迭代器，下标运算符都是返回的元素的引用。

c++->服务器开发 （BOOST STL  tcp/ip FTP HTTP 数据库 ）  
c++->Linux 应用开发

第三章：字符串、向量和数组

1．定义和初始化string对象

String s1;//默认初始化，s1是一个空字符串

String s2=s1;//s2是s1的副本

String s2（s1）;

String s3=”string”;//s3是字符串字面值的副本，除了最后的\n全部到被拷贝到s3中

String s3（”string”）;

String s4（10，’c’）;//s4的内容是cccccccccc。

使用”=”初始化的方式为拷贝初始化。如果不使用”=”则是直接初始化。

2.string操作

Getline(is,s);//从输入流读取一行赋值给s，返回is。

s.empty();//s为空返回true。否则为false

s.size();//返回字符串s中的字符个数。返回值类型为string::size\_type。允许编译器用auto推断。

S[n];//返回s中第n个字符的引用，位置n从0计起。

S1+s2;//返回s1和s2连接后的结果，一个临时的string变量。

例子：string s1(“hello”);string s2(“world”);

string s3=s1+”,”+s2+”\n”;

当把string对象和字符字面值或字符串字面值混在一条语句中使用时，必须确保每个加法运算符（+）的两侧运算对象至少有一个是string

例子：

String s5=”hello”+”,”;//错误，两个运算对象都不是string

String s6=s1+”,”+”world”;//正确，s1+”,”返回一个临时的string变量。

S1=s2;//用s2的副本覆盖s1中原来的字符

S1==s2;//如果s1和s2中所含的字符完全一样，则它们相等，string对象的相等性判断对字母的大小写敏感。

S1!=s2;//s1!=s2返回true

3.处理string对象中的字符#include<cctype>

isalnum( c )//当c是字母或数字时为真

isalpha( c )//当c是字母时为真

iscntrl( c)//当c是控制字符时为真

isdigit( c)//当c是数字时为真

isgraph( c)//当c不是空格但可打印时为真

islower( c)//当c是小写字母时为真

isprint( c)//当c是可打印字符时为真（即c是空格或者c具有可视形式）

ispunct( c)//当c是标点符号时为真

isspace( c)//当c是空白是为真（即c是空格，横向制表符，纵向制表符，回车符，换行符）

isupper( c)//当c是大写字母时为真

isxdigit( c)//当c是十六进制数字时为真

tolower( c)//如果c是大写字母输出对应的小写字母

toupper( c)//如果c是小写字母，输出对应的大写字母

c++11版本定义了新的遍历string的for循环格式

for(auto i:string){

cout<<i<<” “;

}

4.处理一部分字符string同时支持迭代器和下标运算符

要想访问string对象中的单个字符有两种方法，一种是使用下标，另外一种是使用迭代器

下标运算符( [] )接收的输入参数是string::size\_type类型的值，这个参数表示要访问的字符的位置，返回值是该位置上字符的引用。

String对象的下标从0开始计起，如果string对象s至少包含两个字符，则s[0]是第一个字符，s[1]是第二个字符，s[s.size()-1]

5．数组

5.1字符数组的特殊性

Char a1[]={‘C’, ‘+’, ‘+’, ‘\0’};//列表初始化，没有空字符

Char a2[]={‘C’, ‘+’, ‘+’, ‘\0’};//列表初始化，含有显式的空字符

Char a3[]=”C++”;//自动添加表示字符串结束的空字符

Const char a4[6]=”Daniel”;//错误，没有空间可存放空字符

6.数组array

和内置类型的变量一样，如果在函数内部定义了某种内置类型的数组，那么默认初始化会令数组含有未定义的值。

6.1数组和指针

使用数组名时，编译器会自动将它转换成指向数组中首元素的的指针。

String name[]={“one”,”two”,”three”};

String\* p=name;//p是一个string类型的指针，指向name[0]。

但是，当使用decltype关键字时上述转换不会发生

Int ia[]={0,1,2,3,4,5,6,7,8,9};

Decltype(ia) ia1={0,1,2,3,4,5,6,7,8,9};

6.2指针也是迭代器

指向数组元素的指针拥有更多功能，vector和string的迭代器支持的运算，数组的指针全都支持。

Int arr[]={0,1,2,3,4,5,6,7,8,9};

Omt \* p=arr;//p指向arr[0]

++p;//p指向arr[1]

可以使用标准库函数begin和end获取数组首元素的指针或者数组指向数组微元素下一位置的指针。

Int\* beg=begin（arr）；

Int\* end=end（arr）；

6.3指针运算

指向数组元素的指针可以执行所有迭代器的运算，包括解引用，递增，比较，与整数相加，两个指针相减等。

Int arr[]={1,2,3,4,5};

