《C++ Primer》读书笔记

整理日期：20120617

主页：http://chuanqi.name

E-mail：chuanqi.tan@gmail.com

C++最佳入门书，毫无疑问；Lippman的大师级经典之作！

By 谭川奇

前 言

最经典的C++入门书，几乎人手一册吧，Lippman大师的文笔正如他的C++技术水平一样出色。正是这本书教会了我用C++写Hello World，因此我对这本书是很有感情的，也很庆幸当时在书店里买下了它。如果你刚接触C++，那么你可能也真的很需要认真的读完这本书。简单整理了一下初学C++时做过的一些笔记，并把以前犯的一些错误标示出来了，我当时犯的错误也很可能是其他初学者容易犯错的地方吧。

虽然里面难免有很多低级错误，但还是希望本笔记能为你的学习提供一些帮助！

谭川奇 2012.06.17 于北京
chuanqi.tan@gmail.com

本文采用的约定

蓝色：概念、讨论主题

下划线：值得注意的内容

紫红色：比较重要的地方

红色：必须理解的地方

绿色：一些评论、注解

斜体：一些总结性的话

粗体：对上述所有标记的加强、警示作用

蓝色背景：对以前的错误理解的纠正和一些追加解释

目 录

第一部分：基本语言

第一章：快速入门

1. C++中，每一个表达式都会产生一个结果，通常是将操作符作用到其操作数所产生的值。当操作符是输出操作符时，结果是左操作数的值。也就是说，输出操作返回的值是输出流本身。所以可以写cout << x << y;
2. 在C++程序中，大部分出现空格符的地方可以用换行符代替。这条规则的一个例外　是字符串字面值中的空格符不能用换行符代替。别一个例外是空格符不允许出现在预处理指示中。
3. 当使用istream对象作为条件，结果就是测试流的状态。如果流是有效的（也就是说，如果读入下一个输入是可能的）那么测试成功。如果遇到文件结束符或无效输入时，则istream对象是无效的，处于无效状态的istream对象将导致条件失败。

第二章：变量和基本类型

1. 一般来说，short为半字机器字长，int为一个机器字长，而long为二个机器字长。（在32位机器中，int类型和long类型通常字长是相同的）
2. 对于要用浮点数的情况，几乎总是用double，用long double的情况太少。而float几乎从不使用。
3. 使用整形字面值时，以0开头表达是8进制，而以0x开头则表示是16进制。字面值后面可以加L和U分别表示是long类型和unsigned类型。
4. 为了兼容C语言，C++中所有的字符串字面值都由编译器自动在未必添加一个空字符。字符串字面值“A”表示包含字母A和空字符共二个字符的字符串。
5. 左值和右值：（变量是左值，而字面值是右值）
   i. 左值：（lvalue）左值可以出现在赋值语句的左边或者右边。
   ii. 右值：（rvalue）右值只能出现在赋值 语句的右边，不能出现在左边。
6. C++程序员经常随意的使用术语“对象”。一般而言，对象就是内存中具有类型的区域，说得更具体一点，就是计算左值表达式就会产生对象。
7. 变量命名统一使用这样的编程风格：hello_word，而不使用HelloWord或者helloWord之类。
8. 直接初始化和复制初始化：

   ```
   int i(1024);                    //直接初始化
   int i = 1024;                   //复制初始化
   ```

   ~~直接初始化更灵活而效率更高。复制初始化相当于先创建变量并赋予初值然后再擦除当前值并用新值代替。~~

   ~~看起来在VC++中这两种写法被优化成了完全一样的，但是C++ Primer反复强调“直接初始化”语法灵活且效率更高！~~

~~但是，至少在VC++中这两种写法完全看不到区别，理论上的差距应该是被编译器优化掉了！~~

更正：这2种写法对于内置类型来说可能没有区别，但是对于自定义类型来说，这是显式调用构造函数和隐式调用构造函数的区别！

9. 变量未初始化的情况：内置类型是否自动初始化取决于变量的位置，在函数体外定义的变量都初始化为0，在函数体内定义的变量不进行自动初始化；对于类类型来说，无论变量在那里定义，默认的构造函数都会被调用。

   ~~变量初始化的情况可以总结为一句话：内置类型的局部变量不会自动初始化，自定义类型一定会自动初始化！~~

   更正：正确的答案是“分配在全局内存空间的变量才会保证进行清0操作，其它地方分配的内存不保证清0操作。”

10. 与其它变量不同（这是因为非const变量默认为extern的），除非特别说明，const对象默认为文件的局部变量。此变量只存在于那个文件，而不能被其它文件访问。通过指定const变量为extern，就可以在整个程序中访问const对象了。

11. const引用是指指向const对象的引用。只有const引用可以绑定到右值（比如字面值）。

    ```
    int i = 42;
    const int &r = 42;
    const int &r2 = r + i;          //在此如果用非const引用将引发编译错误
    ```

    非const引用只能绑定到与该引用（严格）同类型的对象。（这是由于将引用绑定到不同类型时将隐式产生一个临时变量将原类型转换为现类型，所以会产生无法修改原类型的情况）

    const引用则可以绑定到不同但相关的类型的对象或绑定到右值。

12. 引用：（最重要的就是理解：引用仅仅只是对象的别名）。引用是一种复合类型，只能而且必须在初始化的时候赋值（绑定对象），不可修改！

13. 枚举：默认时第一个枚举成员为0，后面的每个枚举成员比前面大1。枚举成员的值只能在初始化时修改，因为枚举成员本身就是一个常量表达式。

14. 定义变量和定义数据成员存在非常重要的区别：一般不能把类成员　的初始化作为其定义的一部分。当定义数据成员时，只能指定该数据成员　的名字和类型。类应该在构造函数中才对数据成员进行初始化。

15. 用class和struct关键字定义类的唯一差别就是默认访问级别：默认情况下，struct的成员为public，而class的成员为private。

    (不能算是唯一的区别，还有默认的继承属性也有区别)

16. 头文件用于声明而不用于定义（定义部分应该在源文件中来做），这是设计头文件的很重要的规则。头于头文件不应该含有定义这一规则，有三个例外。头文件可以定义类，值在编译时就已经知道的const对象和inline函数。

17. 用头文件和源文件分开类的声明和定义的正确的方式：

    myClass.h　中包含类的声明部分，　myClass.cpp包含类的定义部分。　在myClass.cpp中需要用#include "myClass.h"来包含myClass的声明。在需要使用myClass的其它文件中只需要包含#include "myClass.h"即可！

18. 避免头文件的多重包含：头文件一定要引用预处理来避免多重包含。

```
#ifndef PATH_OF_THIS_HEAD_FILE_FILENAME_H
#define PATH_OF_THIS_HEAD_FILE_FILENAME_H
int i = 10;
#endif
```

编写头文件保护既方便又容易，而且是必要的。

19. C/C++的头文件的处理机制是：对于如 #include <iostream> 编译器会将iostream文件的内容全部复制过来替代这一行语句。
20. 有一种情况下，必须总是使用完全限定的标准库名字：在头文件中。理由是头文件的内容会被预处理器复制到程序中，可能有些程序不想使用using namespace std之类的。头文件中应该只定义确实必要的东西！

第三章：标准库类型

1. const_iterator对象和const的iterator对象的区别：
   const_iterator对象是可以改变的，例如可以用自增和自减操作，只是不能通过这个迭代器的解引用来改变原对象；而const的iterator对象一经初始化就不能改变，不能自增也不能自减，所以也几乎是没有用的。
2. 任何改变vector长度的操作都会使已存在的迭代器失效。因为改变vector长度的操作完全有可能引起vector的动态增长。
3. 应该尽量使用标准库类型的一些配套类型，通过使用这些配套类型，库类型的使用就能实现与机器无关。
   比如 string::size_type（不要把string.size()返回值赋给一个int类型）, vector<int>::size_type等。
4. 当进行string对象和字符串字面值混合连接操作时，+操作符的左右操作数必须至少有一个是string类型。比如： string s4 = "hello" + "world";将会产生错误。
5. 字符串转换函数：
   - double atof (const char *nPtr);
   - long atoi (const char *nPtr);
   - long atol (const char *nPtr);
   - double strtod (const char *nPtr, char ** endPtr);
   - long strtol (const char *nPtr, char ** endPtr, int base);
6. string对象和bitset对象之间是反向转化的，当用string对象初始化bitset对象时，记住这一差别很重要。 string ”1110000” ⇔ bitset<10> ”0001110000” 这更是符合我们的直观感觉的。

第四章：数组和指针

1. 声明指针有二种风格：int* a;和int *a;第二种风格（*靠近标识符）更清楚明了，应该使用第二种风格。第一种风格很容易让人误解为int*是一种类型，但实际上它与第二种完全等价。
2. 在使用任何变量之前都应该先初始化，尤其是指针。
3. void*指针，它可以保存任何类型的对象的地址。void*表明该指针与一地址值相关，但并不清楚存储在此地址上的对象的类型。而且不允许使用void*指针操纵它所指向的对象。
4. 指针和引用的两个区别：
   - 引用总是指向某个对象：定义引用时没有初始化是错误的。引用一经初始化，就始终指向同一个特定的对象。没有为空的引用！
   - 赋值行为的差异：给引用赋值修改的是该引用所关联的对象的值，而并不是使引用与另一个对象相关联；给指针赋值就是使该指针与另一个对象相关联。
5. 指向const对象的指针和const指针 （理解指针的定义时切记从右到左的读）
   - const int *p = &a;这里const是修饰int的，即是指向const对象的指针，p本身的值是可以修改的，但是不能通过*p来修改p所指向的对象的值。
   - int *const p = &a;这里const是修饰“*”的，即是const指针，这个指针是不能改变的，始终指向变量a的地址，但是可以通过*p来修改a的值的。
   - **更好的理解办法：**const不要写在最前面，于是就只有int const *p和int *const p;两种写法，记住const永远修饰它前面的那个符号，就OK了。
6. C++中提供的普通关系操作符可以用于比较指向C风格的字符串，但是实际效果却很不同：实际上，此时比较的是指针上存放的地址值，而并非它们所指向的字符串。所以应该使用<cstring>指代的strcmp函数。但最好就是不要使用C风格的字符串。
7. 动态分配的数组必须显式的释放，否则将发生内存泄露。动态分配的对象是在程序运行的堆（或者自由存储区）里的。
8. 动态分配数组时，如果数组元素具有类类型，将使用该类的默认构造函数实现初始化；如果数组元素是内置类型，则无初始化。但是可以在数组长度后面用一对空圆括号显式要求进行初始化。
   int *p = new int[10];　　//这样是没有初始化的，　可以写成 int *p = new int[10]();来显式初始化为0
9. 释放动态分配的数组空间时应该这样写：delete [] pia;这个“[]”不可忽略。
10. 严格的说，C++中没有多维数组，所谓的多级数组其实就是数组的数组。
    int (* ip)[4]; 这个ip是定义一个指向int[4]型数组的指针，而不是定义一个指向int类型的指针的数组。（从右向左理解就比较容易了，从里向外）
    对比：int *ip[4] 定义了4个指向int类型的指针的数组。

