 }
```

- 虚函数 Virtual Function

- 一种用于**实现多态**的机制，核心理念是通过**基类访问派生类定义**的函数
 - 这种方式称为**动态绑定** (binding)
 - 动态绑定调用函数操作是通过指向对象的**指针或对象引用**来实现的
 - 那么，什么时候会执行函数的动态绑定？这需要符合以下三个条件。
 - 通过指针来调用函数

- 指针upcast向上转型（继承类向基类的转换称为upcast，关于什么是upcast，可以参考本文的参考资料）
- 调用的是虚函数

对象本身访问的虚函数仍然是静态绑定

Q1：在构造函数、析构函数中调用虚函数不是动态绑定 √

- 子类在构造父类的时候，自己的虚函数还没写到Vptr里面

Q2：在虚函数中不能使用this指针 ×

- 虚函数的this指的是vptr

```

1  #include <iostream>
2  struct A {
3      virtual void foo(int a = 1) {
4          std::cout << "A" << "\n" << a;
5      }
6  };
7  struct B : A {
8      virtual void foo(int a = 2) {
9          std::cout << "B" << "\n" << a;
10     }
11 };
12 int main () {
13     A *a = new B;
14     a->foo();
15 }
16
17

```

这个默认参数是编译时刻决定的，是静态绑定，和动态绑定无关，输出1与指针的类型有关

- 虚函数的继承一定是**公有继承**
- 虚函数的**类外声明**不需要再加**virtual**关键字
- 用于区分派生类中和基类同名的方法函数，需要将**基类**的成员函数类型声明为**virtual**
 - 基类中的**析构函数一定要为虚函数**，否则会出现对象释放错误
 - 基类的一定要写，派生类的可以不写
- 只要你有一个类，这个类的析构函数就要加上**virtual**，因为我们不能预判未来他会被什么继承；所以所有的类都应该要有一个**virtual table**
 - 作用是**delete**动态对象时释放资源
- 子类的函数返回类型可以是自己类的指针或者引用，也属于返回类型相同
 - 但是只能限定为**指针或引用**

Relaxation example

```
class Expr {  
public:  
    virtual Expr* newExpr();  
    virtual Expr& clone();  
    virtual Expr self();  
};  
  
class BinaryExpr : public Expr {  
public:  
    virtual BinaryExpr* newExpr(); // Ok  
    virtual BinaryExpr& clone(); // Ok  
    virtual BinaryExpr self(); // Error!  
};
```

- 返回对象本身会有**slide-off**, 舍去一些子类有的属性
 - override语法：派生类中可以用**override**关键字来声明，表示对基类虚函数的覆盖

overload

成员函数被**重载**的特征

- (1) 相同的范围（在同一个类中）；
- (2) 函数名字相同；
- (3) 参数不同；
- (4) **virtual** 关键字可有可无。

override

“**覆盖**”是指派生类函数覆盖基类函数，特征是：

- (1) 不同的范围（分别位于派生类与基类）；
- (2) 函数名字相同；
- (3) 参数相同；
- (4) 基类函数必须有**virtual** 关键字，子类不必要，可以加(好习惯，因为说不定有孙子，不过不加也是**virtual**了)
- (5) 返回类型相同，可以是比如子类返回子类的指针，父类返回父类的指针

namehide: 如果父类里面有**overload**，但是在子类里面只**override**了其中一个，那么父类里的其他**overload**函数会被隐藏

- Tip:
 - 不要在子类重复定义一个**非virtual**的和父类同名的函数
 - 不要在子类重复定义一个在父类的继承函数的**默认参数**；不要轻易写默认参数
- 虚函数需要借助->**达到多态的效果**

- 如果基类指针/引用**指向基类**, 那就正常调用基类的相关成员函数
- 如果基类指针**指向派生类**, 则调用的时候会调用派生类的成员函数
- 派生类指针**不能**指向基类

```

1 class A{
2 public:
3     virtual void foo(){
4         cout<<"A"<<endl;
5     }
6 };
7
8 class B{
9 public:
10    virtual void foo(){
11        cout<<"B"<<endl;
12    }
13 }
14
15 int main()
16 {
17     A *a=new B();
18     a->foo(); //结果为B
19     return 0;
20 }
```

◦ 多态变量

- 有两个类型
 - 静态: shape*
 - 动态: 运行的时候才知道

```

1 void render (shape* p){
2     p->render();
3 }
4
5 void func(){
6     Ellipse ell(10.20);
7     ell.render(); //静态
8
9     Circle circ(40);
10    circ.render(); //静态
11
12    render(&ell); //动态
13    render(&circ); //动态
```

- 虚函数的实现方式：**虚函数表** virtual table
 - 每一个有虚函数的类都会有一个**虚函数表**，该类的任何对象中都存放着**虚函数表的指针**，**虚函数表中列出了该类的虚函数地址**
 - 虚函数表是一个指针数组，里面存放了一系列**虚函数的指针**
 - 虚函数的调用需要经过虚函数表的查询，非虚函数的调用不需要经过虚函数表
 - 虚函数表在代码的**编译阶段**就完成了构造，虚表内的条目，即虚函数指针的赋值发生在编译器的**编译阶段**，也就是说在代码的编译阶段，虚表就可以构造出来了。当类的对象在创建时便拥有了这个**指针**，且这个指针的值会自动被设置为**指向类的虚表**。
 - 我们把经过虚表调用虚函数的过程称为**动态绑定**，其表现出来的现象称为运行时多态。动态绑定区别于传统的函数调用，传统的函数调用我们称之为静态绑定，即函数的调用在编译阶段就可以确定下来了。

那么，什么时候会执行函数的动态绑定？这需要符合以下三个条件。

 - 通过指针来调用函数
 - 指针upcast向上转型（继承类向基类的转换称为upcast，关于什么是upcast，可以参考本文的参考资料）
 - 调用的是虚函数

如果一个函数调用符合以上三个条件，编译器就会把该函数调用编译成动态绑定，其函数的调用过程走的是上述通过虚表的机制。

 - 一个类只有一张虚函数表，每一个对象都有指向虚函数表的一个指针 `_vptr`
 - **构造出来，_vptr不改变**，_vptr赋值之后也不改变；赋值会类型转换

```

1 //在构造函数里面调用虚函数f()，会表现的好像没有做动态
2 //绑定
3 //A的f被调用
4 //因为_vptr是在构造的时候写进去的，所以_vptr是指向自己
5 //这个类的函数;
6 #include<iostream>
7 using namespace std;
8 class A{
9 public:
10 A(){f();}

```

```

9   virtual void f(){cout<<"A::f()"<<endl;}
10 }
11 class B:public A{
12 public:
13   B(){f();}
14   void f(){cout<<"B::f()"<<endl;}
15 }
16 int main()
17 {
18   A a;
19   a.f();
20   B b;
21   b.f();
22 }
23 输出:
24 A::f()
25 A::f()
26 A::f()
27 B::f()
28 B::f()

```

- 多态的函数调用语句被编译成一系列根据**基类指针所指向的对象存放的虚函数表的地址**，在从虚函数表中查找地址调用对应的虚函数
- 事实上虚函数会给函数加上一个新的参数，是一个指针，**占用8字节(64位)**

构造函数不能为虚函数，虚函数表在构造函数调用后才建立，虚函数的调用需要虚函数表指针，而指针存放在对象的内存空间里，若构造函数声明为虚函数，那么对象还没创建，还没有内存空间，更没有虚函数表地址来调用虚函数

内联函数没有必要是虚函数，当虚函数呈现多态性的时候不可以内联，其他情况随便，其实是可以的

• 接口(C++抽象类)

- 描述了一个类应该有的功能和行为，但是不用在这个类中实现，而是在派生类中实现；他不应该有实例化
- **抽象类不能被用于实例化对象**，它只能作为**接口**使用。如果试图实例化一个抽象类的对象，会导致编译错误。可用于实例化对象的类被称为**具体类**。
 - 如果类中**至少有一个**函数被声明为**纯虚函数**，则这个类就是抽象类。
 - 纯虚函数：`virtual int func() = 0;` 表明该函数没有主体，基类中没有给出有意义的实现方式，需要在派生类中进行扩展

- 只提供一个接口，必须在派生类中实现
 - 不在派生类实现的话，就不能被创造出来
 - 将构造函数说明为纯虚函数是没有意义的
 - 因为构造函数不能是虚函数
1. 抽象类只能用作其他类的基类，**不能定义抽象类的对象。**
 2. 抽象类**不能用于参数类型、函数返回值或显示转换的类型**
 3. 抽象类**可以定义抽象类的指针和引用**，此指针可以指向它的派生类，进而实现多态性。
- 可以使用纯虚函数来实现抽象类的定义，比如：

```

1 class Shape {
2 public:
3     virtual double getArea() = 0;
4     Shape(int a, int b): length(a), width(b) {}
5 protected:
6     int length;
7     int width;
8 };
9
10 class Rectangle: public Shape {
11 public:
12     double getArea() {
13         return length * width;
14     }
15 };
16
17 class Triangle: public Shape {
18 public:
19     double getArea() {
20         return length * width / 2;
21     }
22 };
23 int main(void)
24 {
25     Rectangle Rect;
26     Triangle Tri;
27
28     Rect.setWidth(5);
29     Rect.setHeight(7);
30     // 输出对象的面积
31     cout << "Total Rectangle area: " << Rect.getArea() << endl;
32
33     Tri.setWidth(5);

```