Int\* ip=arr;

Int\* ip1=ip+4;//ip1指向arr的尾元素arr[4]

6.4 下标和指针

在很多情况下使用数组的名字其实用的是一个指向数组首元素的指针，一个典型的例子是当对数组使用下标运算符时，编译器会自动执行上述转换过程。

Int ia[]

Auto n=end（arr）-begin（arr）；//n的值为5，也就是arr中元素的数量。

第九章：顺序容器

1.vector:可变大小数组

deque:双端队列

list：双向链表

forward\_list:单向链表

array：固定大小数组

string：与vector相似的容器，但专门保存字符，

vector、string将元素保存在连续的内存空间中。

list、forward\_list在任何位置插入都很快，但不支持随机访问，必须遍历所有元素。

vector是首选的，应尽量避免在中间位置插入数据，可以在尾部插入，再利用算法重新排序，

或者先在输入阶段使用list，之后将数据拷贝到vector中。

2.容器的初始化：

C c；

C c1（c2）//或者C c1=c2；c1和c2必须为相同类型-相同容器且保存相同类型元素

C c（b，e）；//范围中的元素类型必须与C的元素类型相容。

C c{a,b,c...}//列表初始化c。

3.array

与内置数组一样，array的大小也是类型的一部分，定义一个array时，除了指定元素类型，还要指定容器大小

array<int,42>;保存42个int类型的元素

array<string,10>；保存10个string类型的元素

array<int,10>ia={42};ia[0]为42，其余为0；

与内置数组不同的是，array允许拷贝，只要大小和元素类型相同就可以

array<int,10>digits{0,1,2,3,4,5,6,7,8,9}；

array<int,10>copy=digits；

copy={0}；//错误，不能将花括号列表赋给一个数组

4.赋值和swap

swap（c1，c2）；//交换c1，c2中的元素，必须具有相同的类型，swap通常比从c2向c1拷贝要快得多。

seq.assign(b,e);//将seq的元素替换为b到e迭代器间的元素

seq.assign(il);//将seq中的元素替换为初始化列表il中的元素

seq.assign(n,t);//将seq中的元素替换为n个值为t的元素

5.拷贝一个元素

可以初始化时利用C c（b，e）完成

可以通过swap

可以通过assign。和C c（b，e）类似，容器类型相容即可

可以通过泛型算法copy 例子：

copy(ivec.begin(),ivec.end(),inserter(lis,lis.begin()));

copy(ivec.begin(),ivec.end(),back\_inserter(lis));

copy接收三个迭代器，前两个表示输入范围，第三个表示目的序列的起始位置，算法将输入范围中的元素拷贝的目的序列。

6.

第十章：泛型算法

#include<algorithm>

#include<numeric>

注意：所有只接受一个单一迭代器来表示第二个序列的算法都假定第二个序列至少与第一个序列一样长。

注意可调用对象的定义，之前学习的可调用对象仅有函数和函数指针。

lambda可以理解为一个未命名的内联函数。

1.lambda函数[捕获列表](参数列表)->返回类型{函数体}

我们只对lambda所在函数定义的非static变量使用捕获列表··，lambda可以直接使用局部static变量和在他所在函数之外声明的名字

2.标准库bind函数

#include<functional>

命名空间placeholders

using namespace std::placeholders

auto newCallable=bind(callable,arg\_list)

newCallable本身是一个可调用对象，arg\_list是一个逗号分隔的参数列表，对应callable的参数，当我们调用newCallable时，newCallable会

调用callabe并将参数arg\_list传递给它。

arg\_list中可能包含占位符\_1,\_2,类似这样的占位符，占位符定义域placeholders的命名空间内，例如：

using namespace std::placeholders;

auto check6=bind（check\_size，\_1,6);

string s="hello";

check6(s);//调用(check\_size(s,6));

使用bind可改写原来基于lambda的find\_if算法调用

原本：auto wc=find\_if(words.begin(），words.end(),[sz](const string& a){return a.size()>=sz};

替换为：auto wc=find\_if(words.begin(),words.end(),bind(check\_size,\_1,sz);

3.再探迭代器

(1）插入迭代器：

back\_inserter::auto it=back\_inserter(ivec);插入之后it指向的还是尾后

front\_inserter::auto it=front\_inserter(ivec);插入之后，it指向的还是第一个元素。

inserter: auto it=inserter(ivec,ivec.begin());插入之后it自动++回到插入前的元素位置，例如：

\*it=val;等价于：it=ivec.insert(it,val);++it;

