C++ object-oriented programming

旨在记录一些C++面向对象技巧

类的定义

- 提醒：
  - 23.4.10：当类的成员函数包含指针时，要么就是浅复制（只传地址），要么就是深复制：既要在构造时new出自己的空间，也在析构时delete [] p;new和delete总是成对出现

运算符重载

- 重载算术运算符+，-，etc.
  - 实质上是重载函数
  - 可以重载为
    - 普通函数：就需要n个形参
    - 成员函数：n-1个形参，另一个就是对象本身
- 赋值运算符重载
  - 只能定义为成员函数，通常用于把一个其他类型的值赋到对象上
  - 返回值为引用： `String & operator = (const char* s);`
    - 注意这里返回引用的原因是尽量保留运算符的原本特性，要求如果这个赋值语句再参与其他运算，应当指的是该对象eg.(a=b)=c
  - 只能用于赋值，不能在定义构造时使用（否则会调用复制构造）
  - String类的深拷贝浅拷贝
    - string类的成员变量为一个字符串指针，如果不定义逐渐的赋值运算，则会导致两个String指向同一处
  - PS.赋值运算符重载的时候要注意，如果涉及到指针的操作，一定要记得给指针赋值之前要确保__指针是NULL__，否则会造成内存泄漏
- 当运算符写为成员函数的形式不能满足要求eg.5+complex
  - 但又需要访问私有变量
  - 则重载为友元函数
- 一个例子：可变长数组：
  - P215
  - 注意：

        - 对于以指针为成员变量的类，其赋值运算和复制构造要小心需不需要重新建立指针、开空间
        - 重载[]时，返回值为引用（本质上是要返回那个变量，而非值）
        - 如果只是复制指针而不是新开空间的话，会在析构时把同一个地址析构两次，导致 **Seg fault**
- 流插入与流提取
    - 流插入 `cout<<` ,cout是ostream类的对象，"<<"提取运算符在这个类上重载
    - 为了使其可以连续运行 `cout<<5<<"this"` ,则应该使其返回类型为ostream &，返回值即本身 *this，类似之前赋值中的的a=b=c
    - 如果要为新的类的对象写输出，则在ostream上重载<<为普通函数，但要在类里加入友元函数，以访问私有变量

```
ostream & operator(ostream & os,const T t)//注意不需要变的引用都加上const
{
  os<<t.fx;
  return os;//记得返回os!!!!!
}
```

  只需要记得cin cout都是对象，所以要返回引用
    - 输入流是一样的，istream类，但函数体可能更复杂，需要拆解输入的字符串并拆解读值
- 类型转换与自增自减
    - 类型转换：指将该类转换为别的类，不需要写返回值类型

```
//complex to double保留实部
Complex::operater double (){return real;}
cout<<(double)c;
double a=c;
```

    - 自增自减：前置与后置
        - 前置作为一元运算符
            - 成员函数定义直接空参数，返回为引用
        - 后置作为二元运算符，多写一个没用的参数(int)
            - 但注意按照定义 后置要返回原值 而不是加过的值

继承

基本概念

- 已有类的基础上，再定义新的类，新类继承了原有类的所有成员变量和成员函数：“B拥有A的全部特点”
    - A为基类，B为派生类

- 派生类拥有基类的全部成员函数和成员变量，不论是private、protected、public
  - 但派生类不能访问基类的private成员变量
- 写法：
  - `class 派生类名：public 基类名`
- 存储：
  - 在基类构建的存储空间后加上派生类的成员变量
- 构建特点：
  - 基类与派生类不是同一层面，基类是一个概括，通常不会直接使用；派生类是一个具体的实例
- 构建方法：
  - 派生类中定义的新的函数时，可以通过A::func()来调用基类的函数
  - 注意最好不要定义与基类同名的函数，这会覆盖基类的函数
    - 问题在于，如果在使用中不小心使用基类定义的指针或引用来表示一个派生类对象，又调用了这个函数，会导致调用的是基类的函数而不是派生类的函数
    - 最好声明为virtual，这样就可以在运行时确定调用的是哪个类的函数