第五章：表达式

1. 短路求值：&&和||操作时，只有当其左操作数无法确定整个逻辑表达式的值时，才会去计算右操作数。

2. 赋值操作具有右结合的性质，而且赋值操作返回的是左值。

   a = b = 0; 的执行顺序是 b = 0; 返回b，然后a = b;

3. 只有在必要的时候才使用后置操作符（后自增，后自减）。因为前置操作符的性能比后置操作符更好，效率更高。

   后置操作符的工作原理：*iter++相当于*(iter++)，先把iter加1，然后返回iter原始值的一个副本。这样做的工作当然比直接加1要更多了，所以前置操作符的效率高。

   这是指在没有编译器优化的情况下，实际上这样的小技巧现代编译器都可以轻松的优化掉。

4. C++中，只规定了操作数计算顺序的操作符有条件（？：）和逗号操作符（从左到右）。除此之外，其它操作符并未指定其操作数的求值顺序。

   f1() * f2(); 这里无法得知是f1()被先调用还是f2()被先调用。

5. 注意以下几个声明的区别：

   - int *pi = new int;　　//未初始化 （在VC++中，这两种情况又被优化成一样了）
   - int *pi = new int();　　//已初始化（在VC++中，这两种情况又被优化成一样了）
   - int x();　　//这是定义一个函数的声明,应该这样写 int x(0);

6. **C++保证：删除0值的指针是安全的。**一旦删除了指针所指向的对象，应立即将指针置0，清楚的表明指针不再指向任何对象，防止悬垂指针。（这是一条铁律，应该时时刻刻的记住并严格执行）

7. 强制类型转换：（本质上，强制类型转换都是危险的）

   - dynamic_cast：支持运行时识别指针或引用所指向的对象。
   - const_cast：转换掉表达式的const性质。 char *pc = const_cast<char *>(pconst_char);
   - static_cast：任何隐式类型转换都可以通过static_cast来显式的执行。如从double显示声明要转换到到int，编译器甚至不会产生任何警告信息。还可以从void*指针中找回类型，甚至可以从父类的指针转换为子类的指针，但这是比较危险的。

     大多数必须使用强制类型转换的时候都建议使用static_cast来进行，相对其它的转换来说这个比较安全。
   - reinterpret_cast：相当于旧式的强制类型转换，可以转换为任何类型，相当危险！
   - 应该完全抛弃旧式的强制类型转换，尽量不要用reinterpret_cast。

第六章：语句

1. 语句的几条特征：

   - 在求解表达式中定义的变量始终存在于整个switch语句中，在switch结构外就不再有效了。
   - 对于switch结构，只能在它的最后一个case标号或default标号后面定义变量，这是为了防止变量的未定义就先使用。要定义局部使用的变量，就用{}括起来使用语句块。
   - 在循环条件中定义的变量在每次循环里都要经历创建和撤销的过程。

- goto语句和获得所转移的控制权的语句必须在同一个函数内。

2. C++中的异常：似乎C++本身并不会抛出任何异常，所有的异常都需要自己手动的抛出。然后在catch中可以捕获任何类型，抛出的类型并不一定要继承自exception。在发生了不能捕获的异常时，系统将调用terminate终止程序的执行。

   但是如果这样的异常是自己抛出的，就意味着自己知道这个异常了，这样的异常处理还有意义吗？这些与C#中是完全不同，完全不如C#中做的好。

   还是有很大意义的，即使知道了错误，也并不一定能处理。比如自己写一个类库，错误是由调用者引起的，而且我还不能默默的处理掉，必须通知类库的用户，这时异常就是最好的办法。对于大规模程序来说，这样的异常是非常有用的。C#系统抛出的异常也都可以自己检测出来，只是C#常见的异常是由系统来抛出，用起来更方便，这两种异常并没有能力上的不同。

   异常当然是非常有用的，当时说异常没什么用只能说明当时完全没有做过什么大的程序，不知道这种机制的真正用处。

3. assert（断言）预处理宏：在NDEBUG未定义时，求解条件表达式，如果表达式的结果为false，则中止程序并报错。assert(3<2);　断言常用来测试“不可能发生”的条件。需要包含<cassert>头文件。

第七章：函数

1. 明确区分引用类型和指针的区别：传指针实际上也是非引用类型，只不过其复制的是地址信息，所以可以通过地址信息找到原对象进行修改。而引用类型完全不同于非引用类型，仅仅是两个标识符指向同一个对象，这两个标识符使用完全相同。所以说很多时候引用比指针更好用。

   如果使用引用形参的唯一目的是为 了避免复制，则应当将形参定义为const引用。

2. 可以将字面值绑定给一个const引用，但不能将绑定给非const引用。所以对于形参中不需要修改的值，都应该设定为const，这样能扩大函数的使用范围。

3. 编译器会忽略为任何数组形参指定的长度：

   void print(int *) 等价于 void print(int []) 等价于 void print(int [10])

   多维数组作为形参时，编译器忽略第一维的长度。

   通过引用传递数组的办法： void print(int (&arr)[10]); //从里向外，从右向左的理解

   这是编译器在传值时的decay！

4. 含有可变形参 void foo(parm_list, ...); 或者 void foo(...);

5. 千万不能返回局部变量的引用，这样的行为是未定义的。

6. 返回引用的函数返回的是一个左值，所以甚至可以给函数的返回值赋值。

```
char &get_val (string &str, string::size_type ix) { return str[ix]; }
int main(){
   string s(“a value”);
   get_val(s,0) = ‘A’;     //这是可行的，这样会修改s的值。s = “A value”
}
```

7. 变量与函数应该在头文件中声明，在源文件中定义。　定义函数的源文件应该包含声明该函数的头文件。虽然不包括也是可以编译的，但是还是应当包含一下（包含相应的头文件大有用处啊）。
8. 如果一个形参具有默认实参，那么，它后面所有的形参都必须有默认实参。
   既可以在函数声明也可以在函数定义中指定默认实参。但是，在一个文件中，只能为一个形参指定默认实参一次。通常在声明中指定默认形参。
   - 默认参数只可在函数声明中设定一次。只有在无函数声明时，才可以在函数定义中设定。
   - 默认参数定义的顺序为自右到左。即如果一个参数设定了缺省值时，其右边的参数都要有缺省值。
   - 默认参数调用时，则遵循参数调用顺序，自左到右逐个调用。这一点要与第（2）分清楚，不要混淆。
   - 默认值可以是全局变量、全局常量，甚至是一个函数。但不可以是局部变量。因为默认参数的调用是在编译时确定的，而局部变量位置与默认值在编译时无法确定。
9. static局部对象：确保不迟于在程序执行流程第一次经过该对象的定义语句时进行初始化。这种对象一旦被创建，在程序结束之前都不会被撤销。
10. 内联一些短小的，常用的函数能在提高程序的可读性的同时又不影响效率。
    inline bool check(my_class c1, my_class c2) { ... }
    内联函数应该在头文件中定义，这点不同于其它的函数。
11. 编译器隐式的将在类内定义的成员函数当作内联函数。
    所以在C++中，对于一些短小的非递归的成员函数应该在类的内部定义。对于递归或者非常复杂庞大的成员函数应该在类内声明，在类的外部定义。
    实际上，将所有的函数放在一起来实现会造成思路的混乱。
12. 常量成员函数：bool Sales::same (const Sales *const this, const Sales &rhs) **const** { ... }
    这个const声明将成员函数定义为常量成员函数，保证不修改调用该函数的对象。
    const对象，指向const对象的指针或引用都只能调用其const成员函数，如果尝试调用非const成员函数，则是错误的。所以，应该把所有不修改调用对象的函数都定义为const函数以扩大函数的使用范围。这点同函数的参数应该尽量设置为const是一个道理的。
13. C++中函数重载的作用域：编译器从内向外开始寻找同名函数，一旦找到同名函数就不会再继续检查这个名字是否在外层作用域中存在，余下的工作就是检查该名字的使用是否有效。
    所以说：C++中，**名字查找发生在类型检查之前！**而且编译器是先查找名字，再确定名字的使用是否合法。（这个应该极少会碰到，但是如果碰到了不知道的话会非常恶心）
    每一个版本的重载函数都应该在同一个作用域中声明。
14. 即使是类类型，作为函数的参数时若未指定为引用，也是和值类型一样创建一个副本传递给形参的。应当注意。(这里其实是调用复制构造函数，如果有指针类型，则只会复制指针的地址值)
15. 指向函数的指针：（阅读函数指针声明的最佳方法是从声明的名字开始**由里向外**理解）
    bool (*pf) (int &i, int &j);　//声明指向返回值为bool形参为 (int i, int j)的函数的指针
    typedef bool (*cmpFun) (int &i, int &j)　//来简化定义，使用cmpFun f1;来定义即可

16. 指向重载函数的指针：指向重载函数的指针必须与重载函数的一个版本精确匹配，否则编译错误，连可以隐式转换的类型也不行。

第八章：标准IO库

1. iostream 继承层次：

2. unitbuf操作符：这个操作符在每次执行完写操作后都刷新流：

   cout << unitbuf << “first” << “second” << nounitbuf;