在容器间互相拷贝

vector<int>ivec{1，2,3,4,5,}；

list<int>lis;

copy(ivec.cbegin(),ivec.cend(),inserter(lis,lis.begin());

（2）流迭代器:p359

istream\_iterator:读取输入流

ostream\_iterator:向一个输出流写数据 ostream\_iterator<int>out(cout," "); for(auto i:vec){out=i;} //这里\*out=out，\*out不做任何事

p360用一对表示元素范围的迭代器来构造vec。

istream\_iterator<T> in（）

第十一章 关联容器--关键字是const的

key\_type：关键字类型

mapped\_type:关键字关联的类型，只有map有

value\_type：对于set与key\_type相同

对于map为pair<key\_type,mapped\_type>

1. 有序容器中，关键字定义为严格弱序，可以将严格弱序看做小于等于

（1）两个关键字不能同时小于等于对方，如果k1小于等于k2，那么k2绝不能小于等于k1

（2）如果k1小于等于k2，且看小于等于k3，则k1必须小于等于k3

（3）如果存在两个关键字，任何一个都不小于等于另一个，那么我们称这两个关键字是等价的，如果k1等价于k2，且k2等价于k3，

则k1等价于k3.

如果两个关键字是等价的，那么容器将他们视为相等来处理，当用作map的关键字时，只能有一个元素与这两个关键字关联，我们可以

用着两者中任意一个来访问对应的值。

2. 组织容器中元素的操作的类型也是容器类型的一部分，要使用自定义的操作，必须在定义关联容器类型时提供此操作（<）的类型。例如：

我们不能直接定义一个Sales\_data类的multiset，因为Sales\_data没有(<)运算符，但是我们可以利用compareIsbn函数定义multiset。

bool compareIsbn(const Sales\_data& ihs,const Sales\_data& rhs){

return ihs.isbn()<rhs.isbn();

}

为了使用自己定义的<操作，在定义multiset时必须提供两个类型，关键字类型Sales\_data以及比较操作--这应该是一个函数指针类型。

multiset<Sales\_data,decltype(compareIsbn)\*>bookstore（compareIsbn）；

3. pair类型

定义在头文件utility中，#include<utility>

一个pair保存两个数据成员，类似容器，pair是一个用来生成特定类型的模板，创建一个pair时，要提供两个类型名，两个类型不要求一样。

（1）初始化

pair<string,string>anon;//两个空string

pair<string,size\_t>word\_count;使用了默认构造函数对数据成员进行初始化//一个空string，一个0

也可以提供初始化器

pair<string,string>author{"james","joyce"};

（2）pair的数据成员是public的，一个命名为first一个命名为second。可以用普通的成员函数访问，author.first,author.second

（3）初始化方式：

pair<T1,T2>p;

pair<T1,T2>p(v1,v2);

pair<T1,T2>p={v1,v2};

make\_pair(v1,v2);返回一个有v1和v2初始化的pair，pair的类型从v1和v2推断。

4.新标准下，我们可以对返回值进行列表初始化

pair<string,int> process(vector<string> &v){

if(!v.empty())

return {v.back(),v.back().size()};

else

return pair<string,int>();

}

5.解引用关联容器迭代器时，得到一个类型为容器的value\_type的值的引用。对map而言是一个pair类型，对于set来说是关键字。

p382

auto map\_it=word\_count.begin();

cout<<map\_it->first//得到此元素的关键字

cout<<map\_it->second//得到此元素的值

map\_it->first="new key"//错误，关键字都是const的。

6.map和set的insert有两个版本

（1）vector<int>ivec{2,4,6,8,2,4,6,8};

set<int>set1;

set1.insert(ivec.begin(),ivec.end());//有4个元素

set1.insert({1,3,5,7,1,3,5,7});//现在有8个元素

结论：插入重复元素对容器没有影响。

（2）map

word\_count.insert({word,1});

word\_count.insert(make\_pair(word,1));

word\_count.insert(pair<string,size\_t>(word,1));

word\_count.insert(map<string,size\_t>::value\_type(word,1));

7.关联容器定义三个版本的erase，通过传递一个迭代器或者迭代器对，另外还有一个额外的操作：

接收一个key\_type

if(word\_count.erase(removal\_word))

cout<<"ok: "<<removal\_word<<" removed\n"

对于不重复关键字容器，erase的返回值只有0或者1，,0代表要删除的元素不再容器中

对于重复关键字的容器，删除元素的数量可能大于1：

auto cnt=authors.erase("Barth,John");p386创建的multimap，则cnt为2。

c.erase(key\_type)//返回删除元素的数量

c.erase(p);//返回删除元素后的元素的迭代器

c.erase(b,e)//删除b，e范围内的元素，返回e。

8.访问元素

使用c.find（k）返回一个迭代器，指向第一个关键字为k的元素

使用c.count(k) 返回关键字等于k的元素的数量，若为不重复关键字元素，返回值只有0或者1

使用c.lower\_bound(k)返回一个迭代器，指向第一个关键字不小于k的元素

使用c.upper\_bound(k)返回一个迭代器，指向第一个关键字不大于k的元素

使用c.equal\_range(k)返回一个迭代器pair，表示关键字等于k的元素的范围，若k不存在，pai的两个成员均等于c.end().