```
34     Tri.setHeight(7);
35     // 输出对象的面积
36     cout << "Total Triangle area: " << Tri.getArea() << endl;
37
38     return 0;
39 }
```

静态成员函数不能是虚函数，所以它们不能实现多态。静态成员函数对于每一个类只有一份代码，所有的对象共享这份代码，**它不归某个对象所有，所以没有动态绑定的必要性，不能被继承**，只属于该类。

多继承：

C++是唯一支持多继承的

为了解决C++没有单根结构

但是会有个问题——Vanilla ML

菱形的继承问题

```
1 class C:public B,public A{
2 };
```

虚拟继承是多重继承中特有的概念。虚拟基类是为解决多重继承而出现的。如：类D继承自类B1、B2，而类B1、B2都继承自类A，因此在类D中两次出现类A中的变量和函数，这时会产生二义性。为了解决二义性，同时为了节约内存，B1、B2对A的继承定义为虚拟继承，而A就成了虚拟基类，这样D中就只有一份A中的变量和函数。

虚继承主要用于菱形形式的继承形式。

虚继承是为了在多继承的时候避免引发歧义，

比如类A有个就是a, B继承了A，C也继承了A，当D多继承B，C时，就会有歧义产生了，所以要使用虚拟继承避免重复拷贝。

```
1 #include <iostream>
2 using namespace std;
3 class CRAFT
4 {
5 protected:
6     double speed;
7
8 public:
9     CRAFT(double s) : speed(s) { cout << "创建航行器(速度：" <<
10         speed << ")" << endl; }
```

```
10 |     virtual ~CRAFT()
11 | {
12 |     cout << "销毁航行器(速度: " << speed << ")" << endl;
13 | }
14 |     virtual void show()
15 | {
16 |     cout << "航行(速度: " << speed << ")" << endl;
17 | }
18 | };
19 | class PLANE : virtual public CRAFT
20 | {
21 | protected:
22 |     double width;
23 |
24 | public:
25 |     PLANE(double s, double w) : CRAFT(s), width(w)
26 |     {
27 |         cout << "创建飞机(翼展: " << width << ")" << endl;
28 |     }
29 |     ~PLANE()
30 |     {
31 |         cout << "销毁飞机(翼展: " << width << ")" << endl;
32 |     };
33 |     void show()
34 |     {
35 |         cout << "航行(速度: " << speed << ","
36 |             << " 翼展: " << width << ")" << endl;
37 |     }
38 | };
39 | class SHIP : virtual public CRAFT
40 | {
41 | protected:
42 |     double depth;
43 |
44 | public:
45 |     SHIP(double s, double d) : CRAFT(s), depth(d)
46 |     {
47 |         cout << "创建船(吃水: " << depth << ")" << endl;
48 |     }
49 |     ~SHIP()
50 |     {
51 |         cout << "销毁船(吃水: " << depth << ")" << endl;
52 |     };
53 |     void show()
54 |     {
```

```

55     cout << "航行(速度: " << speed << ","
56             << " 吃水: " << depth << ")" << endl;
57 }
58 };
59 class SEAPLANE : virtual public PLANE, SHIP
60 {
61 public:
62     SEAPLANE(double s, double w, double d) : PLANE(s, w),
63             SHIP(s, d), CRAFT(s)
64     {
65         cout << "创建水上飞机" << endl;
66     }
67 ~SEAPLANE()
68 {
69     cout << "销毁水上飞机" << endl;
70 }
71 void Show()
72 {
73     cout << "航行(速度: " << speed << ","
74             << " 翼展: " << width
75             << ","
76             << " 吃水: " << depth << ")" << endl;
77 }
78 };
79
80 int main()
81 {
82     CRAFT *p;
83     double s, w, d;
84     cin >> s >> w >> d;
85     p = new SEAPLANE(s, w, d);
86     p->Show();
87     delete p;
88     return 0;
89 }

```

2.7 虚函数使用总结

- 情况1：基类和派生类都不是virtual
 - 此时对于基类的对象和基类的指针，执行的就是基类的f，对于派生类的执行的就是派生类的f

```

1 #include<iostream>
2 using namespace std;

```

```

3 class A {
4 public:
5     void f() {
6         cout << "af" << endl;
7     }
8 };
9
10 class B: public A {
11 public:
12     void f() {
13         cout << "bf" << endl;
14     }
15 };
16
17 int main()
18 {
19     A a;
20     B b;
21     a.f();
22     b.f();
23     A* pb = &b;
24     pb->f();
25 }
26 af
27 bf
28 af

```

- 情况2：派生类中的同名函数是虚函数：无影响，和1一模一样
- 情况3：基类中的是虚函数，派生类中不注明是虚函数：此时派生类的对象和指向派生类的指针执行的都是派生类的函数f，基类的对象和指向基类对象的指针执行的都是基类的函数f
- 总结：
 - virtual的虚函数关键字是**向下负责的**，派生类声明virtual对基类**无任何影响**
 - 对于指针和引用而言
 - 不是虚函数的时候，调用的函数取决于**指针和引用的变量类型**(基类指针调用基类，派生类指针调用派生类)
 - 是虚函数的时候，调用函数取决于指针和引用**指向的变量类型**(指向基类调用基类，指向派生类调用派生类)
 - 以上面情况1的代码为例，如果是虚函数，输出的就是af bf bf
 - 当然如果派生类里**没有新的同名函数**，那么执行的都是基类里的
 - 要注意**派生类指针不能直接指向基类的对象**

- 如果虚函数里还需要调用其他函数，调用的规则也和上面的一样，比如下面有个历年卷上面的神题：
- 历年卷上的一个题目：写出程序的输出

```

1 class B {
2 public:
3     void f() {
4         cout << "bf" << endl;
5     }
6     virtual void vf() {
7         cout << "bvf" << endl;
8     }
9     void ff() {
10        vf();
11        f();
12    }
13     virtual void vff() {
14        vf();
15        f();
16    }
17 };
18
19 class D: public B {
20 public:
21     void f() {
22         cout << "df" << endl;
23     }
24     void ff() {
25         f();
26         vf();
27     }
28     void vf() {
29         cout << "dvf" << endl;
30     }
31 };
32
33 int main()
34 {
35     D d;
36     B* pb = &d;
37     pb->f();
38     pb->ff();
39     pb->vf();
40     pb->vff();

```

- 这道题的分析过程如下：

- 首先调用f，而f不是虚函数，所以根据指针类型调用了B中的f，输出bf
- 再调用ff，因为ff也不是虚函数，所以调用B中的ff，B中的ff调用了vf和f，而vf是虚函数，B类型指针指向的是D，所以调用D中的vf，输出dvf，调用f则和上面一样输出bf
- 再调用vf，由于vf是虚函数，所以要调用D中的vf，输出dvf
- 再调用vff，虽然是虚函数但是D中没有定义同名函数，所以调用B中的vff，vff中调用vf和f，同2一样输出的是dvf和bf
- 所以最终的输出是

```

1 bf
2 dvf
3 bf
4 dvf
5 dvf
6 bf

```

- 这道题涵盖了单继承虚函数的所有情况

- 基类虚函数的优先级高于派生类中的需要强制类型转换的同名函数

- 比如下面一段代码中

```

1 class A {
2 public:
3     virtual void f(int i) {
4         cout << 1 << endl;
5     }
6 };
7
8 class B: public A {
9 public:
10    virtual void f(double i) {
11        cout << 2 << endl;
12    }
13 };
14
15 int main()
16 {
17     A* pa = new B;
18     pa->f(1);

```

```
19     return 0;  
20 }
```

- 这里输出的结果是1，事实上两个f并不构成虚函数的关系，因为f(1)中1是int类型，所以优先调用了对int匹配度高的
- 事实上如果是f(1.1)输出的结果仍然是1，并且CLion会提示参数需要类型转换
- 事实上两个f不构成虚函数的多态关系，所以调用哪个并不看指针指向的对象，而是看指针本身的类型！

2.8 强制类型转换

2.8.1 static_cast

- `static_cast` 用于数据类型的强制转换，有这样几种用法
 - 基本数据类型的转换，比如char转换成int
 - 在类的体系中把基类和派生类的指针和引用进行转换
 - 向上转换是安全的；子类指针或引用转换成基类
 - 向下转换是不安全的
 - 只能在有相互联系的类型中进行相互转换，不一定包含虚函数
 - 把空指针转换成目标类型的空指针
 - 把任何类型转换成void类型
- `static_cast`不能转换掉有`const`、`static`的变量
- `static_cast`不做任何安全性的检查，安全性需要程序员自己保证

2.8.2 const_cast

- `const_cast` 可以强制去掉`const`的常数特性，只能用在指针和引用上面
 - 常量指针被转化成非常量的指针，仍然指向原来的对象
 - 常量引用被转换成为非常量的引用，仍指向原来的对象
- 来看一段代码
 - 打印出来的结果是a=10，而p和q所指向的值是20，a的地址和pq指向的地址是一样的
 - 事实上第五行的赋值是一种未定义行为，最好别用

```
1 const int a = 10;
2 const int *p = &a;
3 int *q;
4 q = const_cast<int *>(p);
5 *q = 20;
6 cout << a << " " << *p << " " << *q << endl;
7 cout << &a << " " << p << " " << q << endl;
```

2.8.3 reinterpret_cast

- `reinterpret_cast` 主要有三种用途
 - 改变**指针或者引用的类型**
 - 将**指针或者引用转换成为足够长的整形**
 - 将**整型编程指针或者引用类型**
- **reinterpret_cast 运算符并不会改变括号中运算对象的值，而是对该对象从位模式上进行重新解释，地址仍是一样的**

2.8.4 dynamic_cast

- 跟其他几个不同，其他几个都是编译时完成的，`dynamic_cast` 是在**运行时**进行类型检查的
- 不能用于**内置**的基本数据类型的强制转换
- 如果成功的，将返回指向类的指针或者引用，转换失败的话会返回NULL
- 转换时基类一定要有虚函数，否则无法通过编译
 - 原因是虚函数表名这个类希望可以用基类指针指向派生类，这样转换才有意义
- 在类的向上转换的时候，和 `static_cast` 效果一样，但是向下转换的时候比 `static_cast` 更**安全**，因此要求也更高
- 来看一段代码