3. 文件操作：
   - fstream my_file(“d:\\1.txt”); 相当于定义并打开了一个fstream对象。
   - IO标准库使用C风格字符串作为文件名，所以要用string.c_str();来获得string对象的C风格字符串。
   - 要把fstream与不同的文件相关联，就必须先关闭现在的文件，再打开另一文件。
   - 调用clear后，就好像重新创建了该对象一样。
   - 文件模式：（模式是文件的属性而不是流的属性）
     a. in：打开文件做读操作
     b. out：打开文件做写操作
     c. app：在每次写之前找到文件尾。（是每次写）
     d. ate：打开文件后立即将文件定位在文件尾
     e. trunc：打开文件时清空已经存在的文件流
     f. binary：以二进制模式进行IO操作
   - 默认情况下，fstream以in|out的模式打开文件，不会清空文件内容。
4. stringstream对象常用于多种数据类型之间的自动格式化和转换。

   ss << “i:= “ << i << “ j:= “ << j; //ss=”i:=1 j:=3”

   ss >> str >> i >> str >> j; //i = 1, j = 3

5. 一般情况下，使用输入操作符读入string时,空白符将会忽略。

使用getline(cin, str);

第二部分：容器和算法

第九章：顺序容器

1. 静态数据成员必须在类模块和用户模块之外进行初始化。
2. 一般的，二元运算符选用友元函数重载，而一元运算符选用成员函数重载，但赋值运算符“=”应选用成员函数重载。原因如下：
   - 对于二元运算符，如“+”运算，2+op和op+2都应该是可行的。如果采用成员函数重载，将只有op+2能运行，因为op+2相当于op.operator+(2)，当然是可行的；而2+op就相当于2.operator+(op)这将产生编译错误。
     若选用友元函数重载，这二种方式都能运行，2将会被转换为op类型。
   - 对于赋值运算符，op=2是正确的，2=op却应该是错误的。但如果采用友元函数重载，则2=op也是合乎语法的，所以赋值运算符应该选用成员函数重载。
   - 当操作符定义为非成员函数时，通常必须将它们设置为所操作类的友元，因为这些操作符重载函数需要访问类的私有类型。
   - 有一条有用的规律：对称的操作符，常定义为友元重载。其它的时候多定义为成员重载。
3. 引用类型不能作为容器的类型，除了引用类型以外，所有的内置或复合类型都可用做容器的类型。
4. 定义容器的容器：vector< vector<string> > lines; 后面的两个“>”必须用空格隔开，否则系统会认为这是一个“>>”操作符。
   C++11中已经解决这个小BUG，GCC使用参数-std=c++0x
5. 对容器进行非const操作都有可能使所有的迭代器失效。意味着：只要进行了非const操作，那么所有的迭代器的值都将变得不可信任。
6. 如果两个容器都不是对方的子序列，则它们的比较结果取决于所比较的第一个不相等的元素。这点和字符串的比较一样的！

第十章：关联容器

1. 在使用关联容器时，必须有一个相关的比较函数，默认情况下，标准库使用键类型定义的<操作符的比较。所用的比较函数必须在键类型上定义严格弱排序。对于键类型，唯一的约束就是必须支持<操作符。
2. 对于两个键，如果它们相互之间都不存在“小于”关系，则容器将之视为相同的键。用作map对象的键时，可使用任意一个键值来访问相应的元素。
   注意“相等”与“等价”之间的区别！
3. 用下标访问不存在的元素时将导致在map容器中添加一个元素，它的键即为该下标值。下标操作将返回左值。所以应该切记下标操作的这个副作用，如果没有必要，不要这样用下标操作符。

4. 对于键容器，无论是什么样的顺序插入的，在容器中总是以“<”操作符排好序的，可以使用迭代器进行顺序遍历。这是个相当好的性质，可以免去排序操作，甚至可以用关联容器来进行排序。同样，如果在multimap和multiset容器中，如果某个键对应多个实例，则这些实例在容器中将相邻存放。

   因为关联窗器的内部实现为红黑树。

第十一章：泛型算法

1. 插入迭代器：必须包含< iterator> 文件
   - back_inserter生成一个绑定在该容器上的插入迭代器。在试图通过这个迭代器给元素赋值时，赋值去处将调用push_back在容器中添加一个具有指定值的元素。
   - front_inserter的操作类似于back_inserter：该函数将创建一个迭代器，调用它所关联的基础容器的push_front成员函数代替赋值操作。
   - insert适配器：提供更为普通的插入形式，insert(ilist，it)返回ilist容器的it位置前一个位置进行插入的迭代器。
2. 算法的copy版本：这些算法不对原序列进行修改，而的创建一个新序列存储元素的处理结果。如replace_copy(); 等等。
3. unique()算法：（调用该方法之前应该保证排好序了，否则该算法不能正常工作）该算法“删除”**相邻**的重复元素，然后重新排列输入范围内的元素，并且返回一个迭代器，表示无重复的值范围的结束。这里的“删除”并不是真的删除，而是移动了位置，算法不直接修改容器的大小，如果需要添加或删除元素，必须使用容器操作。这也是算法能够实现泛型的原因。
4. iostream迭代器：（这些迭代器将它们所对应的流视为特定类型的元素序列）
   - istream_iterator<T> in(strm); 创建从输入流strm中读取的T类型对象的输入迭代器。
   - istream_iterator<T> in; 创建istream_iterator对象的超出末端迭代器。
   - ostream_iterator<T> in(strm); 创建将T类型的对象写到输出流strm的输出迭代器。
   - ostream_iterator<T> in(strm, delim); 同上，在输出过程中使用delim作为元素的分隔符。delim是以空字符串结束的字符数组。
5. 任何已定义输入操作符“>>”的类型都可以定义istream_iterator。类似的，任何已定义输出操作符“<<”的类型都可以定义ostream_iterator。
6. 输入迭代器的使用范例：　　//这是从输入中初始化vector的极佳的例子

   ```
   istream_iterator<int> in_iter(cin);
   istream_iterator<int> in_eof;
   vector<int> vec(in_iter, in_eof);
   ```
7. 输出迭代器的使用范例：

   ```
   ostream_iterator<string> out_iter( cout, “\n”);
   istream_iterator<string> in_iter(cin), eof;
   ```

```
while (in_iter != eof)
    *out_iter++ = *in_iter++;
```

8. 反向迭代器与原迭代器的相互转化： r_iter.base();
9. map、set和list类型提供双向迭代器，而string、vector和deque（deque是基于vector实现的）容器上定义的迭代器都是随机访问迭代器，用作访问内置数组元素的指针也是随机访问迭代器。
10. 对于list对象，应该优先使用list容器特有的成员版本，而不是泛型算法，泛型算法用在list对象上时效率非常低下。

    任何时候都应该优先使用成员函数版本，一般对于同一个功能来说成员函数版本都能提供更佳的效率！

第三部分：类和算法

第十二章：类

1. 在类已经声明还未定义时，类是一个不完全类型。不完全类型可以定义指向其的指针或引用，或者用于声明（而不是定义）使用该类型作为形参类型或返回类型的函数。
   在创建类的对象之前，必须完整的定义该类！
2. 基于成员函数是否为const，可以重载一个成员函数；同样地，基于一个指针形参是否指向const，可以重载一个函数。
3. 可变数据成员：mutable int size; 任何成员函数，包括const函数，都可以改变mutable成员的值。
4. 构造函数不能声明为const，普通构造函数是可以初始化const对象的。而且，事实上const对象只能在构造函数中初始化。
5. 在包含自定义类的类中，如果那个自定义类没有提供默认的构造函数，那么就必须用初始化式来初始化这个成员。
6. 因为构造函数相当于分两个阶段来执行的：（1）初始化阶段；（2）普通的计算阶段。初始化阶段发生在执行初始化列表的时候，普通的计算阶段才发生在构造函数体执行阶段。所以，初始化const或引用类型的数据成员的唯一机会是在构造函数初始化列表中。
7. 成员被初始化的次序就是定义成员的次序。而销毁的次序与定义成员的次序正好相反。
8. 类通常需要一个默认的构造函数，因为需要时候编译器都需要隐式的调用类的默认构造函数。实际上，如果定义了其他的构造函数，则提供一个默认的构造函数几乎总是对的。
   实际上，不要轻易的提供默认构造函数，如果他没有必须的话。编译器自动生成的trivial版本可能能提供更好的性能。
9. 定义隐式类型转换：可以用单个实参来调用的构造函数定义了从形参到该类类型的一个隐式转换。
   Sales_item (const std::string &book) {} 定义了一个从string到Sales_item的隐式类型转换。
   可以通过声明为explicit来防止在需要隐式转换的上下文中使用构造函数。explicit只能用于类内部的构造函数声明上，在类的定义体外部所做的定义上不需要要重复。（对于所有的单形参的构造函数，除非明确的要使用这个构造函数来完成隐式转换，否则必须加上explicit来防止隐式转换）
   除非有明显的理由想要定义隐式转换，否则，单形参构造函数应该声明为explicit
10. 重载函数与友元：类必须将重载函数中每一个希望设为友元的函数都声明为友元。
11. static数据成员独立于该类的任意对象而存在；每个static数据成员是与类关联的对象，并不与该类的对象相关联。
    static成员函数没有this形参，它可以直接访问所属类的static成员，但不能直接使用非static成员。
    static成员函数也不能被声明为虚函数。
12. static数据成员必须在类定义体的内部进行声明，在外部定义（必须正好一次），static成员不是通过类构造函数来进行初始化的，而是应该在定义时进行初始化。
13. static关键字只能用于类定义体内部的声明中，定义不能标示为static。

```
struct A{
    static  string  const_err_msg;                  //内部的声明，需要标示为static
};
string  A::const_err_msg = “Some error!”;     //外部的定义，不能标示为static
```