**第十二章 动态内存**

1.静态内存用来保存局部static对象，类static数据成员，以及定义在任何函数之外的变量。

栈内存用来保存定义在函数内的非static对象。

这些内存中的对象由编译器自动创建和销毁，栈对象仅在定义的程序块运行时才存在，static对象在使用之前分配在程序结束时销毁。

2.除了静态内存和栈内存，每个程序还拥有一个内存池--称为自由空间或“堆”。

程序用堆来存储动态分配的对象-即对象的生存期由程序来控制，当动态对象不再使用时，代码必须显示销毁它们。

3.c++中动态内存由一堆运算符完成

new：在动态内存中为对象分配空间并返回一个指向该对象的内置指针。

delete：接收一个动态对象的指针，销毁该对象，并释放与之关联的内存。

内存泄漏：忘记释放动态内存会造成内存泄漏。

引用非法内存的指针：尚有指针引用内存的情况下释放内存，会产生引用非法内存的指针。

4.智能指针也是模板，实例化时要显示声明模板参数：shared\_ptr<int>p1;

解引用一个智能指针返回一个它所指向的对象。

5.shared\_ptr<T>p;

(1)p作条件判断，若p指向一个对象则为true

（2）\*p获得它指向的对象

（3）p.get（）返回p中保存的指针，小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了。

（4）make\_shared<T>（args）返回一个shared\_ptr，指向一个动态分配的类型为T的对象，使用args初始化此对象。

（5）p.unique（）若p.use\_count（）为1，则为true。

6.make\_shared函数

此函数在动态内存中分配一个对象并初始化它，返回指向此对象的shared\_ptr。

shared\_ptr<int>p=make\_shared<int>(3)；

7.每个shared\_ptr都有一个关联的计数器，被称为“引用计数”。当拷贝一个shared\_ptr指针时，计数器会递增。例如：

（1）用shared\_ptr初始化另一个shared\_ptr时。

（2）将他作为参数传递给一个函数(传参时自动被拷贝)

（3）作为函数的返回值时。（自动被拷贝）

计数器递减:

（1）给shared\_ptr赋予一个新值

（2）shared\_ptr被销毁，比如离开其作用域。

shared\_ptr的析构函数会递减它所指向的对象的引用计数。

8.使用new动态分配内存和初始化对象

使用传统的构造方式（圆括号）：

string\* ps=new string(10,'c'）；

int\* p=new int(42);

使用列表初始化

vector<int>\* pv=new vector<int>{0,1,2,3,4,5,6,7,8,9};

使用值初始化：后跟一空括号

string\* ps=new string();

默认初始化与值初始化不同。默认初始化无括号。

9.delete

delete p;//p必须指向一个动态分配的对象或是一个空指针。

与new类似，delete执行两个动作，1.销毁给定的指针指向的对象，2.释放对应的内存。

10.p411空悬指针

即，指向一块曾经保存数据对象，但现在已经无效的内存。

11.shared\_ptr与new结合使用

shared\_ptr<int>p(new int(42));//正确

shared\_ptr<int>p1=new int(1024);//错误，必须使用直接初始化形式

shared\_ptr<int> clone(int p){

return new int （p）;

}//错误，不能将一个内置指针隐式的转换为一个智能指针。

shared\_ptr<int> clone(int p){

return shared\_ptr<int>(new int(p));//正确

}

P412表12.3要记住，其中：

shared\_ptr<T>p(q)//p管理内置指针q所指向的对象，q必须指向new分配的内存，且能够转换为T\*。

shared\_ptr<T>p(q，d)//p接管了q所指向对象的所有权，且p将调用可调用对象d来代替delete。

12.其他shared\_ptr操作

使用reset使shared\_ptr指向一个新内存（将新指针赋值给此shared\_ptr）

shared\_ptr p=new int（1024）；//错误，不能将你内置指针赋予shared\_ptr

p.reset(new int(1024));//正确，p指向一个新对象。

与赋值类似，reset会更新引用计数，如果需要的话，会释放p指向的对象，reset成员经常与unique一起使用，来控制多个shared\_ptr共享的对象,在改变底层对象之前，我们检查自己是不是当前对象仅有的用户，如果不是，在改变前要制作一份新的拷贝

if(!p.unique())

p.reset(new string(\*p））；//不是仅有的用户复制一份拷贝

\*p+=newVal;