```
1 int main()
2 {
3     A a;
4     B b;
5     A *ap = &a;
6     if(dynamic_cast<B*>(ap)) {
7         cout << "OK1" << endl;
8     }
9     else {
10        cout << "Fail" << endl;
11    }
12    if(static_cast<B*>(ap)) {
```

```

13         cout << "OK2" << endl;
14     }
15     else {
16         cout << "Fail" << endl;
17     }
18     ap = &b;
19     if(dynamic_cast<B*>(ap)) {
20         cout << "OK3" << endl;
21     }
22     else {
23         cout << "Fail" << endl;
24     }
25     if(static_cast<B*>(ap)) {
26         cout << "OK4" << endl;
27     }
28     else {
29         cout << "Fail" << endl;
30     }
31     return 0;
32 }
```

- 运行的结果是第一个失败，其他的都成功
- 推测导致这个结果的原因：

- 当ap指向派生类的时候，进行强制类型转换变成派生类是可以成功的
- 当ap指向基类的时候，`dynamic_cast` 转换是否成功取决于**指针指向的类型和即将转换的类型是不是一样**，不一样就会失败，返回一个NULL，而static是可以成功的
- 其实是更安全的机制导致dynamic的检查更多，要求更高

2.9 列表初始化

- 列表初始化

```

1 class Test
2 {
3     int a;
4     int b;
5 public:
6     Test(int i, int j);
7 };
8 Test t{0, 0}; //C++11 only,
9 相当于 Test t(0,0);
10 Test *pT = new Test{1, 2}; //C++11 only,
11 相当于 Test* pT=new Test(1,2);
```

```

12 int *a = new int[3]{1, 2, 0}; //C++11 only
13
14 容器初始化
15 // C++11 container initializer
16 vector<string> vs={"first", "second",
17 "third"};
18 map<string,string> singers ={ {"Lady Gaga",
19 "+1 (212) 555-7890"}, {"Beyonce Knowles", "+1
20 (212) 555-0987"}};

```

2.10 输入/输出

- way in
 - void f(Student i);
 - a new object is to be **created** in f
 - void f(Student *p);
 - better with **const** if no intend to modify the object
 - void f(Student& i);
 - better with **const** if no intend to modify the object
- way out
 - Student f();
 - a new object is to be **created** at returning
 - Student* f();
 - what should it points to?
 - Student& f();
 - what should it refers to?

对于输入输出究竟选择哪种，往往会有阅读障碍，我们有以下tips

- Pass in an object if you want to **store** it
 - 指针会回收
- Pass in a const pointer or reference if you want to **get the values**
- Pass in a pointer or reference if you want to **do something** to it
- Pass out an object if you create it in the function
- Pass out pointer or reference of the passed in only
 - 只有传进去的是指针或者引用，传出来的也是
- **Never new something and return the pointer**

2.11 左值和右值

值代表的是运算结果

- 可以简单地认为能出现在赋值号左边的都是左值：
 - 变量本身、引用
 - *、[]运算的结果
 - 只能出现在赋值号右边的都是右值
 - 字面量
 - 表达式
 - 引用只能接受左值—>引用是左值的别名
 - 调用函数时的传参相当于参数变量在调用时的初始化
-
- &左值引用
 - &&右值引用，右值引用本身是一个左值，一旦做了引用，引用就是一个左值了

```
1 int x=20; // 左值
2 int&& rx = x * 2; // x*2的结果是一个右值，rx延长其生命周期；加上了&&，不会增加一个rx的本地变量，只是把x*2的值留下来，可能在寄存器，可能在栈里
3 int y = rx + 2; // 因此你可以重用它:42
4 rx = 100; // 一旦你初始化一个右值引用变量，该变量就成为了一个左值，可以被赋值
5 x = 30; // 不会修改rx
6 int&& rrx1 = x; // 非法：右值引用无法被左值初始化，无法接受左值
7 const int&& rrx2 = x; // 非法：右值引用无法被左值初始化
8
```

```
1 // 接收左值
2 void fun(int& lref) {
3     cout << "l-value" << endl;
4 }
5 // 接收右值
6 void fun(int&& rref) {
7     cout << "r-value" << endl;
8 }
9 int main() {
10     int x = 10;
11     fun(x); // output: l-value reference
12     fun(10); // output: r-value reference
13 }
14
```

```
15 //如果左值引用有const  
16 void fun(const int& clref) {  
17     cout << "l-value const reference\n";  
18 }  
19 //也可以接受右值
```

2.12 移动构造

```
1 vector<int> v1{1, 2, 3, 4};  
2 vector<int> v2 = v1; // 此时调用用拷贝构造函数,  
3 //v2是v1的副本  
4 vector<int> v3 = std::move(v1); // 此时调用移动构造函数  
5 //通过std::move将v1转化为右值, 从激发v3的移动构造函数,  
6 //实现移动语义  
7
```

移动构造存在的意义：

a = b赋值完之后，我们不需要b了

有没有一种可能，我们不copy了，我们直接把b的东西抢过来，让a指向b

```
1 A(const A&that) {  
2     ...//拷贝构造  
3 }  
4 A(const A&&that) { //传右值  
5     size = that.size;  
6     buf = that.buf;  
7 }
```

std::move作用主要可以将一个左值转换成右值引用，从而可以调用C++11右值引用的拷贝构造函数

3 重载

3.0 函数的重载 overload

- 多个同名函数的参数的**个数或者类型**不相同，这几个函数就构成重载关系
 - C++禁止通过返回值来判定函数重载的依据
 - 在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。

实现函数重载的条件

- 同一个作用域
- 参数个数不同

- 参数类型不同
- 参数顺序不同
- 调用的时候优先调用匹配度最高的重载函数，重载的优先级是：
 - 先找完全匹配的普通函数
 - 再找**模板函数**
 - 再找需要隐式转换的函数
- 当没有参数类型恰好匹配的重载函数的时候，就会将参数进行隐式转换之后来寻找可以调用的重载函数，此时如果重载函数有多个符合条件，就会产生error

```

1 #include <iostream>
2 using namespace std;
3
4 void foo(int a, int b)
5 {
6     cout<<1<<endl;
7 }
8
9 void foo(double a, double b)
10 {
11     cout<<2<<endl;
12 }
13
14 int main()
15 {
16     foo(1, 2);
17     foo(1.0, 2.0);
18     foo(1, 2.0); // 这一行是有语法错误的!
19 }
```

3.1 运算符的重载

- 重载的一些基本性质
 - 大部分运算符都可以重载
 - c++和python有
 - `.* :: ?:` 之类的运算符不能重载，`sizeof`, `typeid`, `static`, `dynamic`, `const`, `reinterpret`之类的**关键字不能重载**
 - `.*` 成员指针：指向成员的指针
 - 不能重载**不存在**的运算符

- 必须对于整个class或者enumeration type进行运算符的重载
- 重载之后的运算符依然保持原来一样的运算优先级和操作数的个数
- 如果是只读的，传进去的值应该是const的
- 返回值可以为左值

■ 1 | + - * / % ^ & | ~

- 逻辑运算符应该返回bool

■ 1 | ! && || < <= == >= >

- 运算符重载的基本语法

- 关键词operator
- 在类定义中对member function进行重载 return_type
`class_name::operator 运算符(parameters)` 此时参数为对应运算符所需参数-1

对单目运算符重载为友元函数时，可以说一个形参。而重载为成员函数时，不能显式说明形参。因为单目运算符就一个参数，而成员函数自带一个参数

■ 1 `const String String::operator + (const String& that) const;`
 2 //有一个默认参数是this
 3 //加法应该先创建一个东西
 4 //在类里面写不需要string::
 5 //传一个同类的参数，用引用；因为我们只是想要一下他的值

■ 1 `x+y ==> x.operator+(y)`
 2
 3 `Integer x(1), y(5), z;`
 4 `z=x+y` 如果当我发现我写了两个重载，就会出问题，不知道选哪个
 5 `z=x+3` 先把3转成对象
 6 `z=3+y` 成员函数不支持，会想把y转成int，不过外部函数可以
 7 `z=3+7` 外部函数直接编译的时候就`3+7=10`，然后把10赋给z的成员

- 对外部函数进行重载 返回值类型 operator运算符(参数表) 此时参数个数和对应的运算符的所需个数相同
 - 如果要访问类的private内容，需要声明这个重载函数为友元
 - 单目运算符一般声明为成员函数，双目运算符一般作为外部的函数
 - [], (), ->, =, ->*等运算符必须作为成员函数

将`=`重载，可以证明以下结论

```
1 | MyType b;
2 | MyType a = b; //拷贝构造
3 | a = b; //只是赋值
```

- 如果没有重载`=`，编译器会自动生成
- return a reference to `*this`
- 从右到左结合

- ```
1 | T& T::operator=(const T& rhs)
2 | //比如A=B, 返回类型是看A的
3 | {
4 | if(this!=&rhs)//判断地址是否相同；防止出现a=a；因为
5 | //赋值会先delete, 再new, 赋值, 如果相同的话, 就都delete了
6 | {
7 | perform assignment
8 | }
9 | return *this;//返回*this
 }
```

- 前缀的`++`和后缀的`++`

- 前后缀是指操作符在前还是后
  - `++i` (前缀)
  - `i++` (后缀)
- 我们要区别前缀和后缀，因为他们都是`++`
- 区别：后缀的`++`参数表是有个`int`类型的变量作为参数的