继承关系和复合关系

- 继承关系：B是A的一种
  - eg.人是动物的一种
  - 一个B对象也是一个A对象
- 复合关系：B有一个A
  - eg.人有一个心脏
  - 一个C对象包含一个D对象,D的一个对象是C的一个成员变量
- 使用：
  - 对逻辑上并列的两类，寻找其共同的特点，将其抽象为一个基类；用派生类声明需要的两类
  - 对于逻辑上包含的关系，用复合关系
    - 使用时，最好在小类里声明大类为友元，这样大类可以直接访问小类的私有成员变量
  - 复合关系的定义有时候逻辑上涉及循环定义
    - eg.人有一个心脏，心脏属于一个人
    - 但我们不能在没有声明完一个类的情况下就建立这个类的对象，否则其存储空间无法确定
    - 所有我们只能用指针或引用的形式来表示这种关系
      - 使用指针或引用，只需要声明类的存在即可，不需要声明类的定义

派生类覆盖基类的成员

- 如前所述，当在派生类里定义基类同名的成员，会覆盖基类的成员
  - 覆盖之意在于形式上，实际上基类的成员还是存在的，只是不能直接访问
  - 缺省的调用这些成员会取派生类的成员

- 但可以通过基类名::成员名来访问基类的成员（非private）
- 尽量不要定义与基类同名的成员变量，成员函数可以通过虚函数实现

类的保护成员

- protected成员
  - 与private类似，但派生类可以访问
  - 但在外部不能访问
  - 派生类的成员函数可以访问当前对象和其它对象的基类的保护成员

派生类的构造和析构

- 发现问题，派生类的构造函数不能直接访问基类的私有变量
  - 必须调用基类的构造函数
  - 通过派生类的构造函数的初始化列表来调用基类的构造函数
    - 派生类名（参数表）：基类名（参数表）
      - 实际上是在以类似A(x_)：x(x_)的函数形式，调用基类的构造函数
    - 注意基类的构造函数的参数表要与基类的构造函数的参数表相同
    - 如果不引入基类的构造函数，会调用基类的缺省构造函数（默认构造函数）
    - 如果基类没有缺省构造函数，会报错
  - 在此基础上再去构造派生类的成员变量
  - > 在执行一个派生类的构造函数之前，总是先执行基类的构造函数。

- 派生类的析构函数被执行时，执行完派生类的析构函数后，自动调用基类的析构函数。
- 有关顺序：

在创建派生类的对象时：
先执行基类的构造函数，用以初始化派生类对象中从基类继承的成员；
再执行成员对象类的构造函数，用以初始化派生类对象中成员对象。
最后执行派生类自己的构造函数

在派生类对象消亡时：
先执行派生类自己的析构函数
再依次执行各成员对象类的析构函数
最后执行基类的析构函数

析构函数的调用顺序与构造函数的调用顺序相反。

public继承的赋值兼容规则

- public 表示了一种继承规则

    - 这种规则表示基类的成员在派生类中的访问属性不变，即完全的继承
    - protected继承时，基类的public成员和protected成员成为派生类的protected成员。
    - private继承时，基类的public成员成为派生类的private成员，基类的protected成员成为派生类的不可访问成员。
    - protected和private继承不是“是”的关系。
- public下的赋值：
    - 派生类对象可以赋值给基类对象
    - 派生类对象可以初始化基类引用
    - 派生类对象的地址可以赋值给基类指针
- 对于private或protected的继承，则上述三条不可行
- 但是对于基类的指针或引用，可以指向派生类的对象
    - 但是只能访问基类的成员
    - 不能访问派生类的成员
    - 但是可以通过强制类型转换（B*）来转换为派生类指针，访问派生类的成员
- 当然，更不要用派生类的指针去指向基类变量

直接基类和间接基类

- 当派生类B派生出派生类C时，B是C的直接基类，A是C的间接基类

多态

- 多态是派生带来的应用方式，cpp中用虚函数实现
- 概念上，虚函数的两种表现形式
    - 通过基类声明的指针调用虚函数时，会根据指针或引用的实际对象类型来确定调用的是哪个类的虚函数
    - 通过基类声明的引用调用虚函数时，会根据引用的实际对象类型来确定调用的是哪个类的虚函数
    - 注意一定是虚函数，而不是普通函数的覆盖（会报错）
- cpp中最好用指针和引用来操作虚函数
    - > 在 C++ 中，虚函数的调用一般都需要使用指针或者引用的形式，而不能直接使用对象的形式。这是因为 C++ 中的虚函数是通过虚表来实现的，每个对象都有一个指向虚表的指针。
    - 如果你是直接用对象调用函数，那就不需要虚函数的概念，因为我们在编译时就知道你的对象是什么类型了，自然就能调用到正确的函数了。
        - 我们认为这不是多态