14. static数据成员的类型可以是该成员所属的类类型，非static成员被限定声明为自身对象的指针或引用。

    ~~这也表明了static的工作原理就是一个指针，指向特定类型的指针。~~

    不对，实际上static成员的工作原理是会被编译器在编译时进行扩展！

第十二章：复制控制

1. 为了防止复制，类必须显式声明其复制构造函数为private。声明而不定义成员函数是合法的，但是，使用未定义成员的任何尝试都会导致编译错误。
2. 复制构造函数与赋值操作的区别：
   - 复制构造函数：A b = a; 或 A b(a); // a来初始化b。调用A(A &_a){}的构造函数。

     复制构造函数就是接受单个类类型的**引用**（必须为引用，如果不是引用会造成无穷递归）形参（通常为const）的构造函数。当定义一个**新**对象并用一个同类型的对象对它进行初始化时，将显式的调用复制构造函数；当将该类型的对象传递给函数或从函数返回该类型的对象时，将隐式的使用复制构造函数。
   - 赋值操作：b = a; //这个是赋值，调用操作符b.operator=(a)，可以重载。

     //错，实际上，这里会调用copy构造函数产生一个临时对象，然后再调用operator=运算符
   - 如果一个类需要一个复制构造函数，那几乎也一定需要重载赋值运算符。
   - 二种初始化[1]：A b=a; 和 A b(a);都是调用复制构造函数。**只有**显式的赋值操作b=a;时才是调用赋值操作符。函数参数和函数返回值的复制也是调用复制构造函数。
   - 可以这样理解，新创建一个变量当然要调用构造函数，所以所有的涉及新变量的都会调用复制构造函数。

     而b=a;中的b早已创建好了，就不应该是构造函数，应该是赋值操作符。
3. 一种特别常见的情况需要类定义自己的复制控制成员的：类具有指针成员。
4. 一般来说，最好或隐式定义默认构造函数和复制构造函数。只有不存在其它构造函数时才合成默认构造函数。如果定义了复制构造函数，也必须定义默认构造函数。
5. 析构函数是变更超出作用域时自动运行，一般地，局部变量在遇到右花括号时将运行局部变量的析构函数。当对象的引用或指针超出作用域时，不会运行析构函数。只有删除指向动态分配对象的指针或实际对象（而不是对象的引用）超出作用域时，才会运行析构函数。

   撤销总是逆序的！
6. 三法则：如果类需要析构函数，则它也需要赋值操作符和复制构造函数。（一般是有指针类型的

[1]这2种初始化形式的区别在前面已经更正过了，**隐式和显式的区别！**

时候才需要考虑这些）

7. 即使编写了自定义的析构函数，合成的析构函数也会在自定义的析构函数运行完**之后**运行。
   合成的析构函数按对象创建时的逆序撤销每个非static成员。
   只有析构函数是一定会被合成出来，并且它会被自定义的析构函数隐式调用的！
8. *C++*
9. 重载操作符必须具有至少一个类类型或枚举类型。
10. 包含指针的类需要特别注意复制控制，原因是复制指针时只复制指针中的地址，而不会复制指针所指向的对象。如果需要，则要自己手动去复制指针所指向的对象。

第十四章：重载操作符与转换

1. 重载操作符必须具有至少一个类类型或枚举类型的操作符。这条规则强制重载操作符不能用于内置类型对象的操作符的含义。
   int operator+(int, int);　　//这是错误的，将引发编译错误
   int operator+(myClass, int);　　//这是正确的重载操作符的方法
2. 除了函数调用操作符operator()之外，重载操作符时使用默认实参是非法的。
3. 重载操作符后将不能保证短路求值的特征，所以重载“&&”、“||”、逗号运算符不是好的做法。
4. 编译器如何隐式的调用操作符运算的：
   - 当为友元重载时：编译器隐式调用operator+(item1, item2)来执行运算符的。
   - 当为成员函数重载时：编译器隐式调用item1.opeator+(item2)来执行运算。
   - ```
     struct my_class{
             friend ostream &operator<< (ostream&, const my_class &); //友元声明
             bool operator==(const my_class &c2)　　//成员函数重载
             {return true;}
     };
     ostream &operator<< (ostream &output, const my_class &c)　　//友元重载
     {return output;}
     ```
5. IO操作符重载必须定义为非成员函数。这是因为IO操作符必须有一个类型为ostream　&或istream &类型的左操作数，因此它必须是一个非成员函数。
   下标操作符重载必须定义为类成员函数。
   成员访问运算符重载（*，->）也必须定义为类成员函数。
   重载一元操作符时，应该把操作符函数用作类的成员而非友元函数。
6. 重载箭头操作符：箭头操作符很特殊，由编译器来处理获取成员的工作 。
   当这样编写： point -> action(); 时，由于优先级规则实际上等价于：(point -> action)();
     ```

(1) 如果point是一个指针，指向具有名为action的成员的类对象，而编译器将代码编译为调用该对象的action成员。

(2) 否则，如果action是定义了operator->操作符的类的一个对象，

则point -> action 与 point.operator->() -> action相同。即执行point的operator->()，然后使用该结果重复这三步。

(3) 否则，代码出错！

重载箭头操作符必须返回指向类类型的指针，或者返回定义了自己的箭头操作符的类类型对象。

7. 操作符重载只是为了编写代码方便简洁，并不具有非常深刻的意义。如果不是必须或者确定使用操作符重载能使程序更加的清晰，就不要重载操作符，大多数时候给操作取一个名字会更好。
   但有例外：如果类要用于容器的话，那么应当定义好“==”和“<”操作符，因为许多算法会假定这些运算符存在，如果未定义，将使容器的许多操作不可用。
8. 重载的复制操作符必须定义为成员函数，而且复制操作符必须返回对*this的引用。
   Sales_item& Sales_item::operator+=(const Sales_item &rhs);
9. 类定义下标操作符时，一般需要定义两个版本：一个为非const成员并返回引用，另一个为const成员并返回const引用。
   int& operator [] (const int index);
   const int& operator [] (const int index) const;
   返回的是引用，因为下标操作符返回的是左值，可以对其进行赋值的。
10. 自增和自减操作符：
    - 前缀式操作符应该返回被增量或减量对象的引用
    - 后缀式操作符应该返回旧值（即未自增或自减的值），并且，应该作为值返回，而不是引用。(所以后缀的形式很多时候并不好用的)
    - 区分前后缀操作符：后缀式的操作符接受一个额外的int型形参（值为0）。
11. 函数对象：定义了调用操作符的类，它是行为类似于函数的对象。函数对象可以代替函数的作用，而且比函数更加的灵活。（看p450的将函数对象用作标准库算法的示例）

```
struct absInt{
   int operator() (int val){
     return val<0 ? –val : val;
   }
}
absInt absObj;              //先创建一个函数对象
int abc = absObj(-34);      //调用的方法
```

12. 标准库定义了许多函数对象#include <functional>。 常用作通用算法的实参。
    定义了二个绑定器：bind1nd,bind2nd将一个操作符绑定到一个函数对象。
    eg:bind1nd(less_equal<int> (), 10) 将lessequal打造成了一个比较是否小于10的函数
    还定义了二个求反器：not1,not2分别对一元操作和二元操作函数对象求反。
13. 除了定义到类类型的转换之外，还可以定义从类类型的转换。

转换操作符：它定义将类类型值转变为其它类型值的转换。转换函数必须是成员函数，不能指定返回类型，并且形参表必须为空。

```
struct small_int{
   operator  int()  const {return val;}      //这个const是可选的
};
```

这种类型转换及其容易出错，一般来说不要对超过一个内置类型使用这种转换。多写一些代码，使用to_int(); to_string(); 会使程序更加清晰容易理解。否则会出现莫名其妙的编译时错误。

但好处也非常明显，当需要时，编译器会自动进行调用以产生这些类型的临时对象。

14. 只能应用一个类型转换，类类型转换之后不能再跟另一个类类型转换。（隐式类型转换的次数不能超过一次！）如果需要多个类类型转换，则代码将出错。示例如下：

```
struct big_int{
  operator small_int() const { return val; }
}
```

这时，big_int可以用在任何需要使用small_int类型的地方，但是不能用在需要使用int类型的地方。即C++内部机制只能判断出一次转换，否则会出错。

第四部分：面向对象编程与泛型编程

第十五章：面向对象编程

1. 面向对象编程基于三个基本概念：数据抽象、继承、动态绑定。C++中：用类进行数据抽象、用派生类继承基类的成员、动态绑定使编译器能够在运行时决定是使用基类中定义的函数还是在派生类中定义的函数。

   区分“面向对象”和“基于对象”这两个概念的不同之处！
2. 动态绑定：编译器生成代码在运行时根据引用或指针的类型调用不同的函数的性质。
3. 动态绑定的两个必要条件：
   - 引用或指针：引用和指针的静态类型和动态类型可以不同，这是C++动态绑定的基石。
   - 虚函数：如果调用非虚函数，则无论对象是什么类型，都执行基类类型所定义的函数。因为非虚函数的调用在编译阶段就已经确定好了，而虚函数则是推迟到运行阶段。
4. 除了构造函数之外，任意非static成员函数都可以是虚函数。

   virtual只在类内部的成员函数的声明中出现，不能在类定义体外部出现的函数定义上。

   基类通常应将派生类需要重定义的任意函数定义为虚函数。而且析构函数几乎一定应该定义为虚函数，因为派生类在运行时都应该调用自己的析构函数。如果析构函数不定义为虚函数，则将调用基类的析构函数。

   还有不行的，比如说成员函数模板也一定不能为virtual，模板与virtual机制是冲突的！
5. 派生类只能通过**派生类对象**访问基类的protected成员，派生类对其**基类对象**的protected成员没有特殊的访问权限。
6. 派生类中虚函数的声明必须与基类中定义的方式完全匹配，但有一个例外：返回对基类型的引用（或指针）的虚函数。派生类中的虚函数可以返回基类函数所返回类型的派生类的引用（或指针）。
7. 一旦函数在基类中声明为虚函数，它就一直为虚函数。派生类用不用virtual都没有影响。派生类可以使用virtual关键字，但是不是必须要这么做。

   良好的编译风格是对于继承而来的virtual函数，也一定要再使用virtual标识出来！
8. 如果需要声明（但不实现）一个派生类，则声明包含类名但不包含派生列表，只有在其定义实现时才需要包含派生列表。
9. 通过基类的引用或指针调用虚函数时，默认实参为在基类虚函数中指定的值；如果通过派生类的指针或引用调用虚函数，则默认实参是在派生类的版本中声明的值。

   Effective C++ 建议：绝不要重定义继承而来的默认实参！
10. 继承的属性：
    - public继承：保持访问级别不变。
    - protected继承：基类的public和protected在派生类中都为protected
    - private继承：基类的public和protected在派生类中都为private

- public继承又称接口继承，它具有与基类相同的接口，public派生类的对象可以用在任何需要使用基类对象的地方，而protected和private继承则不可以。

11. 在protected和private继承中，可以使用using使得个别成员恢复继承成员的访问级别，但绝不能够比基类型更宽松。

    eg: using Base::size;　　//将Base::size声明为上一个访问标签所控制的范围

12. 使用class保留字定义的派生类默认具有private继承，而用struct保留字定义的类默认具有public继承。

    但在实际中，几乎所有的继承都应该是public继承。（即所有的class都应该显式声明为public继承）

13. 友元关系不能继承。
14. 如果基类定义了static成员，则在整个继承层次中只有一个这样的成员。
15. 派生类的指针或引用是可以自动的转换成基类的指针或引用的。但是对于派生类的对象，却没有自动的转换能够转换成基类的对象，只能用派生类对象对基类对象进行初始化或赋值。这个时候，将调用基类的初始化构造函数或赋值操作符。
16. 如果是private继承，则从private继承派生的类不能转换为基类。如果是protected继承，则后续派生类的成员可以转换为基类类型。

    要确定到基类的转换是否可行，可以考虑基类的public成员是否可以访问，如果可以，转换是可行的，否则是不可行的。

17. 构造函数和复制控制成员不能继承，每个类定义自己的构造函数和复制控制成员。
18. 编译器生成的默认的构造函数会首先调用基类的默认构造函数初始化基类部分，然后再运行派生类的默认构造函数。
19. 一个类只能初始化自己的直接基类。