```
1 | Integer& operator++(); //prefix++
2 | Integer operator++(int x); //postfix++
```

```
1 | const Integer& Integer::operator++(){
2 | *this+=1;
3 | return *this;
4 |
5 | const Integer Integer::operator++(int x){ //这个x不会被使用, 只是规
6 | //定了, 告诉编译器这个是后缀++
7 | Integer old(*this);
8 | ++(*this);
9 | return old;
 }
```

- 比较大小关系的运算符重载时，可以通过**代码的复用**减少不必要的代码量；全都用小于解决问题

```

1 bool Integer::operator==(const Integer& rhs) const {
2 return i == rhs.i;
3 }
4 // implement lhs != rhs in terms of !(lhs == rhs)
5 bool Integer::operator!=(const Integer& rhs) const {
6 return !(*this == rhs);
7 }
8 bool Integer::operator<(const Integer& rhs) const {
9 return i < rhs.i;
10 }
11
12 // implement lhs > rhs in terms of lhs < rhs
13 bool Integer::operator>(const Integer& rhs) const {
14 return rhs < *this;
15 }
16 // implement lhs <= rhs in terms of !(rhs < lhs)
17 bool Integer::operator<=(const Integer& rhs) const {
18 return !(rhs < *this);
19 }
20 // implement lhs >= rhs in terms of !(lhs < rhs)
21 bool Integer::operator>=(const Integer& rhs) const {
22 return !(*this < rhs);
23 }
```

- `operator []`
  - 必须是**成员函数**
  - 单个参数，可以是int 也可以是string
  - 需要**返回一个引用**

- new, delete运算符重载

new, delete算符重载

```
class rect
{
private:
 int length, width;
public:
 rect(int l, int w)
 { length = l; width = w; }
 void *operator new(size_t size) //size_t是unsigned integer
 { return malloc(size); }

 void operator delete(void*p)
 { free(p); }

 void disp()
 { cout << "area: " << length * width << endl; }
};

void main()
{
 rect *p;
 p = new rect(5, 9);
 p->disp();
 delete p;
}
```

- 自定义的类型转换

- 例如

```
1 class Rational {
2 public:
3 double numerator_;
4 double denominator_;
5 operator double() const;
6 }
7 Rational::operator double() const {
8 return numerator_/(double)denominator_;
9 }
10 Rational r(1,3);
11 double d = 1.3 * r;//把rational变成double
```

- 不需要声明返回类型，是const的

- 可以是普通的类型，也可以是类

-

# C++ type conversions

- Built-in conversions

- *Primitive*

```
char ⇒ short ⇒ int ⇒ float ⇒ double
⇒ int ⇒ long
```

- *Implicit (for any type T)*

```
T ⇒ T& T& ⇒ T T* ⇒ void*
T[] ⇒ T* T* ⇒ T[] T ⇒ const T
```

- User-defined  $T \Rightarrow C$

- if  $C(T)$  is a valid constructor call for  $C$

- if operator  $C()$  is defined for  $T$

- 类型转换的重载和构造函数都可以解决类型转换的问题；但是不可以同时出现，会导致编译器不知道用哪个。

- 隐式类型转换：

- Single-argument constructors
  - implicit type conversion operators
  - 避免隐式转换的方法：把复制构造函数声明为 `explicit`，表示不能进行隐式的类型转换，只能显示的
    - 显示的转换只有  $\textcircled{O}$

C++会检查最合适的；意味着最便携

```
1 class PathName {
2 string name;
3 public:
4 // or could be multi-argument with defaults
5 PathName(const string&);
6 ~PathName();
7 };
8 ...
9 string abc("abc");
10 PathName xyz(abc); // OK!
11 xyz = abc; // OK abc => PathName
12
13 class PathName {
14 string name;
15 public:
16 explicit PathName(const string&);
17 ~PathName();
```

```
18 };
19 ...
20 string abc("abc");
21 PathName xyz(abc); // OK!
22 xyz = abc; // error! 隐式的
23
```

```
1 #include <iostream>
2 #include <string.h>
3 using namespace std;
4 class mystring
5 {
6 char *buf;
7
8 public:
9 mystring()
10 {
11 buf = new char[100];
12 cout << "construct 0" << endl;
13 }
14 mystring(char *_buf)
15 {
16 buf = new char[100];
17 strcpy(buf, _buf);
18 cout << "construct 1" << endl;
19 }
20 ~mystring()
21 {
22 delete buf;
23 cout << "destruct" << endl;
24 }
25 void operator=(mystring const &str1)
26 {
27 strcpy(this->buf, "C++");
28 strcat(this->buf, str1.buf);
29 }
30 //如果不加引用，就会拷贝构造一个新的，而char*的默认构造函数是直接获取
指针，不会开新的内存，而拷贝的mystring析构的时候释放了一块内存，原来的
mystring析构的时候也释放了同一块，因为是浅拷贝；所以尽可能用string而
不用char*
31 void show()
32 {
33 cout << buf << endl;
```

```
34 }
35 };
36
37 int main()
38 {
39 char ch[100];
40 cin >> ch;
41 // printf("%s",ch);
42 mystring str1(ch);
43 str1.show();
44 mystring str2;
45 str2 = str1;
46 str2.show();
47 return 0;
48 }
```

### 3.2 输入输出流的重载

- 全局的函数，不是成员函数
- stream的种类
  - Text stream 用ASCII码来解析，包括files和character buffers
  - Binary stream 二进制数据，no translation
- stream的类型：
  - cin标准输入，使用流提取运算符>>，在C++标准库的istream中定义
  - cout标准输出，使用流插入运算符<<，在ostream中定义
  - cerr标准错误提示(unbuffered error)
  - clog标准错误提示(buffered error)
- 对>>和<<的重载
  - Has to be a **2-argument** free function
  - **不能是成员函数**

```

1 oostream& operator << (ostream& output, Complex& c)
2 {
3 output<<"(";<<c.real;
4 if(c.imag>=0) output<<"+";//虚部为正数时，在虚部前加“+”号
5 output<<c.imag<<"i)"<<endl; //虚部为负数时，在虚部前不加“+”号
6 return output;
7 }
8 istream& operator >> (istream& input,Complex& c) //定义重载运算
9 符“>>”
10 {
11 cout<<"input real part and imaginary part of complex
12 number:";
13 input>>c.real>>c.imag;
14 return input;
15 }
```

- 对manipulators的重载(类似endl)

```

1 oostream& tab (ostream& out)
2 {
3 return out << '\t';
4 }
5 cout<<"Hello"<<tab<<"world"<<endl;
```

- 控制输出的格式，使用头文件 #include<iomanip>
- 其他的输入输出的方式：
  - `int get()` 支持单个字符读入
    - Returns the **next character** in the stream
    - Returns EOF if no characters left
  - `get(char *buf, int limit, char delim='\n')`
  - `getline(char *buf, int limit, char delim='\n')`

## 4 模板

### 4.1 namespace 命名空间

- 使用命名空间来划分全局的各类类名可以避免名字的冲突，可以在**不同的命名空间里定义相同的变量名**

```
1 namespace old1{
2 void f();
3 void g();
4 }
5 namespace old2{
6 void f();
7 void g();
8 }
```

- 他是一个声明，**不需要逗号**
- 在引用的时候加上命名空间的限制符就可以了
- 或者也可以用 `using namespace xxx` 来说明程序中接下来用哪个命名空间中的东西

```
1 namespace space1{
2 string name = "randomstar";
3 void foo();
4 class cat{
5 public:
6 void meow();
7 };
8 }
9
10 namespace space2{
11 string name = "ToyamaKasumi";
12 void foo();
13 }
14 namespace space2{
15 void g(); //是同一个，想想编译出来的名字
16 }
17
18 int main()
19 {
20 cout<<space1::name<<endl;
21 using namespace space2;
22 cout<<name<<endl;
23 space1::foo();
24 using space1::foo;
25 using space1::cat;
26 foo();
27 cat c;
28 c.meow();
29 using namespace space1;
30 using namespace space2;
```

```
31 | foo(); //会出错，同时引用多个namespace，且有相同名字的函数
32 | }
```

链接器不判断函数的参数类型，只记得名字

c++编译器会：

f(int)会编译成\_\_f\_int

A::f(double)编译成\_\_A\_f\_double

老版本的c不会

## 4.2 template编程

Why templates?

假设你有a list of X and a list of Y，他们有相近的代码，但是存储不同的type

- 相同的父类
  - C++可以存储对象，会很复杂且不可行
- Clone mode
  - hard to manage
- Untyped lists
  - type unsafe

灵活结构成员

```
1 typedef struct
2 {
3 int len;
4 char content[]; //char是任意东西
5 }string
```

string\* str = (string\*)malloc(sizeof(int)+32)

实际上为\*str里的content分配了32个字节

我们可以给多少，就分配给content多少

真正的解决方案

元代码：template

用来让编译器在**编译过程中产生代码**

PL

- 命令式
  - 面向过程
  - 面向对象
  - 泛型
- 函数式
- 逻辑式
- 模板编程是一种复用代码的手段，是generic programming(泛型编程)，把**变量的类型当作参数来声明**
  - 类型为变量，后来才确定
  - 之前java没有，后来加了，叫做GP
  - 多文件，模板一定要放到.h文件里，模板是声明；类似于inline
  - 模板参数表中有**类型参数和非类型参数**
- **函数模板**：使用关键字template 后面加若干个变量作为类型声明一个函数模板，类型变量可以代替**函数的返回值类型，参数类型和函数中的变量的类型**

template下面一句话是什么，就是什么模板

T就是未确定的类型