- 优点在于当一个函数的参数或一个需要操作的对象可能是很多种派生类时，可以直接将其写为大类的形式，并以大类的形式调用虚函数。此时当执行时，该值被确定为哪个派生类，就调用哪个派生类的函数。
    - 实际上即以大类的形式同一的构建关系，但执行时按派生类分方式执行。
    - 最方便的地方在于，对于一个类的函数，可以直接用基类的指针或引用来调用，而不用考虑是哪个派生类
        - 这就像一个可以在运行时才被确定的“变量”，给了程序更强的适应性
- 实例：用一个基类指针（！）数组储存所有派生类的变量指针，但每个变量指针都是以派生类创造的，这样对数组内值的同一操作（写法上）可以产生对应的不同的效果
    - > 用基类指针数组存放指向各种派生类对象的指针，然后遍历该数组，就能对各个派生类对象做各种操作，是很常用的做法

- 究竟调用哪个派生类的虚函数，看的是调用指针所指对象的类型，而非调用指针的类型（指针和引用的类型应该都是基类）
    - > 在 C++ 中，虚函数的动态绑定是基于对象的实际类型来进行的，而不是基于指针或引用的类型。当通过基类指针或引用调用虚函数时，编译器会根据指针或引用所指向的对象的实际类型来确定调用哪个版本的虚函数，而不是根据指针或引用本身的类型来确定。

- 虚函数表
    - 有虚函数的类的任何对象中都放着虚函数表的指针
    - 根据调用的类的类型，来确定调用的虚函数地址
- 判断是否为多态
    - 在构造函数和析构函数中调用虚函数，不是多态。编译时即可确定，调用的函数是自己的类或基类中定义的函数，不会等到运行时才决定调用自己的还是派生类的函数。
    - 换句话说，多态指的是运行时才能确定的函数调用
    - 同时，对于一个可能是虚函数的调用，通常只要其是以指针形式被调用的（包括被省略的this->）,都会产生多态
    - 同时，如果基类函数是虚函数，那么其所有的同名同参函数都是虚函数
    - 注意语法检查先于多态的运行，所有当基类的虚函数被标记为private会导致编译出错
- 虚析构函数：
    - 如果用基类的指针来删除派生类对象，就需要指定基类的析构函数为虚函数，否则只会调用基类的析构函数，而不会调用派生类的析构函数
        - 派生类的析构函数也是虚函数，但可以不用声明为virtual
    - 当析构函数被声明为虚时，则会先按虚函数规则执行对应函数，此后再执行基类的析构函数
    - 通常，含有虚函数的基类都应该使用虚析构函数
    - 注意：不允许以虚函数作为构造函数

纯虚函数和抽象类

- 抽象类中存在纯虚函数
  - 用来定义接口，没有函数体
  - 用来声明一个抽象类用作基类
  - 用来规范派生类的行为
  - 不能被实例化！！！
  - 派生类必须实现纯虚函数
  - 用法：在函数声明后加=0
- 在抽象类的成员函数内可以调用纯虚函数
  - 如前所述，此时的调用是基于this指针，会在运行时被指派为派生类里实现的函数
  - 但是在构造函数和析构函数中不能调用纯虚函数，如前所述，这不是多态

输入输出

- 相关类
  - iostream:标准输入输出流
  - fstream：文件
- iostream
  - istream
    - cin:用while(cin>>x)，此时cin作为是一个流对象，支持 bool 类型转换运算符，可以将其转换为 bool 值。从而没有数据可读的时候，就返回false，结束循环
      - 更严格的说：

        > 当输入流遇到文件末尾或错误时，会将流状态置为 eofbit 或 failbit。在布尔上下文中，istream 对象被隐式转换为 false。因此，在输入操作返回 istream 对象时，检查输入流的状态即可判断输入操作是否成功。
        > 因此我们可以用

  - istream类的成员函数
    - istream & getline(char * buf, int bufSize,char delim);
      - 从输入流读写到缓冲区buf，到delim结束（缺省则为'\n'），delim会被读，但不会被写入buf，数据结尾添\0。
      - 但最多读bufsize-1个，达到或超过都会导致之后的读入错误
      - 可以用 if(!cin.getline(…)) 判断输入是否结束,与上文类似
    - bool eof()判断输入流是否结束

```
while (true) {
  int x;
  cin >> x;
  if (cin.eof()) {
      break; // 输入流已经结束，退出循环
  }
  // 处理读入的数据
}
```