13.p416页使用我们自己的释放操作。

14.unique\_ptr--注意与shared\_ptr的区别

（1）unique\_ptr不支持普通的拷贝和赋值操作

unique\_ptr<string>p1(new string("string"));//正确

unique\_ptr<string>p2(p1)//错误，不支持拷贝操作

unique\_ptr<string>p3;

p3=p1//错误，unique\_ptr不支持赋值

（2）

unique\_ptr<T> u1//空unique\_ptr，可以指向类型为T的对象，u1会使用delete来释放它的指针。

unique\_ptr<T,D>u2//u2使用D的可调用对象来释放它的指针。

unique\_ptr<T,D>u(d)//空unique\_ptr，用类型为D的对象d来代替delete。

u=nullptr//释放u指向的对象，将u置空

u=release()//u放弃对指针的控制权，返回指针，并将u置空

u.reset();

u.reset(q);

15.new和数组

int \*pia= new int[get\_size()]//pia指向第一个int

也可以用一个表示数组类型的类型别名来分配一个数组，这样，new表达式中就不需要方括号了

typedef int arrT[42];//arrT表示42个int的数组类型

int\* p=new arrT;//分配一个42个int的数组，p指向第一个int。

动态数组并不是数组类型！！！！！返回的只是一个元素类型的指针。

（1）初始化动态分配对象的数组

int\* pia=new int[10];//10个未初始化的int

int\* pia1=new int[10]();//10个值初始化为0的int。

新标准中，我们还可以提供一个元素初始化器的花括号列表

int\* pia2=new int[10]{0,1,2,3,4,5,6,7,8,9};

（2）释放动态数组

为了释放动态数组，我们使用一种特殊形式的delete--在指针前加上一个空方括号对。

delete []pa;//销毁pa指向的数组中的元素，并释放对应的内存。元素按逆序销毁。

※p426，使用动态指针与动态数组绑定时，unique\_ptr与shared\_ptr不同。

unique\_ptr：

unique\_ptr<int[]> up(new int[10]);

up.release()//自动调用delete []删除其指针。！！！！与shared\_ptr的区别

for(size\_t i=0;i!=10;++i){

up[i]=i;

}

shared\_ptr:

shared\_ptr不直接支持管理动态数组，需提供一个删除器

shared\_ptr<int> sp(new int[10],[](int \*p){delete []p;}

sp.reset();

for(size\_t i=0;i!=10;++i)

\*（sp.get（）+i)=i;

16.allocator类 定义在头文件memory中。

当分配一大块内存时，我们通常计划在这块内存上按需构造对象，此时我们希望将内存分配和对象构造分离。

它分配的内存时原始的，未构造的，类似vector，allocator也是模板。为了定义allocator对象，我们必须指明这个allocator

可以分配的对象类型。当allocator对象内存时，它会根据给定的对象类型来确定恰当的内存大小和对齐位置。

allocator<string>alloc;//可以分配string的allocator对象。

auto const p=alloc.allocate(n);//分配n个未初始化的string。

这个allocate调用为n个string分配了内存。

第十三章 拷贝控制

问题：explicit问题？

1.拷贝构造函数，拷贝赋值运算符，析构函数。

拷贝构造函数的第一个参数必须是自身类类型的引用，额外参数必须有默认值。

拷贝初始化发生条件：

1. 使用“=”初始化
2. 将一个对象作为实参传递给一个非引用类型的形参
3. 从一个返回类型为非引用类型的函数返回一个对象
4. 用花括号列表初始化一个数组中的元素或一个聚合类中的成员。

2.std::move(s);将s转换为右值。

右值引用就是绑定到右值的引用，通过&&来获得右值引用。

右值引用有一个重要的性质-只能绑定到一个将要销毁的对象，因此，我们可以自由地将一个右值引用的资源移动到另一个对象中。

同左值引用一样，右值引用也是某个对象的另一个名字，我们可以将右值引用绑定到要求转换的表达式，字面常量或是返回右值的表达式。

3.int i=42；

int &r=i;

int &&rr=i//错误，不能右值引用绑定到一个左值上

int &r2=i\*42 //错误，i\*42是一个右值

const int &r3 =i\*42；正确，可以将const的引用绑定到一个右值上。

int &&rr2=i\*42; //正确

返回左值引用的函数，以及赋值，下标，解引用和前置递增/递减运算符，都是返回左值的表达式的例子。可以将左值引用绑定到这类表达式的结果上。

返回非引用类型的函数，联同算数，关系，位，以及后置递增/递减运算符，都生成右值，我们不能将一个左值引用绑定到这类表达式上，但我们可以将一个const的左值引用或者一个右值引用绑定到这类表达式上。