```

1 template < class T >
2 void swap(T& x, T& y) {
3 T temp = x;
4 x = y;
5 y = temp;
6 }
```

- 函数模板需要**实例化**(instantiation)之后再使用，如果没有被调用就不会被实例化
  - 实例化是将模板函数中的模板替换为**对应的变量类型**，然后生成一个对应的函数
  - 函数模板——>函数
    - 模板实参实例化的函数称为模板函数
- 函数模板在使用过程中不能发生参数和返回值的类型转换，必须**要类型完全匹配才能使用**

```
1 void f(float i, float k) {};
2 void f(int i, int k) {};
3 template <class T>
4 void f(T t, T u) {};
5 f(1.0, 2.0);
6 //会编译成_f<double>_double_double
7 f(1, 2);
8 f(1, 2.0)
```

```
1 //如果一个template函数的输入不是T，需要在调用的时候注明T的类
2 //型，举例如下
3 f<double>(c)
```

## ◦ 函数重载时候的优先级

- 先找是否有完全匹配的普通函数
  - 再找是否有完全匹配的模板函数
  - 再找有没有通过进行隐式类型转换可以调用的函数
- 类模板

```
1 template <class T>
2 class Vector
3 {
4 public:
5 Vector(int);
6 ~Vector();
7 Vector(const Vector &);
8 Vector &operator=(const Vector &);
9 T &operator[](int);
10
11 private:
12 T *m_elements;
13 int m_size;
14 };
15
16 template <class T>
17 Vector<T>::Vector(int size) : m_size(size)
18 {
19 m_elements = new T[m_size];
20 }
21 template <class T>
22 //注意不要忘记<T>
23 T &vector<T>::operator[](int idx)
24 {
```

```

25 if (indx < m_size && indx > 0)
26 {
27 return m_elements[indx];
28 }
29 else
30 {
31 ...
32 }
33 }
34
35 vector<int> v1(100);
36 vector<Complex> v2(256);
37 v1[20] = 10;
38 v2[20] = v1[20]; // ok if int->Complex
 // defined

```

- 类模板里的函数都是**函数模板**
  - 在类里面写不需要特殊处理
  - 在外面写需要重复一下 `template<class T>`, 记得加上 `vector<T>::`
- 和函数模板差不多, template中声明的既可以是类内各种需要变量类型的地方
- 模板类也可以继承非模板类, 也可以继承模板类(**需要实例化**)

```

1 template <class A>
2 class Derived : public List<A>
3 {
4 };
5 class SupervisorGroup : public List<Employee *>
6 {
7 };

```

- 非模板类可以继承自模板类(**需要实例化**)
  - | 继承模板类的都需要实例化

- `Vector< int (*)>(vector&, int)>`里面那个是**函数指针**

- Expression parameter
  - 模板中的可以声明一些常数, class和typename的类型名可以有**默认值 default value**, 比如 `template<typename T = int>`

```

1 template <class T, int bounds = 100>
2 class FixedVector{
3 public:

```

```
4 T& operator[](int x);
5 T elements[bounds];
6 };
7
8 template <class T, int bounds>
9 T& FixedVector<T, bounds>::operator[] (int i){
10 return elements[i]; //no error checking
11 }
12
13 //usage
14 FixedVector<int, 50> v1;
15 FixedVector<int, 10*10> v2;
16 FixedVector<int> v3 //default value.
```

- 模板一些小点
  - friends
  - static members
  - compiler/linker有一种机制防止multiple definitions
    - 假如实例化两次，怎么防止multiple
      - 做成static
      - 做成weak
  -

```

1 #include <iostream>
2 using namespace std;
3 template <class T>
4 class A {
5 template<class Tp> // base T, int bounds>
6 friend void f(A<Tp> &a);
7 int k;
8 public:
9 return elements[i]; // no error checking
10 static int kk;
11 };
12
13 void f(A<int> &a) {std::cout << a.k;}
14
15 template<> FixedVector<int> v3; // default value
16 int A<int>::kk = 0;
17
18 int main() {
19 A<int> a;
20 f(a);
21 cout << A<int>::kk;
22 }

```

PS C:\Users\lenovo\OneDrive - zju.edu.cn\Desktop> g++ .\test.cpp  
 PS C:\Users\lenovo\OneDrive - zju.edu.cn\Desktop> .\a.exe  
 00

- Specialization 特化--用于萃取器

- 全特化：将模板类中所有的类型参数赋予明确的类型，并写了一个类名和主模板类名相同的类，这个类就是全特化类
  - 全特化之后，已经失去了template的特性

```

1 template<class T> bool compare(T x, T y)
2 {
3 return x>y;
4 }
5
6 // 这个就是对上面写的模板函数的全特化
7 template<> bool compare(int x, int y)
8 {
9 return x>y;
10 }

```

- 偏特化：只给模板类赋一部分的类型，得到的**偏特化类/函数**可以作为一个子模版使用
  - 还保留了一部分template的功能
- 模板类调用的**优先级**

- 全特化类>偏特化类>主版本的模板类（就是直接调用模板类，最常见的用法）
- 有隐式转换的优先级比较低，先考虑所有不需要隐式转换的模板，再考虑需要隐式转换的模板

## 4.3 STL和迭代器

更强鲁棒性：不需要考虑内存

可迁移性

可维护性

- STL=Standard Template Library 是标准库的一部分，使用C++STL可以减少开发时间，提高可读性和鲁棒性，STL包含了：
  - a pair class(pairs of anything )
  - 容器
    - vector(expandable array)
    - deque(expandable array,expands at both ends)
    - list(double-linked)
    - sets and maps
      - set就相当于map，只不过键值和内容一样
      - 字母序
  - 迭代器
  - 基本的算法
    - sort
      - sort(a.begin(),a.end(),cmp)
        - bool cmp(int a,int b){return a>b;}
      - search
  - 数据结构
    - map
      - any key type,any value type
      - sorted
    - vector
      - like c array,but auto-extending
    - list
      - doubly-linked list
- All sequential containers

- vector: variable array
- deque: aual-end queue
- list: double-linked-list
- forward\_list: as it
- array: as “array”
- string: char.array

- STL容器的使用

- 容量无限大，随着input而增长
- size返回了他的空间；记录了存入的顺序，你可以按顺序取回他们
- 在STL容器里用你自己的class的话，需要有以下东西
  - Assignment Operator
  - operator = ()
  - Default Constructor
- 关于vector：

两种实现方式

- 链表
- 数组

```

1 #include<iostream>
2 #include<vector>
3 using namespace std;
4
5 int main()
6 {
7 vector<int> x;
8 for(int a=0;a<1000;a++)
9 x.push_back(a);
10 vector<int>::iterator p;
11 //只是声明，还需要定义
12 for(p = x.begin();p<x.end();p++)
13 cout<<*p<<" ";
14 return 0;
15 }
16 //你猜，等同于上面
17 for(auto k:x){
18 cout<< k << " "
19 }
20 // -std=c++11
21 //冒号后面可以是集合，也可以是数组
22 //for each循环

```

- Constructors
  - vectorc;
  - vectorc1(c2);
    - 复制c2到c1
- Simple Methods
  - V.size()
  - V.empty()  
==,!=<>,<=,>=
  - V.swap(v2)
- Iterators
  - I.begin() //first position
  - I.end() //last position
- Element access
  - V.at()
  - V[index]
  - V.front() //first item
  - V.back() //last item
- Add/Remove/Find
  - V.push\_back(e)
    - 会调用构造函数，其实是构造一个，然后push进去
  - V.pop\_back()
  - V.insert(pos,e)
  - V.erase(pos)
  - V.clear()
  - V.find(first,last,item)

将栈里的东西复制，只是浅拷贝

push和insert不会改变size，vector是可扩充的，你在一个很奇怪的地址可以插入东西，但是不算在我们的size里

- List:

类似与vector

但是一定用双向链表实现

遍历两者差不多

list最快的是插入和删除，只有大量使用插入和删除的时候才可

vector最快的是random access

list需要保存链表，vector占据的空间较小，一般优先用vector

表：

- uuid保存key
- 冷保存和热保存
- 分库分表，在不同的表，不同的库里写
- Ability to compare lists (==, !=, <, <=, >, >=)
- Ability to access front and back of list
  - x.front()
  - x.back()
- Ability to assign items to a list, remove items
  - x.push\_back(item)
  - x.push\_front(item)
  - x.pop\_back()
  - x.pop\_front()
  - x.remove(item)
- 不能用 `p < s.end()`，因为地址不一定严格小，所以用 `p!=s.end()`

```
1 #include <iostream>
2 using namespace std;
3 #include <list>
4 #include <string>
5 int main() {
6 list<string> s;
7 s.push_back("hello");
8 s.push_back("world");
9 s.push_front("tide");
10 s.push_front("crimson");
11 s.push_front("alabama");
12 list<string>::iterator p;
13 for (p=s.begin(); p!=s.end(); p++)
14 cout << *p << " ";
15 cout << endl;
16 }
17 //取东西要*
```

- 关于map：map的key必须是可以满足**可以排序**性质的，如果是自定义的类需要**重载比较函数**，否则这个类不能作为key值

map可以用[]

## map迭代也是判断不等于end

- pair (key: value)
- key可以是任意类型
- map会根据key来进行排序，当key的类型为int时为升序，当key的类型为string时为字典序