```
class bulk_item : public item_base{
  public: bulk_item(string book, double price, int qty) :
                        item_base(book, price), min_qty(qty)  {}
}
```

    构造函数初始化列表为类的基类和成员提供初始值，它并不指定初始化的执行次序。首先初始化基类，然后根据声明次序初始化派生类的成员。（这一点在前几章也有提到过）

20. 复制控制与继承：如果派生类显式定义自己的复制构造函数或赋值操作符，则该定义将完全覆盖默认定义。
    - 如果派生类定义了自己的复制构造函数，该复制构造函数一般应显式地使用基类复制构造函数初始化对象的基类部分。因为编译器将不会再隐式的调用基类的复制构造函数。

      ```
      class derived：public base{
          public: derived(const derived &d)：base(d) {}
      }
      ```

    - 如果派生类定义了自己的赋值操作符，则该操作符必须对基类部分进行显式赋值。同样也是因为编译器不会再隐式的调用基类的部分了。

```
derived& derived::operator=(const derived &rhs){
    if (this!=&rhs){              //赋值操作符必须防止自身赋值
      base::operator=(rhs);       //显式调用基类的部分
    }
}
```

21. 由于析构函数的特殊性，要保证运行适当的析构函数，基类中析构函数必须为虚函数。 所以即使析构函数没有工作要做，继承层次的根类也应该（必须）定义一个虚析构函数。

    这是因为：经常通过基类的指针调用delete删除派生类对象，如果没有将析构函数定义为虚函数，将不符合动态绑定从而调用基类的析构函数。这将只删除派生类的基类部分，派生类扩充的部分将不会被删除，造成了内存泄漏。

22. 在构造、赋值、析构函数中，析构函数总是应该定义为虚函数（ ）。而构造函数和赋值操作符总是不应该为虚函数。

    构造函数是因为：构造函数运行的时候，对象的动态类型还不完整，这时根本不能发生动态绑定；

    赋值操作符是因为：基类与派生类的赋值操作符的形参类型是不相同的，如果基类的赋值操作符为虚函数，则所有的派生类也将得到一个形参为基类类型的赋值操作符，这极其容易引起混淆，而且没有任何的好处。

23. 在基类构造函数或析构函数中，此时对象还不是一个派生类对象，应该将派生类当作基类类型对象对待。如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本。因为在基类的构造函数中，派生类特有的部分还不存在，调用派生类的函数时访问派生类特有的部分时将会引发内存级别的错误。（考虑构造函数和析构函数的执行顺序可以很容易的理解！）

24. 名字查找在编译时发生，所以对象、引用、指针的静态类型决定了对象能够完成的行为。

25. 在基类和派生类中使用同一名字的成员函数，其行为与数据成员一样：在派生类作用域中派生类将屏蔽基类成员。即使函数原型不同，基类成员也会被屏蔽。（因为名字查找发生在编译阶段）

    通俗理解：派生类不能重载基类的成员函数，使用同名的成员函数将会屏蔽所有的基类同名成员函数。解决方法是使用using（只指定一个名字，不能指定形参，using就是为解决作用域而生）扩展作用域。

26. 含有（或继承）一个或多个纯虚函数的类是抽象基类。除了作为抽象基类的派生类的对象的组成部分，不能创建抽象类型的对象。

    virtual bool ck(string str) = 0;　　//纯虚函数首先是一个虚函数，所以=0只能用于virtual

27. 容器与继承：将派生类型对象加入到基类容器中时，是将派生类型复制一份为基类类型然后放入容器中，这是由容器的性质决定的。而将派生类对象复制为基类对象时，派生类对象的特有部分将被切掉。

    解决办法是使用容器保存基类类型的指针或引用。即vector< myClass * > vec;

28. 句柄类：因为C++动态绑定的必要条件之一是指针或引用，而管理指针或引用是很复杂的事情，

所以可以通过句柄类的技术来通过对象管理指针，可以让用户通过类对象来调用函数并获得多态行为。

例如：vector<item_base *>是一个包含指向item_base类型的指针的容器，但程序员必须自己管理指针，容易出错。可以通过一个定义一个Sales_item的句柄类，通过使用vector<Sales_item>来避免使用指针。操作指针的细节交由Sales_item类在内部进行处理。Sales_item在这里表现得像一个智能指针，通过自定义默认构造函数，复制构造函数、赋值操作符、析构函数、解引用操作符、箭头操作符 等等使用可以通过Sales_item来防止指针悬垂等等问题。

简单地说：句柄类就是使用对象获得了指针的性质的技术。可以获得动态绑定行为而无需管理指针。

直白一点就是：句柄类就是一个管理指针的类。

第十六章：模板与泛型编程

1. 泛型编程与面向对象编程时的多态性的区别：面向对象编程所依赖的多态性是运行时的多态性，根据运行时的不同类型来确实运行时的行为。而泛型编程的多态性是编译时多态性或参数式多态性。
2. 模板的工作原理：

```
template <class T>
inline int compare(const T &v1, const T &v2){
  if (v1<v2) return -1;
  if (v1>v2) return 1; //实际上，这里用if(v2<v1)更好，对T类型要求更少，只要定义“<”
  return 0;
}
```

对于模板形参表中的T，编译器会在编译时将T替换为调用的类型，这是编译时的工作。所以这将产生一个compare的副本（实例），其中的T已经替换为了int或者string。

但是，类模板的每次**实例化**都会产生一个独立的类类型。即二次用相同的形参实例化模板得到的也是不同的二个实例。

3. 模板实参演绎：多个类型形参的实参必须完全匹配。

如对于template < class T > int compare(T &v1, T &v2); 调用 compare(short, int);将会产生错误，虽然short可以转换为int。

这是因为在调用compare时，从第一个实参推断出的模板类型是short，从第二个实参推断出int类型，两个类型不匹配，所以模板实参推断失败。除完全匹配外，编译器还接受两种转换：

- const转换：接受const引用或指针的函数分别可以用非const对象的引用或指针来调用，无须产生新的实例化。
- 数组或函数到指针的转换：如果模板形参不是引用类型，则对数组或函数类型的实参应用

常规指针转换。

4. 在为模板进行实例化推断的时候，必须是在编译期间就能唯一确定模板的类型，不能产生二义性，否则将产生编译（或链接）错误。
5. 模板编译模型：与普通的类和函数不同，模板要进行实例化，编译器必须能够访问定义模板的源代码。当调用函数模板或类模板的成员函数的时候，编译器需要函数定义，需要那些通常放在源文件中的代码。

   常用的有二种编译模型：

   1. 包含编译模型：编译器必须看到模板的定义，可以通过在声明函数模板或类模板中的头文件中添加一条#include指示定义可用，该#include引入包含相关定义的源文件：

      ```
      //header file utlities.h
      #ifndef UTLITIES_H
      #define UTLITIES_H
      template < class T > int compare( const T&v1, const T &v2 );
      //这条语句引入的头文件对应的源文件，使得模板的定义可见而且保持了头文件和源文件的分离
      #include "utilties.cpp"
      #endif
      ```

   2. 分别编译模型：使用export关键字让编译器知道需要记住给定的模板的定义。在函数模板或类模板的定义中指明函数模板为导出。（头文件不能声明为export，否则只能被程序中的一个源文件引用）

      ```
      //define file utilties.cpp  头文件和普通函数的头文件不变
      export template < teypename Type > Type sum( Type t1, Type t2 )
      { ... }
      //define file Queue.h   头文件都不变
      template < class Type > class Queue { ... };
      //define file Queue.cpp  在源文件中要重新声明一次，然后加上export
      export template < class Type > class Queue;
      #include "Queue.h"
      ```

   3. 分别编译模型在C++11中已经被否决！以后只用包含模型！
6. 在模板定义内部指定类型 T::size_type，由于编译器无法判定这是一个类型还是指T类型的一个成员，所以编译器会默认的当作T类型的一个成员来处理。

   如果T::size_type指的是一个类型，则需要在内的内部显式的声明：

   ```
   typename T::size_type current_size;
   ```

7. 显式实参和隐式实参混合使用：

   ```
   templates <class T1, class T2, class T3>
   T1 sum(T2, T3);
   ```

   可以这样调用：long val = sum<long>(i, j); 这里显式指定了T1的类型，T2,T3的类型可以通过调用的i,j的类型隐式指定。这种方法必须将T2,T3放在模板形参的后面。

8. 对于稍复杂的类，将声明与定义分开是非常有助于在宏观上理解程序的。除了非常简单的成员函数外，稍微复杂一些的成员函数都应该将声明与定义分开。
9. 一般而言：当调用函数时，编译器只需要看到函数的声明。类似地，定义类类型的对象时，类定义必须可用，但成员函数的定义不是必须存在的。因些，应该将类定义和函数声明放在头文件中，而普通函数和类成员函数的定义放在源文件中。
10. 非类型模板实参必须是编译时常量表达式，因为模板推断是在编译时进行的。
11. 类模板中的友元声明：