        - int peek(); 返回下一个字符,但不从流中去掉.如果输入流已经结束，则返回 EOF（即 -1）。
            - 如果想要其从流中去掉，用cin.get()
            - 这两者方式让我们更好的控制如何读输入
        - istream & putback(char c); 将字符ch放回输入流
        - istream & ignore(int nCount = 1, int delim = EOF);从流中删掉最多nCount个字符，遇到EOF时结束。
    - 重定向
        - ```
          freopen("t.txt","r",stdin); //cin被改为从 t.txt中读取数据
          freopen("test.txt","w",stdout); //将标准输出重定向到 test.txt文件
          ```
        - 其他正常写即可
    - 补充：
        - cerr对应于标准错误输出流，用于向屏幕输出出错信息
        - clog对应于标准错误输出流，用于向屏幕输出出错信息
        - cerr和clog的区别在于cerr不使用缓冲区,直接向显示器输出信息；而输出到clog中的信息先会被存放在缓冲区,缓冲区满或者刷新时才输出到屏幕
        - cout对应于标准输出流，用于向屏幕输出信息
        - cin对应于标准输入流，用于从键盘获取用户输入
- 流操纵算子
    - #include
    - 整数流的基数（进制）：流操纵算子dec,oct,hex
    - 浮点数精度：
        - precision是成员函数，其调用方式为：cout.precision(5);
        - setprecision 是流操作算子，其调用方式为：cout << setprecision(5); // 可以连续输出
        - 默认是n位有效数字（非定点）
            - 加入setiosflags(ios::fixed)，小数点固定，变为小数点后n位
            - 再加入resetiosflags(ios::fixed)，则取消固定
        - 只对浮点数，对整型不影响
    - 设置域宽
        - `cin >> setw(n) or cin.width(n);` cout类似
          ```

        - 宽度设置有效性是一次性的，在每次读入和输出之前都要设置宽度。
    - 默认靠右，左侧填充，可以在 `setw<<left` 等控制
    - 可以选择setfill('*')填充物
- 自定义流操作算子
    - ```
      ostream & tab(ostream & output){
        return output << '\t';
      }
      cout << “aa” << tab << “bb” << endl;
      ```
    - 原因是<<重载了这样的参数

      ```
      ostream & operator<<( ostream & ( * p ) ( ostream & ) ) ;
      ```
- 文件
    - #include
    - ofstream

      ```
      ofstream fout;
      fout.open("test.out",ios::out|ios::binary);
      ```
    - ios::out是删除重写，ios::app是续写
    - 读写操作是基于读写指针的，用过tellp()获取，通过seekp()移动
    - 文件流是和标准流一样的，可以用<<,>>,也可以加成员函数和操作算子
        - 文件专有的，read 和 write

模板

函数模板

- 函数模板的定义格式如下：

  ```
  template <typename T>
  T func_name(T arg1, T arg2, ...)
  {
      // 函数体
  }
  ```
- 当然也可以有多于一个类型参数
  ```

```
template <class T1, class T2>
T2 print(T1 arg1, T2 arg2)
{
  cout<< arg1 << " "<< arg2<<endl;
  return arg2;
}
```

- 也可以主动实例化模板，主动决定类型

```
template<typename T>
T Inc(T n)
{
    return 1 + n;
}

int main()
{
    cout << Inc<double>(4)/2.0; // 输出 2.5
    return 0;
}
```

- 函数模板可以重载，只要它们的形参表或类型参数表不同即可
- 选择函数调用的顺序
  1. 先找参数完全匹配的普通函数(非由模板实例化而得的函数)。
  2. 再找参数完全匹配的模板函数。
  3. 再找实参数经过自动类型转换后能够匹配的普通函数。
  4. 上面的都找不到，则报错。
  - 匹配模板函数时，不进行类型自动转换
- 有时候类型参数表的意义是开放的，比如可以用一个类型参数来标记一个参数为函数指针

```
template <typename T, typename F>
T Sum(T a[], int n, F op)
{
    T s = 0;
    for (int i = 0; i < n; ++i)
        s = s + op(a[i]);
    return s;
}
```

调用时