```
1 #include <map>
2 #include <string>
3 map<string, float> price;
4 price["snapple"] = 0.75;
5 price["coke"] = 0.50;
6 string item;
7 double total=0;
8 while (cin >> item)
9 total += price[item];
```

```
1 //map自动字母序，本质是红黑
2 #include <iostream>
3 #include <map>
4 using namespace std;
5 int main()
6 {
7 int n;
8 map<string, int> m;
9 map<string, int>::iterator u;
10 cin >> n;
11 getchar();
12 for (int i = 0; i < n; i++)
13 {
14 int num;
15 string temp;
16 cin >> num;
17 getchar();
18 m.clear();
19 for (int j = 0; j < num; j++)
20 {
21 string s;
22 getline(cin, s);
23 m[s]++;
24 }
25 for (u = m.begin(); u != m.end(); u++)
26 {
```

```

27 cout << u->first << " " << u->second <<
28 endl;
29 }
30 return 0;
31 }
32 //->first ->second输出键值和value

```

- 迭代器: `STL<parameters>::iterator xxx`

- 迭代器是一种顺序访问容器的方式: Generalization of pointers; 理解为指针
- 常用的迭代器按功能强弱分为输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机访问迭代器 5 种

输入迭代器和输出迭代器比较特殊, 它们不是把数组或容器当做操作对象, 而是把输入流/输出流作为操作对象。

- 1. 前向迭代器 (forward iterator)

假设 `p` 是一个前向迭代器, 则 `p` 支持 `++p`, `p++`, `*p` 操作, 还可以被复制或赋值, 可以用 `==` 和 `!=` 运算符进行比较。此外, 两个正向迭代器可以互相赋值。

- 2. 双向迭代器 (bidirectional iterator)

双向迭代器具有正向迭代器的全部功能, 除此之外, 假设 `p` 是一个双向迭代器, 则还可以进行 `--p` 或者 `p--` 操作 (即一次向后移动一个位置)。

- 3. 随机访问迭代器 (random access iterator)

随机访问迭代器具有双向迭代器的全部功能。除此之外, 假设 `p` 是一个随机访问迭代器, `i` 是一个整型变量或常量, 则 `p` 还支持以下操作:

- `p+=i`: 使得 `p` 往后移动 `i` 个元素。
- `p-=i`: 使得 `p` 往前移动 `i` 个元素。
- `p+i`: 返回 `p` 后面第 `i` 个元素的迭代器。
- `p-i`: 返回 `p` 前面第 `i` 个元素的迭代器。
- `p[i]`: 返回 `p` 后面第 `i` 个元素的引用。

此外, 两个随机访问迭代器 `p1`、`p2` 还可以用 `<`、`>`、`<=`、`>=` 运算符进行比较。另外, 表达式 `p2-p1` 也是有定义的, 其返回值表示 `p2` 所指向元素和 `p1` 所指向元素的序号之差 (也可以说是 `p2` 和 `p1` 之间的元素个数减一)。

表 1 所示, 是 C++ 11 标准中不同容器指定使用的迭代器类型。

| 容器    | 对应的迭代器类型 |
|-------|----------|
| array | 随机访问迭代器  |

| vector<br>容器                       | 随机访问迭代器类型 |
|------------------------------------|-----------|
| deque                              | 随机访问迭代器   |
| list                               | 双向迭代器     |
| set / multiset                     | 双向迭代器     |
| map / multimap                     | 双向迭代器     |
| forward_list                       | 前向迭代器     |
| unordered_map / unordered_multimap | 前向迭代器     |
| unordered_set / unordered_multiset | 前向迭代器     |
| stack                              | 不支持迭代器    |
| queue                              | 不支持迭代器    |

注意，容器适配器 stack 和 queue 没有迭代器，它们包含有一些成员函数，可以用来对元素进行访问。

- 两个特殊的迭代器：begin( ) 和 end( ) 分别表示容器的**头和尾(最后一个元素的后面)**
  - 很多时候end并不能达到，因此到结束的判断条件往往是 `!=st1.end()`
- 迭代器支持的操作
- `iter++ iter+=2` (不是任何容器的迭代器都可以)
  - `*iter`

```

1 list<int> L;
2 list<int>::iterator li;
3 li=L.begin();
4 ++li;
5 *li = 10;
6 L.erase(li); //
7 ++li; //error!!! 没有li了，但是可以使用li=L.erase(li);这样会给你一个新的，还可以继续使用li

```

- 可以看成是一种**泛化的指针**
- Typedefs

- o

```
1 -map > phonebook;
2 -map >::iterator finger;
3
4 -typedef PB map<Name, list<PhoneNum> >;
5 -PB phonebook;
6 -PB::iterator finger;
```

- Pitfall

- 性能差

- ```
1 if(foo["bob"]==1)
2 //会默认认为你创造一个bob
3 if(foo.count("bob"))
4 //用count判断是不是存在
```

- ```
1 if(my_list.count()==0){.....}
2 //slow
3 if(my_list.empty()){.....}
4 //fast
```

## 2-5 分数 2

下列创建vector容器对象的方法中，错误的是。

- A. `vector<int> v(10);`
- B. `vector<int> v(10, 1);`
- C. `vector<int> v{10, 1};`
- D. `vector<int> v = (10, 1);`

(35条消息) C++ vector的初始化 锤某的博客 CSDN 博客 c++ vector 初始化

- a: 用10个0初始化
- b: 用10个1初始化
- c: 初始化10,1, 00000
- d: (10,1)是括号运算符，最后为1

## 5 Exceptions 异常处理

构造函数和析构函数不抛出异常

- 用于异常处理的语法

- `try{ } catch{ }`：捕获`throw`抛出的异常
  - `catch (...)` 表示捕捉**所有可能的异常**
  - `try`括号中的代码被称为保护代码
- `throw`语句：抛出异常
  - `throw exp;` 抛出一个表达式：`throw`的参数可以是任何的表达式，表达式中的类型决定了抛出的结果的类型
  - `throw;` 只有在**catch子句中有效**，把原本捕捉到的异常抛出

```
1 try
2 {
3 open the file;
4 determine its size;
5 allocate that much memory;
6 read the file into memory;
7 close the file;
8 }
9 catch (fileOpenFailed)
10 {
11 doSomething;
12 }
13 catch (sizeDeterminationFailed)
14 {
15 doSomething;
16 }
17 catch (memoryAllocationFailed)
18 {
19 doSomething;
20 }
21 catch (readFailed)
22 {
23 doSomething;
24 }
25 catch (fileCloseFailed)
26 {
27 doSomething;
28 }
```

- 异常处理的执行过程

- 程序按照正常的顺序执行，到达try语句，开始执行try内的保护段
- 如果在保护段执行期间没有发生异常，那么跳过所有的catch
- 如果保护段的执行期间有调用的任何函数中有异常，则可以通过**throw**创建一个异常对象并**抛出**，程序转到对应的**catch**处理段，会不断的去找**catch**，跳过很多代码，眼里只有**catch**。
- 首先要按顺序寻找匹配的**catch**处理器，如果没有找到，则**terminate()**会被自动调用，该函数会调用**abort**终止程序
  - 如果在函数中进行异常处理并且触发了**terminate**，那么终止的是当前函数
  - 异常类型需要**严格的匹配**
  - 子类异常可以用父类来**catch**，父类不可用子类异常**catch**；当如果父类、子类都在，只会捕捉第一个（**通常是子类**，如果父类在上，则子类永远不会被捕获）
- 如果找到了匹配的**catch**处理程序，并且通过值进行捕获，则其形参通过**拷贝异常对象进行初始化**，在形参被初始化之后，展开栈的过程开始，开始对对应的try块中，从开始到异常丢弃地点之间创建的所有局部对象的析构

```

1 #include <iostream>
2 using namespace std;
3
4 double division(int a, int b)
5 {
6 if(b == 0)
7 {
8 throw "Division by zero condition!";
9 }
10 return (a/b);
11}
12
13 int main ()
14 {
15 int x = 50;
16 int y = 0;
17 double z = 0;
18
19 try {
20 z = division(x, y);
21 cout << z << endl;
22 }catch (const char* msg) {
23 cerr << msg << endl;
24 }
25 }
```

```
26 return 0;
27 }
```

- C++中自带的**异常的继承体系**, 定义在头文件 `<exception>` 中
  - `what`方法给出了产生异常的原因, 是异常类之间都有的公共方法, 已经被所有的子异常类重载
  - 自定义的异常类: 需要**继承exception类**

```
1 #include <iostream>
2 #include <exception>
3 using namespace std;
4
5 struct MyException : public exception
6 {
7 const char * what () const throw ()
8 {
9 return "C++ Exception";
10 }
11};
12
13 int main()
14 {
15 try
16 {
17 throw MyException();
18 }
19 catch(MyException& e)
20 {
21 std::cout << "MyException caught" << std::endl;
22 std::cout << e.what() << std::endl;
23 }
24 catch(std::exception& e)
25 {
26 //其他的错误
27 }
28 }
```

- 函数**定义的异常声明**
  - 可以在函数名后面加 `noexcept` 关键字, 说明该函数在运行的过程中**不抛出任何异常**, 如果还是产生了异常, 就会调用 `std::terminate` 终止程序
  - 可以在函数声明中列出**所有可能**抛出的异常类型, **不可能抛出其他类型的异常**, 比如 `double f(int, int) throw(int);` 如果是 `throw()` 表示**不抛出异常**, 就算函数里有 `throw` 也不会执行; 可以放多个, 按逗号隔开

`void fun() throw();` //表示fun函数不允许抛出任何异常，即fun函数是异常安全的。

- `void fun() throw(...);` //表示fun函数可以抛出任何形式的异常。

`void fun() throw(exceptionType);` // 表示fun函数只能抛出exceptionType类型的异常。

- `throw`会导致一个函数没有执行完毕，但是在函数`throw`之前会执行所有**局部变量的析构函数**

- 最好不要在析构函数中抛出异常

- 会内存泄露；会导致不`free`

- 构造函数里面抛异常

- 解决方案：两阶段构造；构造函数不能接触资源，可能失败的事情不在构造函数里面做；申请资源的，在后面再做

- 1 //以下方法对象必须是new出来的  
2 A(){  
3 try{}  
4 catch(...)//表示都可以  
5 {  
6 delete this;  
7 throw;  
8 }  
9 }

- 设计原则：连续的程序指令 需要在一个try块