3.1

左值持久，右值短暂：右值要么是字面常量要么是表达式求职过程中创建的临时对象。

因为右值引用只能绑定到临时对象，可以得出：

1. 所引用的对象将要被销毁
2. 该对象没有其他用户

3.2变量是左值

Int &&rr1=42；//正确

Int &&rr2=rr1；//错误，表达式rr1是左值。

类似于其他表达式，变量表达式也有左值/右值属性，变量表达式都是左值，带来的结果就是，我们不能将右值引用绑定到一个右值引用类型的变量上。

3.3标准库move函数

虽然我们不能将右值引用绑定到一个左值上，但我们可以显式的将一个左值转换为对应的右值引用类型，我们还可以通过调用一个名为move的函数来获得一个绑定到左值上的右值引用，此函数定义在utility头文件中，

Int &&rr3=std::move(rr1);//正确

Move调用告诉编译器，我们有一个左值，但是我们希望像一个右值一样处理它，我们必须意识到：调用move就是承诺，除了对rr1赋值或销毁它外，我们将不再使用它。我们不能对移后源对象的值做任何假设。

4.p472标准库move函数p610页介绍move函数模板。

p478页介绍的HasPtr rhs是关键。

5.p483页没看懂。

6.析构函数：

在构造函数中，成员的初始化实在函数体执行之前完成的且按照他们在类中出现的顺序初始化。

但在析构函数中，先执行函数体再销毁成员，成员按初始化的逆序销毁。

内置类型指针的析构：

隐式销毁一个内置指针类型的成员不会delete他指向的对象。

7.什么时候调用析构：

1）变量离开作用域

2）一个对象被销毁时，其成员被销毁。

3）容器被销毁时，其元素被销毁

4）对于动态分配的对象，当对指向他的指针应用delete运算符时被销毁

5）对于临时对象，当创建它的完整表达式结束时被销毁

8.标准库容器，shared\_ptr和string既支持拷贝又支持移动，IO类和unique\_ptr不支持拷贝只支持移动。

9.

第十五章: 面向对象程序设计

错题：#include<fstream>

#include<sstream>

#include<string>

#include<vector>

#include<algorithm>

#include<utility>

using namespace std;

class base{

friend void print(const base& a);

public:

base()=default;

base(int ID,string NAME):id(ID),name(NAME){}

virtual void show()const{

cout<<"base"<<endl;

}

virtual ~base()=default;

protected:

int id;

private:

string name;

};

class sun:public base{

friend void print(const base &);

public:

sun()=default;

sun(int ID,string NAME):base(ID,NAME){}

~sun()=default;

void mem(){

cout<<id<<endl;

}

virtual void show()const override{

cout<<"sun"<<endl;

}

};

void print(const base &a){//这边是对象是const的，所以调用的函数不能改变调用对象，所以show函数也要是const的。

a.show();

}

int main(){

base bas(111,"lala");

sun son(1124,"sus");

print(son);

print(bas);

// son.mem();

// son.name;

}

只有当show函数的调用对象满足是const时才可以编译。

面向对象程序设计（OOP）的核心思想是数据抽象（类），继承和动态绑定（多态）。

1.只有公有派生类型的对象可以绑定到基类的引用或指针上。不然会提示[Error] 'base' is an inaccessible base of 'sun'。

2.派生类成员不能访问基类的private成员。

3.进一步思考我们可以得出，派生访问说明符其实定义的是，在派生类将基类的成员继承过来之后，这些成员在D中的新的访问权限。具体地说就是，如果以public来继承B，则B中成员的访问权限在D中具有相同的访问权限；如果以protected来继承B，则D中成员的访问权限的变化是public成员改变成protected成员，其余不变；如果以private来继承B，则D中的所有成员都变成了private成员。如果按照成员在基类B中的访问权限来划分的话，private成员不管怎么被继承，依然是private的；protected在private继承下，会变成private成员；而public成员在protected继承下会变成protected成员，在private继承下会变成private成员。

4.构造函数的成员初始化顺序要和定义顺序一致。

5.override加在const的后面：double net\_price(size\_t)const override;