```
template <class Type> class queue_item{
  template <class T> friend class queue;//使用不同的类型形参，将queue的所有实例设为友元
  friend class queue<int>;            //友元关系只扩展到queue的特定实例queue<int>
  template <class Type> friend class queue;//使用相同的类型形参，只授于相同实参的实例为友元
}
```

编译器将友元声明也当作类或函数的声明对待，但是如果想要限制对特定实例化的友元关系时，必须在可以用于友元声明之前声明类或函数。（指上条中的后二种友元声明）

12. 类模板的static成员，模板类的每个实例化都有自己的static成员。foo<int>类型的任意对象共享同一个static成员foo<int>::size_t，而foo<string>类型的任意对象共享另外一个同名的static成员foo<string>::size_t。

类模板的static成员在外部定义时(static内部声明外部定义)必须指出它是类模板的成员

~~template <class T> std::string foo<T>::msg = "Message";　　//已试验为错误！~~

正确用法如下（经过试验）：

```
template < class T > struct Foo
{
    static string ToString;
};
string Foo< int >::ToString = "int!";     //这其实，是在使用特化！
int main( int argc, char** argv )
{
    cout << Foo<int>::ToString;
    return 0;
}
```

13. 函数模板特化：

```
template < class T >                //定义泛型模板
int compare( const T &v1, const T &v2 )
{
if ( v1 < v2 ) return -1;
```

```
if ( v2 < v1 ) return 1;
return 0;
}

template<>                              //告诉编译器接下来的是模板特化
int compare< const char* >( const char* const &v1, const char* const &v2 )
{
return strcmp( v1, v2 );
}

int main( int argc, char** argv )
{
cout << compare< const string >( "zHello", "Worlddd" );        //将调用泛型版本
cout << compare< const char* >( "zHello", "Worlddd" );         //将调用特化的版本
return 0;
}
```

14. 类模板的特化与函数模板的特化类似，但值得注意的是,特化可以定义与模板本身完全不同的成员。所以可以这样理解特化：在调用特化版本的时候是将泛型版本的定义完全屏蔽，所以二者的定义都可以不同。也可以这样想：编译器通过特别的处理，使得这二个看似同名的类定义特化为二个不同名的类定义，所以他们的定义可以不同。

    在类的外部定义成员时不能加 template<> 标记：

    ```
    void Queue< const char* >::push( const char * val )
    { ... }
    ```

    还可以用这样的语法只特化某些成员而不是特化整个类。

15. 类模板的部分特化，也叫作偏特化：（C++中只允许对class templates部分特分，在function templates身上部分特化是行不通的）

    ```
    template < class T1, class T2 > class some_template{
        // ...          泛型版本
    };
    template < class T1 > class some_template< T1, int >{
        // ...          特化版本
    };
    some_template< int, string > foo;              //因为第二个模板参数不是int，所以调用泛型版本
    some_template< string, int > bar;              //因为第二个模板参数是int，所以调用特化版本
    ```

16. 重载与函数模板： 当两个函数一样匹配时，普通函数优先于模板版本。

    设计即包含函数模板又包含非模板函数的重载函数集合是困难的，而且会使函数的用户感到奇怪。所以：定义函数模板特化几乎总是比使用非模板版本更好。

    ```
    /**********************重载函数模板匹配约定**************************
    ```

* 1. 寻找和使用最符合函数名和参数类型的函数，若找到则调用它

* 2. 否则，寻找一个函数模板，将其实例化产生一个匹配的模板函数，若找到则调用它

* 3. 否则，寻找可以通过类型转换进行参数匹配的重载函数，若找到则调用它

* 4. 如果按以上步骤均未找到匹配函数，则调用错误

* 5. 如果调用有多于一个的匹配选择，则调用匹配出现二义性

**********/

17. 模板匹配的时候必须是严格匹配。

对于template < class T > int compare(T &v1, T &v2); 和 int compare(int, int); 来说：

compare(char, int)将调用普通函数版本，因为模板匹配的时候要求严格匹配。如果此时普通函数版本的compare不存在，将在编译时出错：模板 参数"T"不明确！（参见第3条：模板实参推断）

第五部分：高级主题

第十七章：用于大型程序的工具

1. 异常以类似于将实参传递给形参的方式抛出和捕获异常，所以形参可以是引用和非引用两种类型。当形参是非引用时，抛出的对象必须是可以复制的。
   异常对象本身是被抛出的对象的副本，是否再次将异常对象复制到catch位置取决于异常说明符是引用类型还是非引用类型。
2. 一般而言，在处理异常的时候，抛出异常的块中的局部存储就不存在了。但是throw表达式会初始化一个异常对象，这个对象由throw创建，并被初始化为抛出表达式的副本。异常对象将传递给catch，并在完全处理了异常后撤销。
   即：异常对象由编译器进行特殊的处理，保证其在异常处理期间可以访问。
   追加：这个复制出来的异常对象是被创建在编译器专门为异常对象保留的内存区域的，以确保它不会在栈展开时被销毁掉！
3. 在处理异常的栈展开期间，释放局部对象所用的内存并运行类类型局部对象的析构函数。如果一个块直接分配资源，而且在释放资源之前发生异常，在栈展开期间将不会释放该资源。
4. 析构函数应该从不抛出异常：在为某个异常进行栈展开的时候，析构函数如果又抛出自己的未经过处理的另一个异常，将会导致调用标准库terminate函数，一般而言，terminate函数将调用abort函数，强制从整个程序非正常退出。
   可以通过set_terminate和set_unexception来指定发生未处理异常时的行为。
5. 在查找异常的匹配代码时，除了下面几种可能外，异常类型与catch说明符的类型必须完全匹配：
   - 允许从非const到const的转换。
   - 允许从派生类型到基类类型的转换。
   - 将数组转换为指向数组类型的指针，将函数转换为指向函数类型的适当指针。
6. 重新抛出异常：throw; 被抛出的异常是原来的异常对象，而不是catch形参。
   注意区别
   throw;
   throw ex;
   前一种写法是直接抛出原来的异常，不会有任何的复制；而后一种写法是再抛出一个新的异常，只是这个异常是使用原来的异常复制得来的。
7. 自动资源释放：（考虑下面的程序）

```
void Foo()
{
    vector<string> v;
    string s;
```

```
    while ( cin >> s )
        v.push_back( s );
    string *p = new string[ v.size() ];
    delete [] p;  //这里如果在函数内部发生异常，数组所分配的内存将不会被释放
}
```

因为vector是一个类，类的析构函数在异常发生时能保证一定运行。而数组是动态分配的空间，所以在内部产生异常时将不能回收，从而产生内存泄露。尽量使用标准库类！

通过定义一个类来封装资源的分配和释放，可以保证正确释放资源，这一技术称为“资源分配即初始化”简称RAII。

8. 函数测试块：可以使用函数测试块将一组catch子句与函数联成一个整体。

```
int main()
try{
    ...
}
catch (...){}
```

9. 构造函数要处理来自构造函数初始化式的异常，唯一的方法是将构造函数编写为函数测试块。并且将try放在成员初始化列表之前。