```
1 class T
2 {
3 T(){
4 cout<<"T()"<<endl;
5 }
6 ~T(){
7 cout<<"~T()"<<endl;
8 }
9 }
10
11 void foo()
12 {
13 T t;
14 throw 1;
15 }
16
17 int main()
18 {
19 try{
```

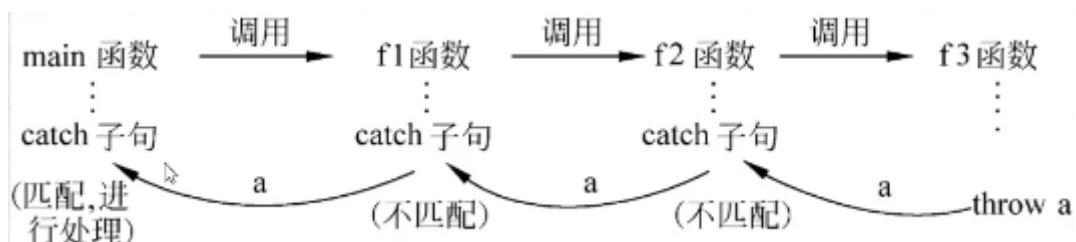
```

20 foo();
21 }
22 catch(...){
23 cout<<"Catched!"<<endl;
24 }
25 return 0;
26 }
27
28 //运行的结果是
29 T()
30 ~T()
31 Caught

```

重抛异常：处理不了的异常，可以通过在catch结构中调用throw重新抛出异常，将当前异常传递到外部的try-catch结构中；直到存在于一个try里面

|                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> int main() {     void f1();     try {         f1();     }     catch (double) {         cout &lt;&lt; "caught double exception." &lt;&lt; endl;     }     cout &lt;&lt; "after catch block. (double)" &lt;&lt; endl;     return 0; } void f1() {     try {         f2();     }     catch (char) {         cout &lt;&lt; "caught char exception.";     }     cout &lt;&lt; "after catch block. (char)" &lt;&lt; endl; } </pre> | <pre> void f2() {     try {         f3();     }     catch (int) {         cout &lt;&lt; "caught int exception." &lt;&lt; endl;     }     cout &lt;&lt; "after catch block. (int)" &lt;&lt; endl; } void f3() {     double a = 0;     try {         throw a; // throw double     }     catch (float) {         cout &lt;&lt; "caught float exception." &lt;&lt; endl;     }     cout &lt;&lt; "after catch block. (float)" &lt;&lt; endl; } </pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



**caught double exception.**  
**after catch block. (double)**

最终答案

## 6 Smart Pointer 智能指针

自动内存管理——C++没有实现

自动内存删除：new之后自动delete

## Java不使用指针引用计数

```
1 | a = new A();
2 | b = a;
3 | b = new A();
4 | a = b;
```

引用计数存在一种bug，存在循环引用，产生垃圾岛

Java用检查堆和栈的方式，来完成自动内存管理

- 普通指针管理内存容易造成内存泄漏(比如用完忘记释放)，C++提供了智能指针用于内存的管理
  - 智能指针使用了一种RAII(资源获取即初始化)技术对普通的指针进行了封装，使得智能指针实质上是一个**对象**，但是行为表现得像一个**指针**
  - 智能指针的相关内容包含在头文件
- 标准库中支持的智能指针类型
  - `std::unique_ptr`只允许这个指针来管理对应的资源，**不允许指针之间的拷贝**，但是
    - 不允许进行指针的赋值和拷贝
    - 可以用**move方法来移动指针的所有权**，比如 `unique_ptr<int> p2 = move(p1);`
    - 可以用**release方法来释放指针的所有权**

```
1 | class A{
2 | public:
3 | A() {
4 | cout<<"A()"<<endl;
5 | }
6 | ~A() {
7 | cout<<"~A()"<<endl;
8 | }
9 | void foo() {
10 | cout<<"1"<<endl;
11 | }
12 |
13 |
14 | int main()
15 | {
16 | {
17 | unique_ptr<A> pa(new A());
18 | pa->foo();
19 | }
```

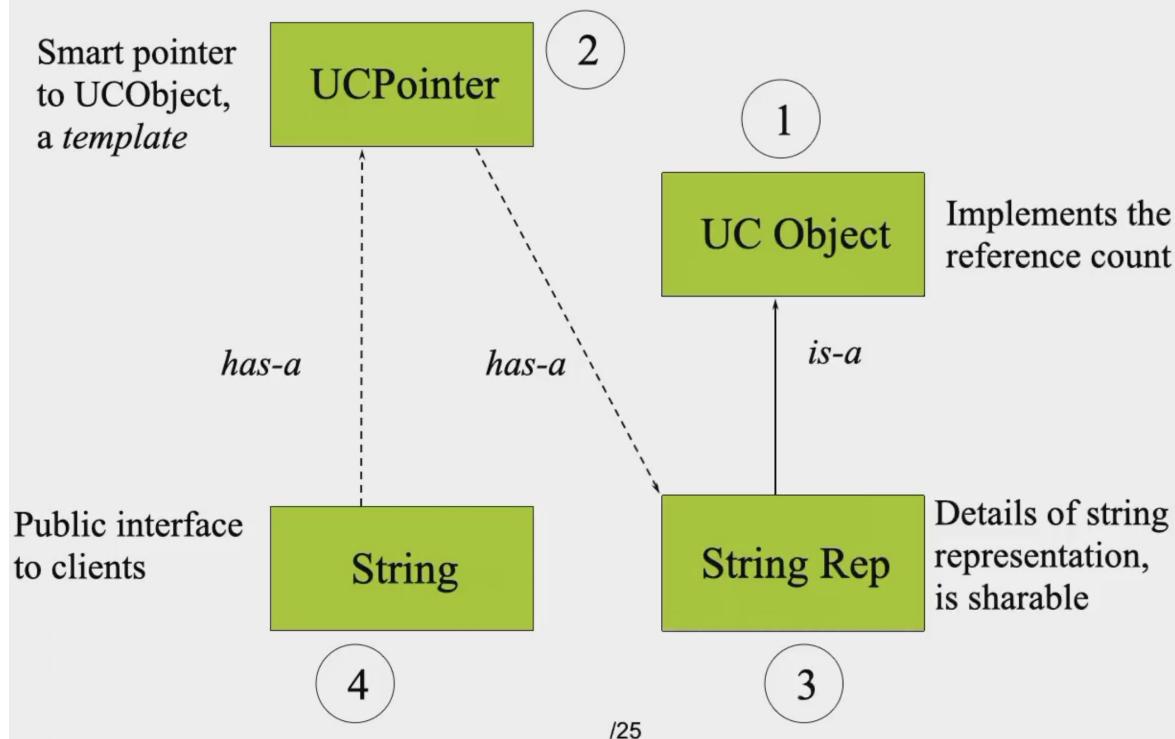
- `std::shared_ptr` 非独占的指针，可以多个指针管理一个对象。  
`shared_ptr` 使用**引用计数**，每一个`shared_ptr` 的拷贝都指向相同的内存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，自动删除所指向的堆内存。`shared_ptr` 内部的引用计数是**线程安全**的，但是对象的读取需要**加锁**
  - 可以用 `use_count` 函数来查看某个对象拥有的指针的个数
  - 可以用 `get` 函数来获取原始指针
  - 不能用一个原始指针初始化多个 `shared_ptr` 否则会造成内存多次释放

```

1 #include<iostream>
2 #include<memory>
3 int main()
4 {
5 int a = 10;
6 std::shared_ptr<int> ptra = make_shared<int>(a);
7 std::shared_ptr<int> ptrb(ptra);
8 cout<<ptra.use_count<<endl; //输出的结果是2
9 }
```

- `std::weak_ptr` 是没用重载\*和->操作符，没有普通指针具有的行为，最大的作用在于协助其他指针的工作，查看对象中的资源使用情况
  - `weak_ptr` 可以通过一个`shared_ptr` 或者 `weak_ptr` 来进行构造，获取观测权，但是 `weak_ptr` **不会造成指针引用计数的增加**

# The four classes involved



一个类里有一个智能指针，指向一个对象，实现引用量的计数

```
1 // ucobject.h
2 #include <iostream>
3 using namespace std;
4 #include "string.h"
5 #include <assert.h>
6 class UCObject
7 {
8 public:
9 UCObject() : mRefCount(0) {}
10 virtual ~UCObject()
11 {
12 assert(mRefCount == 0);
13 }
14 UCObject(const UCObject &) : mRefCount(0) {} //新复制出来的
15 //拷贝构造 被引用的次数是0
16 void incr()
17 {
18 mRefCount++;
19 }
20 void decr()
21 {
22 mRefCount--;
23 if (mRefCount == 0)
```