6.因为在派生类对象中含有与其基类对应的组成部分，所以我们能把派生类的对象当成基类的对象来使用，而且我们也能将基类的指针

或引用绑定到派生类对象中的基类部分上。

这种隐式特性以为这我们可以把派生类对象或者派生类对象的引用用在需要基类引用的地方。同样的，我们也可以把派生类对象的指

针用在需要基类指针的地方。

7.派生类的作用域嵌套在基类的作用域内。

8.如果基类定义了一个静态成员，则在整个继承体系中只存在该成员的唯一定义。无论从基类派生出多少个派生类，对于每个静态成员

来说只存在唯一的实例。

9.c++11新标准提出来了一个防止继承发生的方法，在类名后面跟一个关键字final：

class NoDerived final{}；

class Bad：NoDerived{}//错误，NoDerived是final的不能被继承。

10.类型转换与继承--基类和派生类之间的类型转换是c++面向对象编程的关键所在

我们可以将基类的指针或引用绑定到派生类对象上隐藏一层含义：当使用基类的引用或指针时，实际上我们不清楚该引用或指针所绑

定对象的真实类型，可能是基类的对象，也可能是派生类的对象。p530解释。p534只能指针类也支持派生类向基类的类型转换，意味着

我们可以将一个派生类对象的指针存储在一个基类的智能指针内。

11.p535在基类构造函数中传递一个派生类的对象，会导致错派生类的部分被切掉，处理的还是基类部分的成员。

12.因为我们只有在运行时才知道调用了虚函数的那个版本，所以所有虚函数都必须有定义。

13.派生类中的虚函数形参类型和返回类型必须与基函数中的虚函数一致，但是有个例外p537，当基类的虚函数的返回类型是类本身的指针

或引用时，上述规则无效，即，如果D由B派生得到，则基类的虚函数可以返回B\*，而派生类的对应函数可以返回D\*，但要求D到B的类型转

换是可访问的。

14.p538，当我们使用override时，如果派生类的函数没有覆盖掉基类中的版本，系统将报错。

我们也可以使用final来禁止派生类覆盖该函数。。

struct D2:B{

void f1(int) const final;//不允许后续的其他派生类覆盖fi（int）

};

struct D3：D2{

void f2();

void f1(int) const;//错误，已经将f1（int）声明成final

};

15.p542子对象这个概念要搞清楚，每个类都包含自己类的子对象及它的基类的子对象。

16.p543派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象中的受保护成员没有访问权限。

class Base{

friend void clobber(Base&);

protected:

int prot\_mem;

};

class sneeky:public Base{

friend void clobber(sneeky&);

int j;

};

void clobber(sneeky& s){

s.j=s.prot\_mem=0;

}

void clobber(Base& s){

s.prot\_mem=0;

}

int main(){

}

17.p544派生类向基类转换的可访问性。

（1）只有当D公有地继承B时，用户代码才能使用派生类向基类的转换

（2）不论D以什么方式继承B，D的成员函数和友元都能使用派生类向基类的转换。派生类向其直接基类的类型转换对于派生类的成员和

友元来说永远是可访问的。

（3）如果D继承B的方式有公有的或者受保护的，则D的派生类的成员和友元可以使用D向B的类型转换，反之，如果D继承B的方式是私有

的，则不能使用。

18.就像友元关系不能传递一样，友元关系也不能继承，基类的友元在访问派生类成员是不具有特殊性，类似的派生类的友元也不能随意

访问基类的成员，类似上面的例子。p545。

class Base{

friend class pal;

protected:

int prot\_mem;

};

class sneeky:public Base{

friend void clobber(sneeky&);

friend void clobber(Base&);

int j;

};

void clobber(sneeky& s){

s.j=s.prot\_mem=0;

}

void clobber(Base& s){

s.prot\_mem=0;

}

class pal{

public:

int f(Base b){return b.prot\_mem;}

int f2(sneeky s){return s.j;}//错误，pal不是sneeky的友元，不可访问sneeky的private成员。

int f3(sneeky s){return s.prot\_mem;}//正确，对基类的访问权限由基类本身控制，即使对于派生类的基类部分也是如此。

};

如前所述，每个类负责控制自己的成员的访问权限，尽管看起来有点奇怪，但f3确实是正确的，pal是Base的友元，所以pal能够访问

Base对象的成员，这种可访问性包括了Base对象内嵌在其派生类对象中的情况。

int main(){

}

19.改变个别成员的可访问性

class Base{

public:

size\_t size()const {return n;}

protected:

size\_t n;

};

class Derived:private Base{

public:

using Base::size;

protected:

using Base::n;

};

因为Derived类使用了私有继承，所以继承而来的成员size和n默认情况下是Derived的私有成员。然而我们使用using声明语句改变了这些成员的可访问性，改变之后，Derived的用户将可以使用size成员，而Derived的派生类将能使用n。

using声明语句中名字的访问权限由该using声明语句之前的访问说明符来决定。也就是说，如果一条using声明语句出现在类的private部分，则该名字只能被类的成员和友元访问，如果using声明语句位于public部分，则类的所有用户都能访问它，如果using声明语句位于protected部分，则该名字对于成员、友元和派生类是可访问的。