```
my_class ()
try : value(0), next(0){
    ...
}
catch (...){}
```

10. auto_ptr是通过一个类来封闭资源的分配和释放，可以保证正确释放资源。在头文件memory中定义。
    - auto_ptr对象的复制和赋值是破坏性的操作，所以不能将atuo_ptr对象存储在标准容器中。
    - auto_ptr重载了*和->操作符，所以可以支持普通指针的行为，同时还保证自动删除auto_ptr对象所引用的对象。
    - auto_ptr不能指向静态分配对象的指针。否则调用析构函数时行为未定义。
    - auto_ptr不能指向数组，因为其使用delete而不是delete []。
    - 永远不要让两个auto_ptr对象指针同一对象。
    - auto_ptr对象的复制和赋值是破坏性操作：当复制auto_ptr对象或者将它的值赋给其它auto_ptr对象的时候，将基础对象的所有权从原来的auto_ptr对象转给副本，原来的auto_ptr对象重围为未绑定状态。
    - auto_ptr也在C++11中被废弃。取而代之的是shared_ptr和scoped_ptr。
11. ~~函数的异常说明：void recoup(int _i) throw (runtime_error);~~

~~throw后面的异常说明表示：如果recoup抛出一个异常，该异常将是runtime_error类型或者是由~~

~~runtime_error派生的类型。空的说明列表指出函数不抛出任何异常：throw();~~

~~在编译时，编译器不能也不会验证异常说明。如果函数违法了异常说明，将只能在运行时发现。~~

所以异常说明用处不大，但经常用来保证不抛出任何异常。即throw();

异常说明符也在C++11中被废弃，以后不要再使用了！

12. ~~基类中虚函数的异常说明，可以与派生类中对应虚函数的异常说明不同。但是，派生类虚函数的异常说明必须与对应基类虚函数的异常说明同样严格或更严格。~~
13. ~~异常说明是函数类型的一部分。这样，也可以在函数指针的定义中提供异常说明：~~

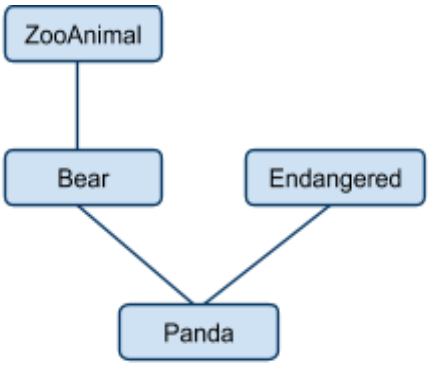
    ~~void (*pf) (int) throw (runtime_error);~~
14. 命名空间可以在全局作用域或其它作用域内部定义，但不能在函数或类内部定义。
15. 名字只在声明名字的文件中可见，所以另一文件中如果需要访问该变量或类，必须声明该名字，即使同属于一个命名空间也一样。
16. 匿名命名空间：未命名的命名空间用来声明局部于文件的实体（　　），从不跨越多个文件。

    每个文件都有自己的未命名的命名空间，在未命名的命名空间中定义的变量在程序开始时创建，在程序结束之前一直存在。

    如果头文件定义了未命名的命名空间，那么，在每个包含该头文件的文件中，该命名空间中的名字将定义不同的局部实体。
17. 命名空间别名：namespace tq = System::Data::Sql;
18. 尽量使用using声明而不要用using指示，不要偷懒。除非是非常小的供测试用的程序。
19. 在类中查找名字有一个重要的特点：如果名字不是局部于成员函数的，就试着在查找更外层作用域之前在类成员中确定名字。所以：类内部所定义的成员可以使用出现在定义文本之后的名字。
20. 有一个或多个类类型形参的函数的名字查找中包括定义每个形参类型的命名空间：

    ```
    std::string s;
    std::cin >> s;          //这里其实是调用std::operator>>( std::cin, s );
    ```

    由于满足了类类型形参和函数二个条件，在名字查找时就包括了定义了cin的std命名空间，于是调用成功。
21. 为了提供命名空间中所定义模板的自己的特化，必须保证在包含原始模板定义的命名空间中定义特化。
22. 多重继承的派生类继承其所有的父类的属性。
23. 多重继承的构造和析构顺序：（只与继承的顺序有关）



class Panda : public Bear, public Endangered;

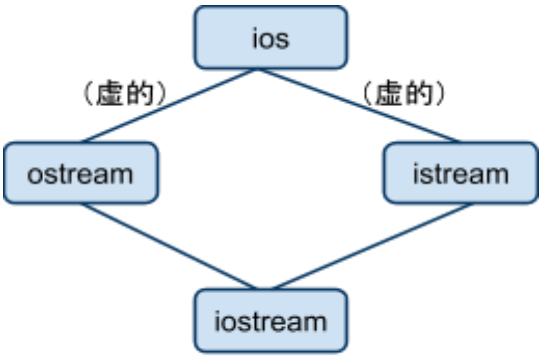
构造顺序： ZooAnimal, Bear, Endangered, Panda

析构顺序：~Panda, ~Endangered, ~Bear, ~ZooAnimal

24. 像单继承一样，如果具有多个基类的类定义了自己的析构函数，该析构函数只负责清除派生类。如果派生类定义了自己的复制构造函数或赋值操作符，则类负责复制（赋值）所有的基类子部分。
25. 虚继承：虚继承是一种机制，类通过虚继承指出它希望共享其虚基类的状态。如在istream和ostream类对它们的基类进行虚继承，通过使基类成为虚基类，istream和ostream指定，如果其它类（如iostream）同时继承它们两个，则派生类中只出现它们的公共基类的一个副本。通过在派生列表中包含关键字virtual设置虚基类：

    class istream : public virtual ios { ... };

    class ostream : virtual public ios { ... };



虚继承iostream层次（简化的）

26. 无论虚基类出现在继承层次中的任何地方，总是在构造函数基类之前构造虚基类。所以，这一整套的类的继承层次往往只能由一个人或一个组织来设计。
27. 特殊的初始化语义：在虚派生中，为了避免重复初始化虚基类，约定由最低层派生类的构造函数初始化虚基类。

    对于iostream的继承层次中，虽然ios不是iostream是直接基类，但是也由iostream的构造函数来初始化ios基类，而且是必须由iostream来初始化。在构造iostream的过程中，会忽略掉ostream和istream对它们的直接基类ios的初始化部分。
28. 无论虚基类出现在继承层次的任何地方，总是在构造非虚基类之前构造虚基类。

第十八章：特殊工具和技术

1. 优化内存分配（内存分配器技术）：自定义特定类型分配内在和初始化的动作，这样的类使用的通用策略是，预先分配用于创建新对象的内在，需要时在预先分配的内存中构造每个对象。最鲜明的例子是标准库的vector。

   关于自己分配内存十分罕见，大部分常用的操作标准库都已经做好了，当需要的时候再翻书，书上这部分介绍的十分清楚易懂。现在研究太深而又用不到的话也会很容易的忘记。
2. RTTI：程序能够使用基类指针或引用来检索这些指针或引用所指对象的实际派生类型。
   - typeid操作符，返回指针或引用所指对象的实际类型。

     typeid()返回type_info类型，多用来比较两个类型是否相同：

     if (typeid(a) == typeid(b)) { ... } //可以用type_info.name()返回类型的名称
   - dynamic_cast操作符，将基类类型的指针或引用安全地转换为派生类型的指针或引用。

     Derived *derivedPtr = dynamic_cast<Derived *>(basePtr);

     Derived &d = dynamic_cast<Derived &>(b);

     当具有基类的引用或指针，但需要执行不是基类组成部分的派生类操作时，需要动态强制类型转换。只要有可能，定义和使用虚函数比直接接管类型管理好得多。
3. 类成员指针：成员指针只应用于类的非static成员，static成员指针是普通的指针。

   string Screen:: *p_screen; //定义了指向string类型的Screen类成员的指针
4. 使用类成员指针：.*和->*将成员指针绑定到实际对象。这两个操作符的左操作数必须是类类型的对象或类类型的指针，右操作数是该类型的成员指针。
   1. 使用成员函数指针：

      char (Screen::*pmf)() const = &Screen::get;

      Screen myScreen;

      char c1 = myScreen.get();

      char c2 = (myScreen.*pmf)();

      Screen *pScreen = &myScreen;

      c1 = pScreen->get();

      c2 = (pScreen->*pmf)();
   2. 使用数据成员的指针：

      Screen::index Screen::*pindex = &Screen::width;

      Screen myScreen;

      Screen::index c1 = myScreen.width;

      Screen::index c2 = myScreen.*pindex;

      Screen *pScreen = &myScreen;

      c1 = pScreen->width;

c2 = pScreen->*pindex;

5. 嵌套类：可以在一个类的内部另外定义一个类，这样的类是嵌套类，也称为嵌套类型。嵌套类最学用于定义执行类，如Queue中的QueueItem类。
    1. 嵌套类是独立的类，基本上与它们的外围类不相关，因此，外围类和嵌套类的对象是相互独立的。很明显，Queue对象并不包含QueueItem对象，QueueItem对象更不包含Queue对象。定义为嵌套类大部分是因为作用域的原因。例如QueueItem类不想被除了Queue对象外的任何对象所访问。
    2. 外围类对嵌套类的成员没有特殊访问权，嵌套类对其外围类也没有特殊访问权。
    3. 在外围类的public部分定义的嵌套类定义了可以在任何地方使用的类型，在外围类的protected部分定义的嵌套类定义了只能由外围类、友元或派生类访问的类型，在外围类的private部分定义的嵌套类定义了只能被外围类或其友元访问的类型。
       多数时候将嵌套类定义为private或protected的访问权限，否则定义嵌套类就换去了意义。
    4. 因为Queue是一个模板，所以它的成员也隐含地是模板，所以QueueItem隐含的是一个模板类。
    5. 在其类外部定义的嵌套类成员，必须定义在定义外围类的同一作用域。因为，嵌套类的成员不是外围类的成员。
       template <class Type>
       Queue<Type>::QueueItem::QueueItem(const Type &t) : item(t), next(0) {}
6. 联合[2]：是一种特殊的类。一个union对象可以有多个数据成员，但在任何时刻，只有一个成员可以有值。当将一个值赋给union对象的一个成员的时候，其它的所有的成员都变为未定义的。union不能具有定义了构造函数、析构函数或赋值操作符的类类型的成员。
7. **局部类：在函数体内部定义的类称为局部类**（区别局部类和嵌套类，局部类在函数体内定义，嵌套类在另一个类内定义）。与嵌套类不同，局部类的成员是严格受限的。
    1. 局部类的所有成员必须完全定义在类定义体内部。
    2. 不允许局部类声明static数据成员。
    3. 局部类可以访问的外围作用域中的名字是有限的。局部类只能访问在外围作用域中定义的类型名、static变量和枚举成员，不能使用定义该类的函数中的变量。
    4. 实际上，局部类中private成员几乎是不必要的，通常局部类的所有成员都应该为public。
8. 固有的不可移植的特征：位域、volatile限定符、链接指示
    1. 位域是一种特殊的类数据成员，用来保存特定的倍数。当程序需要将二进制数据传递给另一程序或硬件设备的时候，通常使用位域。
       位域必须是整形数据类型，可以是singed或unsigned，最好设置为unsigned，因为signed的行为标准未定义。通过在成员后面接一个冒号以及指定位数的常量表达式，指出成员是一

[2]疑问：联合不能继承，那么将联合指定为私有的或受保护的有什么意义呢？还有联合可以定义成员函数，包括构造函数和析构函数，这些都有什么用处呢？
原因在于：联合是可以继承的！联合很有用，和普通类有很多类似的地方！

个位域：

```
class File{
    unsigned int Mode: 2;          //指出这是一个2位的位域
}
```

2. volatile限定符：当可以用编译器的控制或检测之外的方式改变对象值的时候，应该将对象声明为volatile。关键字volatile是告诉编译器对这样的对象不应该执行优化。

   合成的复制控制不适用于volatile对象，如果类希望允许复制volatile对象，它必须自己定义复制构造函数和/或赋值操作符版本。

3. 链接指示：extern "C"。链接指示不能出现在类定义或函数定义的内部，它必须出现在函数的第一次声明上。