```
23 delete this;
24 }
25 int references()
26 {
27 return mRefCount;
28 }
29
30 private:
31 int mRefCount;
32 };
33
34 // UCPointer.h
35 template <class T>
36 class UCPointer
37 {
38 private:
39 T *m_pobj;
40 void increment()
41 {
42 if (m_pobj)
43 m_pobj->incr(); //指向一个obj, 让他增加
44 }
45 void decrement()
46 {
47 if (m_pobj)
48 m_pobj->decr();
49 }
50
51 public:
52 UCPointer(T *r = 0) : m_pobj(r)
53 {
54 increment();
55 }
56 ~UCPointer()
57 {
58 decrement();
59 }
60 UCPointer(const UCPointer<T> &p) //引用内容复制, 引用次数+1
61 {
62 m_pobj = p.m_pobj;
63 increment();
64 }
65 UCPointer &operator=(const UCPointer<T> &p)
66 {
67 if (m_pobj != p.m_pobj)
```

```
68 {
69 decrement();
70 m_pobj = p.m_pobj;
71 increment();
72 }
73 return *this;
74 }
75 T *operator->() const
76 {
77 return m_pobj;
78 }
79 T &operator*() const
80 {
81 return *m_pobj;
82 }
83 };
84 // // StringRep
85 // #include "UCObject.h"
86 #include <cstring>
87
88 class StringRep : public UCObject
89 {
90 public:
91 StringRep(const char *s)
92 {
93 if (s)
94 {
95 int len = strlen(s) + 1;
96 m_pChars = new char[len];
97 strcpy(m_pChars, s);
98 }
99 else
100 {
101 m_pChars = new char[1];
102 *m_pChars = '\0';
103 }
104 };
105 ~StringRep()
106 {
107 delete[] m_pChars;
108 }
109 StringRep(const StringRep &sr) // UCO本身的拷贝构造在制造新的
东西
110 {
111 int len = sr.length();
```

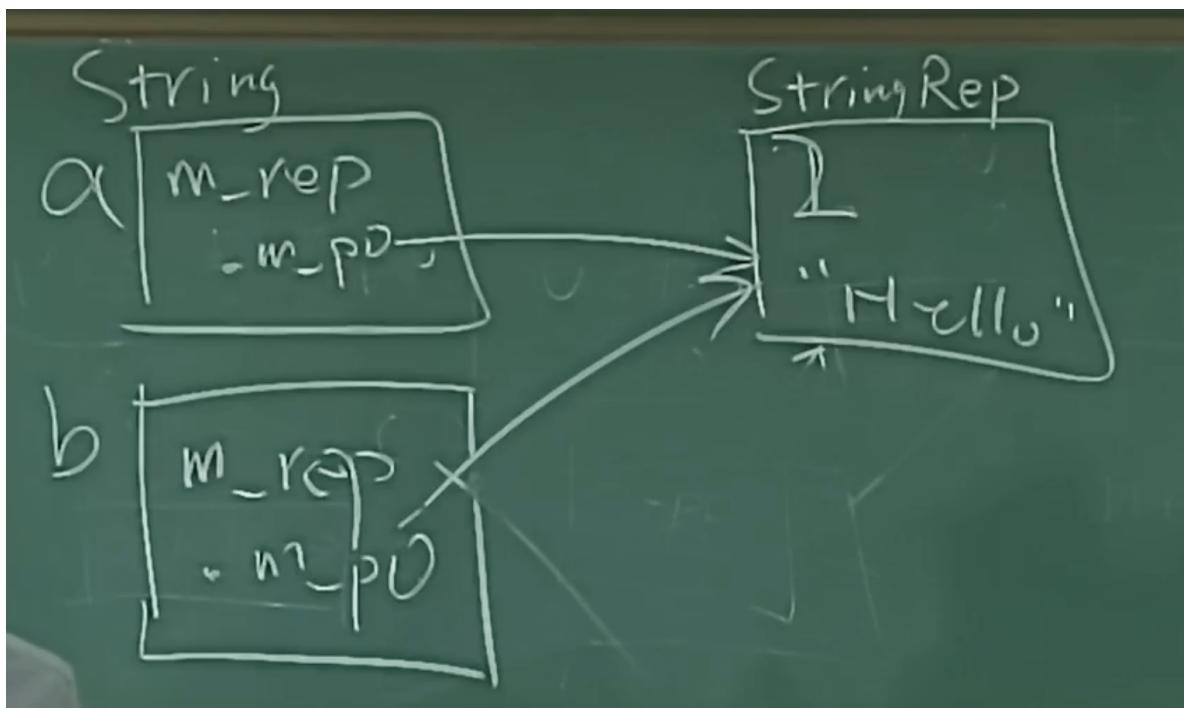
```
112 m_pChars = new char[len + 1];
113 strcpy(m_pChars, sr.m_pChars);
114 }
115 int length() const
116 {
117 return strlen(m_pChars);
118 }
119 int equal(const StringRep &sp) const
120 {
121 return (strcmp(m_pChars, sp.m_pChars) == 0);
122 }
123
124 private:
125 char *m_pChars;
126 //禁止做赋值
127 void operator=(const StringRep &) {}
128 };
129 // String
130 class String
131 {
132 public:
133 String(const char *s) : m_rep(0)
134 {
135 m_rep = new StringRep(s);
136 }
137 ~String() {}
138 String(const String &s) : m_rep(s.m_rep) {}
139 String &operator=(const String &s)
140 {
141 m_rep = s.m_rep;
142 return *this;
143 }
144 int operator==(const String &s) const
145 {
146 // overloaded -> forwards to StringRep
147 return m_rep->equal(*s.m_rep); // smart ptr *
148 }
149 int length() const
150 {
151 return m_rep->length();
152 }
153 operator const char *() const;
154
155 private:
156 UCPointer<StringRep> m_rep;
```

```

157 };
158 // main
159 int main()
{
160 {
161 cout << "hello\n";
162
163 String a = "Hello";
164 String b = a;
165 b = "bye";
166 b = a;
167 cout << (a == b) << endl;
168 }

```

- UCP维护引用次数
- UCO把引用计数的细节隐藏，外界不需要管什么时候销毁，只需要减，当没有人引用，自然会销毁
- StringRep只考虑实际数据存储
- UCO和UCP可重用



## Other smart pointers

- Standard library holder for raw pointers on stack
- Releases resource when destroyed (latest)

```

1 template <class X> std::auto_ptr {
2
3 public:

```

```
4
5 explicit auto_ptr(x* = 0) throw();
6
7 auto_ptr(auto_ptr&) throw();
8
9 auto_ptr& operator=(auto_ptr&) throw();
10
11 ~auto_ptr();
12
13 x& operator*() const throw();
14
15 x* operator->() const throw();
16
17 ...
18
19 };
20
21 #####
```

## 7 奇奇怪怪的小知识点

### 7.1 格式

```
1 #include <iomanip>
2 cout << setfill('0') << setw(6) << it->first << " ";
3 //可以固定长度为6，并且前面补0
```

加上fixed表示浮点数精度位数，不加为全部有效位数

### 7.2 sort

sort需要 `include <algorithm>`

可以用于数组、vector、map

可以自己编写自己的compare

```
sort(v.begin(), v.end(), cmp)
```

### 7.3 cmath

数学公式的h文件是cmath

需要 `include<cmath>`

## 7.4 stable\_sort

普遍的 `sort` 不是稳定的，相等的值可能会改变顺序

而 `stable_sort` 是稳定的，会保证相等的值，前面的值一直在前面

## 7.5 typeid

`typeid().name()`

操作符

```
1 A *ap=new B;
2 std::cout<<typeid(*ap).name()<<std::endl;
```

## 7.6 构造析构

先构造父类的类里的组合，再构造父类，再构造子类的类里的组合，再构造子类；

析构相反

## 7.7 \_\_gcd

`__gcd(_fz, _fm)`

# 7 错题集

## 7.1

- 连空格读入字符串可以用 `getline()`
- 在字符串中删去某一些段，可以使用 `.erase(pos, length)`

## 7.2

12、假定AA为一个类，a为该类公有的数据成员，x为该类的一个对象，则访问x对象中数据成员a的格式为（D）。

- A. `x(a)`    B. `x[a]`    C. `x->a`    D. `x.a`

13、假定AA为一个类，a()为该类公有的函数成员，x为该类的一个对象，则访问x对象中函数成员a()的格式为（B）。

- A. `x.a`    B. `x.a()`    C. `x->a`    D. `x->a()`

14、假定AA为一个类，a为该类公有的数据成员，px为指向该类对象的一个指针，则访问px所指对象中数据成员a的格式为（C）。

- A. `px(a)`    B. `px[a]`    C. `px->a`    D. `Px.a`

## 7.3

Resolver `::` is used to: D

- A.Define a member function outside class declaration
- B.Access a member of a namespace
- C.Access a static member of a class
- D.All of the others

B: 用using namespace std会加入很多不用的东西，最好是using std::cout;  
using std::endl;

C: 类的成员可以声明为static

```
1 class Student {
2 static int a;
3 }
4
5 int main() {
6 Student::a = 1;
7 }
```

## 7.4

2-4 分数 2

下列关于异常类的说法中，错误的是。

- A. 异常类由标准库提供，不可以自定义
- B. C++的异常处理机制具有为抛出异常前构造的所有局部对象自动调用析构函数的能力
- C. 若catch块采用异常类对象接收异常信息，则在抛出异常时将通过拷贝构造函数进行对象复制，异常处理完后才将两个异常对象进行析构，释放资源
- D. 异常类对象抛出后，catch块会用类对象引用接收它以便执行相应的处理动作