```
int a[] = {1, 2, 3, 4, 5};
cout << Sum(a, 5, [](int x) { return x * x; }); // 输出 55
cout << Sum(a, 5, [](int x) { return x * x * x; }); // 输出 225
// 上面使用了lambda表达式，也可以使用普通函数的函数名作为函数指针
```

多说一句，函数指针的实例化类型表达式是这样的： 返回值类型（*函数指针名)(类型参数表）其中*代表了这是一个指针，而括号是必须的，否则就变成了返回值是函数指针的函数了

类模板

- 与函数模板类似，类模板的定义格式也是类型参数表+类定义
  - 成员函数如果使用函数模板的定义，无论是在内部还是外部，都需要写类型参数表！
    - 因为很多时候，成员函数模板只是部分使用类模板的类型参数，还需要新的类型
  - 必须主动声明类型；实例化类的时候：类模板名 <真实类型参数表> 对象名(构造函数实参表);
  - 注意我们实例化出来的是一个类，而不是一个对象；对象需要根据实例化出来的类来创建
- 类模板的类型参数表里可以出现非类型参数，比如一个需要给出的数组长度int size
- 类模板与派生
  - 从类模板派生类模板

```
template <typename T>
class DerivedClass : public BaseClass<T> {
  // 派生类的成员和特性
};
```

  - 实例化时，逻辑上可以理解为，我把派生类实例化了，从而确定了T，从而再实例化基类
  - 其具体形式是同质的，核心思想是从叶向根实例化，先实例化派生类，再实例化基类
    - 注意对于类型参数的复用，如果在派生的定义中是同名的，那就构成复用，他们指代的同一个类型
- 类模板与友元
  - 在模板中定义的友元函数，那所有实例化的对象都有这些友元函数
  - 友元函数也可以以函数模板形式存在
  - 友元类也可以以类模板的形式存在
  - 总之，实例化的过程是先实例化类，再实例化友元
- 类模板也可以包含static变量，每一个被实例化的类都有自己的那个static变量

string类

- 初始化
  -