```
extern "C" size_t strlen( const char * );
extern "C" {
    int strcmp( const char *, const char * );
    char* strcat( char *, const char * );
    #include <string.h>          //C++标准库中使用C标准库的方法
}
```

   导出C++函数到其它语言：extern "C"/"Ada"/"FORTRAN" double calc(double dparm) { ... }

   用链接指示定义的函数的每个声明都必须使用相同中的链接指示。因为键接指示属于函数原型的一部分。（因为它决定了编译器mangled之后的真正名字！）

附录

标准库算法

1. 查找对象的算法：
    - 简单查找算法：
        - find (beg, end, val)
        - cout (beg, end, val)
        - find_if (beg, end, unaryPred)
        - cout_if (beg, end, unaryPred)
    - 查找许多值中的一个的算法
        - find_first_of (beg1, end1, beg2, end2)在第一个范围中查找与第二个范围中任意元素相等的第一个（或最后一个）元素。
        - find_first_of (beg1, end1, beg2, end2, binaryPred)
        - find_end(beg1, end1, beg2, end2)
        - find_end(beg1,end1, beg2, end2, binaryPred)
    - 查找子序列的算法
        - adjacent_find(beg, end)返回重复元素的第一个相邻对。如果没有相邻对的重复元素，就返回end。
        - adjacent_find(beg, end, binaryPred)
        - search(beg1, end1, beg2, end2)返回输入范围中第二个范围作为子序列出现的第一个位置。如果找不到子序列，就返回end。
        - search(beg1, end1, beg2, end2, binaryPred)
        - search_n(beg, end, count, val)返回count个相等元素的子串开关的迭代器。如果不存在这样的子串，就返回end。
        - search_n(beg, end, count, val, binaryPred)
2. 其它只读算法：
    - for_each(beg, end,f)对输入范围中的每个元素应该函数f，如果f有返回值，则忽略。
    - mismatch(beg1, end1, beg2)比较两个序列中的元素，返回一对表示第一个不匹配元素的迭代器。如果所有的元素都匹配，则返回的pair是end1，以及beg2中偏移量为第一个序列长度的迭代器。
    - mismatch(beg1, end1, beg2, binaryPred)
    - equal(beg1, end1, beg2)确定两个序列是否相等，如果输入范围中的第个元素都与beg2开始的序列中的对应元素相等，就返回true。
    - equal(beg1, end1, beg2, binaryPred)
3. 二分法查找算法：

- lower_bound(beg, end, val)返回第一个迭代器，可以将val插入而仍保持有序。
- lower_bound(beg, end, val, comp)
- upper_bound(beg, end, val)返回最后一个迭代器，可以将val插入而保持有序。
- upper_bound(beg, end, val, comp)
- equal_range(beg, end, val)返回一个表示子范围的迭代器，可以插入val还保持有序。
- equal_range(beg, end, val, comp)
- binary_search(beg, end, val)二分法搜索，如果x>y和y<x都不假，则x,y相等。
- binary_search(beg, end, val, comp)

4. 写容器的算法
   - 只写不读的算法：
     - fill_n(dest, cnt, val)将cnt个val值的副本写到dest上
     - generate_n(dest, cnt, Gen)
   - 使用输入迭代器写元素的算法：
     - copy(beg, end, dest)
     - transform(beg, end, dest, unaryOp)对输入范围中的每个元素应用指定操作
     - transform(beg, end, beg2, dest, binaryOp)
     - replace_copy(beg, end, dest, old_val, new_val)将每个元素复制到dest用new_val代替指定元素。
     - replace_copy(beg, end, dest, unaryPred, new_val)
     - merge(beg1, end1, beg2, end2, dest)两个序列都必须是已排序的，合并后写到dest。
     - merge(beg1, end1, beg2, end2, dest, comp)
   - 使用向前迭代器写元素的算法：
     - swap(elem1, elem2)
     - iter_swap(elem1, elem2)
     - swap_ranges(beg1, end1, beg2)
     - fill(beg, end, val) 将新值赋给序列中的每一个元素。
     - generate(beg, end, Gen)
     - replace(beg, end, old_val, new_val)
     - replace_if(beg, end, unaryPred, new_val)
   - 使用双向迭代器写元素的算法：
     - copy_backward(beg, end, dest)
     - inplace_merge(beg, mid, end)将同一序列中的两个相邻子序列合并为一个有序序列。
     - inplace_merge(beg, mid, end, comp)
5. 划分与排序算法：
   - 划分：（要求双向迭代器）

- stable_partition(beg, end, unaryPred)
- partition(beg, end, unaryPred)使用unaryPred划分序列，使unaryPred为真的放在序列的开头，为假的在序列的后面。返回指向为真的最后一个元素的下一位置的迭代器。

■ 排序：（要求随机访问迭代器）

- sort(beg, end)从小到大排序，调用“<”操作符。
- stable_sort(beg, end)
- sort(beg, end, comp)
- stable_sort(beg, end, comp)
- partial_sort(beg, mid, end)对mid-beg个元素进行排序，前面mid-beg个元素有序，后面的元素是无序的。
- partial_sort(beg, mid, end, comp)
- partial_sort_copy(beg, end, destBeg, destEnd)
- partial_sort_copy(beg, end, destBeg, destEnd, comp)
- nth_element(beg, nth, end)实参nth是一个迭代器，运行nth_element后，该迭代器表示的值就是：如果整个序列已排序，这个位置应该放置的值。nth之前的元素小于nrt，之后的元素大于nrt。
- nth_element(beg, end, end, comp)

6. 通用的重新排序操作：

■ 使用向前迭代器：

- remove(beg, end, val) 返回未移去的最后一个元素的下一位置
- remove(beg, end, unarryPred)
- unique(beg, end)
- unique(beg, end, binaryPred)
- rotate(beg, mid, end)围绕由mid表示的元素旋转元素

■ 使用双向迭代器：

- reverse(beg, end)颠倒序列中的元素。
- reverse_copy(beg, end, dest)

■ 写到输出迭代器：

- remove_copy(beg, end, dest, val)
- remove_copy_if(beg, end, dest, unarryPred)
- unique_copy(beg, end, dest)
- unique_copy(beg, end, dest, binaryPred)
- rotate_copy(beg, mid, end, dest)

■ 使用随机访问迭代器：

- random_shuffle(beg, end)打乱序列中的元素

- random_shuffle(beg, end, rand)该rand函数必须返回迭代器的difference_type类型的值。

7. 排列算法：（返回值与是否是最大或最小序列有关）
   - next_permutation(beg, end)
   - next_permutation(beg, end, comp)
   - prev_permutation(beg, end)
   - prev_permutation(beg, end, comp)
8. 有序序列的集合算法：
   - includes(beg, end, beg2, end2)如果输入序列包含第二个序列中的每一个元素，就返回true，否则返回false。
   - includes(beg, end, beg2, end2, comp)
   - set_union(beg, end, beg2, end2, dest)
   - set_union(beg, end, beg2, end2, dest, comp)
   - set_intersection(beg, end, beg2, end2, dest)交集元素的有序序列
   - set_intersection(beg, end, beg2, end2, dest, comp)
   - set_difference(beg, end, beg2, end2, dest)在第一个容器中但不在第二个容器中的元素的有序序列。
   - set_difference(beg, end, beg2, end2, dest, comp)
   - set_symmetric_difference(beg, end, beg2, end2, dest)创建在任一容器中存在，但不在两个容器中同时存在的元素的有序序列。
   - set_symmetric_difference(beg, end, beg2, end2, dest, comp)
9. 最大值和最小值：
   - min(val1, val2)
   - min(val1, val2, comp)
   - max(val1, val2)
   - max(val1, val2, comp)
   - 这四个算法都有其迭代器的版本，如min_element(beg, end, comp)
   - lexicographical_compare(beg1, end1, beg2, end2)按字典顺序进行比较。
   - lexicographical_compare(beg1, end1, beg2, end2, comp)
10. 算术算法：
    - accumulate(beg, end, init)累加，init为初始
    - accumulate(beg, end, init, BinaryOp)
    - inner_product(beg1, end, beg2, init)返回两个序列乘积而生成的元素的总和
    - inner_product(beg1, end, beg2, init, BinOp1, BinOp2) Op1代替+，Op2代替*
    - partial_sum(beg, end, dest)将新序列写到dest，其中每个新元素的值表示输入范围中在它的位置之前（不包括）的所有元素的总和。

- partial_sum(beg, end, dest, BinaryOp)
- adjacent_difference(beg, end, dest)将新序列写到dest，除第一个元素外，其它元素都是当前元素与前一元素的差。
- adjacent_difference(beg, end, dest, BinaryOp)

其它部分

1. I/O格式控制：(#include <iomanip>)
   - cout.precision()返回当前的显示精度，或者通过cout.precision(5)设置显示精度。
   - setw(12)指定下次输出占用12个宽度。
   - internal左对齐符号且右对齐值，用空格填充介于其间的空间。
   - setfill指定填充输出时所有的字符，默认是空格。
2. 为什么多数程序喜欢即使不修改参数的情况下也传递引用呢？因为传递引用不用进行复制操作，直接传值复制的开销很大，特别是容器。这样提高了程序的效率！
3. 大多数时候，带头结点的键表都比不带头结点的键表好用。不带头结点的表就是建起来方便一些，用的时候几乎都不如带头结点的。
4. VC++中其实也是可以用描述的

   ```
   struct my_class{
     /* 在这里描述 */
     int value;   /* 或者在这里描述 */   };
   ```

   这个描述的内容会在智能提示中出现的
5. 数学库函数：#include <cmath>;
   - ceil(x) 求x的整数部分，使其不小于x的最小整数。
   - floor(x) 求x的整数部分，使其不大于x的最大整数。
   - fmod(x) 求x/y的浮点余数。
   - pow(x,y) 求x的y次方。
   - sqrt(x) 求x的平方根。
6. 存储类说明符：auto, register, extern, mutable, static。这些说明符决定了标识符在内存中的存在的期限。局部变量默认为auto；常用的变量可以声明为register将变量保存在寄存器中；extern和static用以声明变更和静态存储类函数的说明符，全局变量和函数名默认为extern。