```
string s1; // 默认初始化，s1是空串
string s2(s1); // s2是s1的副本
string s3("value"); // s3是字面值"value"的副本，除了字面值最后的那个空字符外
string s4(n, 'c'); // 把s4初始化为由连续n个字符c组成的串
string s5 = "value"; // 等价于s3
string s6 = s1; // 等价于s2
```

    - 初始化不能用字符，但可以用字符来赋值
- 赋值，连接，运算
    - 支持getline(cin, s)来读取一行
    - =赋值；assign赋值，assign可以用来赋值子串
    - 用at来读单个字符（会检查范围），用[]来读单个字符，用substr来读子串
    - string的大小比较是字典序
        - compare可以比较子串
        - compare函数返回-1，0，1
        - 注意string的函数中，表达子串都是起始位置和长度，而不是起始位置和终止位置
    - 连接用+，+=
    - swap交换两个string的内容
    - find函数，返回第一个匹配的子串的起始位置，如果没找到，返回string::npos
        - find函数有多个重载，可以指定起始位置，可以指定子串的起始位置和长度
        - 还有很多衍生函数，比如rfind，find_first_of，find_first_not_of等等
    - erase可以删除子串，也可以删除单个字符
    - replace可以替换子串
    - insert可以插入子串
    - c_str函数返回一个C风格的字符串，即以空字符结尾的字符数组

STL

- STL是标准模板库，包含了很多类模板和函数模板
- 概念
    - 容器：用来存放数据的类模板，比如vector，list，set，map等等
    - 迭代器：用来遍历容器的类模板，比如vector::iterator，作用类似指针
    - 算法：用来操作容器的函数模板，比如sort，find，copy等等
- 容器概述
    - 顺序容器：vector，deque，list，forward_list，array
    - 关联容器：set，map，multiset，multimap
    - 无序容器：unordered_set，unordered_map，unordered_multiset，unordered_multimap
    - 容器适配器：stack，queue，priority_queue
    - 顺序容器和关联容器都是线性结构，无序容器是哈希表，容器适配器是基于顺序容器的封装
    - 对于想放入有序的容器的类，至少需要重载==和<运算符，以使得函数可以使用
- 顺序容器：略
- 关联容器：
    - 元素是排序的，且不能重复（加multi可以重复）
    - 非常便于查找，红黑树实现，O logn

    - set是集合
    - map是键值对pair
- 迭代器
    - 容器类名::iterator 变量名;
        - 容器类名::const_iterator 变量名;
    -
        - 迭代器变量名来访问元素
    - 分为双向迭代器和随机访问迭代器
        - 双向迭代器只能访问，赋值或判断等，或++进行1步的移动
        - 随机访问迭代器可以进行+-n的移动，可以比较大小，求差值
            - vector和deque的迭代器是随机访问迭代器
            - 关联容器和list都是双向迭代器
            - 容器适配器不支持迭代器
        - 我们可以根据迭代器的种类来判断是否可以使用某个函数（算法）
- 算法
    - 算法存在于algorithm头文件中，是一些独立的函数模板，但可以应用在不同的容器上（实际上是基于容器的迭代器来工作）
    - find函数，返回第一个匹配的元素的迭代器，如果没找到，返回last
        - p = find(v.begin(),v.end(),3);
    - 在STL中
        - 大小是"<"定义的
        - 相等有时候是==，用于无序序列；有时候是双向"<"同时为假，用于有序序列；因此我们应该保证有序序列的元素是全序的
- 各个容器详解
    - vector & deque
    - list:
        - list不支持stl的sort，但有成员函数sort
- 函数对象
    - > 是个对象，但是用起来看上去象函数调用，实际上也执行了函数调用。
    - 回想，函数的写法是()运算符重载，所以函数对象的写法也是()运算符重载
    - 函数对象的好处是可以保存状态，比如计数器
    - 另外函数对象比指针看起来要实在一点，也方便作为参数传到模板里
    - eg. sort(v.begin(), v.end(), greater());
        - sort的第三个参数是函数对象，greater()是一个临时对象，它的()运算符重载了，可以用来比较两个int
        - op为函数对象，op(a,b)返回true表示a应该在b前面，（默认是小于号，升序排序）
            - 当自定义了op后，所有的比较都会用op来比较，op代替了小于号
        - 那么改为greater就变为降序

    - 可用的函数对象类模板：
        - equal_to
        - greater
        - less
- 方便输出的ostream_iterator
    - copy
- 关联容器
    - set和multiset
        - 可以给出比较函数作为第二个类型参数，比如greater，这样就可以按照降序排列
        - 当你定义了自己的比较函数后，所有的比较都会用这个函数来比较，就不一定需要重载小于号了
        - set::insert返回一个pair，pair的first是一个迭代器，指向插入的元素，second是一个bool，表示是否插入成功
    - map和multimap
        - map的元素是键值对，pair，pair的first是键，second是值
        - 输入的第一个参数类型是键，第二是值，第三个是键的比较函数
        - 可以用类名::value_type()来生成一个pair，或make_pair
        - map中用[]来访问元素，如果不存在，会自动插入一个默认值
            - 因此可以用pairs[key]来统一的添加或修改元素
        - 而multimap 中允许多个元素的关键字相同，因此上方法不好用，也米有重载[]，只能用insert

C++11

- {}初始化器，来导入初始化参数
- 成员变量可以在类内初始化
- auto关键字，自动推导类型
    - 可以用于一些迭代器的声明，避免写出太长的类型
    - 但不能用于函数参数，因为函数参数必须有明确的类型
- 函数尾部声明
    - 结合auto关键字，函数的返回值类型可以在函数体后面声明，并使用decltype（）来推导该类型
- 智能指针shared_ptr
    - shared_ptr ptr(new T); // T 可以是 int ,char, 类名等各种类型
    - 从而实现对指针的托管，当ptr消失时，会自动释放内存
    - shared_ptr对象不能托管指向动态分配的数组的指针
- 右值引用&&，move语义

    - 右值引用是一个新的引用类型，它是一个必须绑定到右值的引用
    - 传统的引用必须绑定到左值，左值是一个有名字的变量，在内存中有确定的存储位置
    - 右值引用的意义在于，如果一个函数的返回值是一个对象，而这个对象没有被命名就直接被输出了，那就需要用右值引用来表示
- 无序容器与哈希表
    - 用于只需要快速查找
- lambda表达式
    - 捕获列表 mutable(可选) 异常属性 -> 返回类型 {函数体}
    - 捕获列表指以何种方式传入参数变量
        - [] 空捕获列表
        - [&] 引用捕获列表
        - [=] 值捕获列表
        - [a,&b] 混合捕获列表
        - [this] 以值的方式捕获this指针