20.继承中的类作用域

每个类定义自己的作用域，在这个作用域内我们定义类的成员，当存在继承关系时，派生类的作用域嵌套在其基类的作用域之内，如果一个名字在派生类的作用域内无法正确解析，则编译器将继续在外层的基类作用域中寻找该名字的定义。所以派生类才可以像使用自己的成员一样使用基类的成员。

Bulk\_quote bulk;

cout<<bulk.isbn();

名字isbn的解析按照下述过程所示：见p254，首先编译成员的声明，知道类全部可见后才编译函数体。

（1）因为我们是通过Bulk\_quote的对象调用isbn的，所以首先在Bulk\_quote中查找，这一步没有找到名字isbn。

（2）因为Bulk\_quote是Disc\_quote的派生类，所以接下来在Disc\_quote中查找。仍然没找到。

（3）因为Disc\_quote是Quote的派生类，所以接着查找Quote，此时找到了名字isbn，所以我们使用的isbn最终被解析为Quote中的isbn。

例子：p548

class Disc\_quote:public Quote{

public:

pair<size\_t,double>discount\_policy()const{return {quantity,discount};}

};

Bulk\_quote bulk;

Bulk\_quote\* bulkp=&bulk;

Quote\* itemp=&bulk;

bulkp->discount\_policy();//正确，bulkp的类型是Bulk\_quote\*

itemp->discount\_policy();//错误，itemp的类型是Quote\*；

尽管在bulk中确实含有一个名为discount\_policy的成员，，但是该成员对于itemp却是不可见的，itemp的类型是Quote的指针，以为这对discount\_policy的搜索将从Quote开始，所以我们无法通过Quote的对象、引用或者指针调用discount\_policy。

21.名字查找先于类型检查。

如之前p210页一样，内层作用域的函数并不会重载声明在外层作用域的函数，因此，定义派生类中的函数也不会重载其基类中的函数，和其他作用域一样，如果派生类（内层作用域）的成员与基类（外层作用域）的某个成员同名，则派生类将在其作用域内隐藏该基类成员，即使派生类成员和基类成员的形参列表不一致，基类成员也会被隐藏掉。

struct Base{

int memfcn();

};

struct Derived:public Base{

int memfcn(int);

};

Derived d;

Base b;

b.memfcn();//正确，调用Base::memfcn

d.memfcn(10);//正确，调用Derived::memfcn

d.memfcn();//错误，Base::memfcn被覆盖

d.Base::fcn();//正确，调用Base::memfcn

p549名字查找与继承。

22.友元和继承，每个类控制自己成员的访问权限。

第十六章：模板与泛型编程。

1.函数模板：

我们可以定义一个函数模板，而不是为每个类型都定义一个新函数，一个函数模板就是一个公式，一个函数模板就是一个公式，可用来生成针对特定类型的函数版本

template<typename T>

int compare(const T& v1,const T&v2){

if(v1<v2)return -1;

if(v2<v1)return 1;;

return 0;

}

模板定义以关键字template开始，后跟一个模板参数列表，这是一个逗号分隔的一个或多个模板参数的列表，用小于号(<)和大于号（>）包围起来。

注意：模板参数列表不能为空。

2.实例化函数模板

当我们调用一个函数模板时，编译器（通常）用函数实参来为我们推断模板实参，当我们调用compare时，编译器使用实参的类型类确定绑定到模板参数T的类型，例如：

cout<<compare(1,0)<<endl;//T为int

编译器用推断出的模板参数来为我们实例化。当编译器实例化一个模板时，它使用实际的模板实参代替对应的模板参数来创建出模板的一个新实例，例如：

cout<<compare(1,0)<<endl;

//实例化出int compare（const int& v1,const int&v2）

3.非类型模板参数

除了定义类型参数，还可以在模板中定义非类型参数，一个非类型参数表示一个值而非一个类型。我们通过一个特定的类型名而非关键字class或者typename来指定非类型参数。

当一个模板被实例化时，非类型参数被一个用户提供的或编译器推断出的值所代替，这些值必须是常量表达式，从而允许编译器在编译时实例化模板。

一个非类型参数可以是一个整形，或者是一个指向对象或函数类型的指针或（左值）引用，绑定到非类型整形参数的实参必须是一个常量表达式。p58，绑定到真真或引用非类型参数的实参必须据欧静态的生存期，我们不能用一个普通（非staitc）局部变量或动态对象作为指针或引用非类型参数的实参，指针参数也可以用nullptr或一个值为0的常量表达式来实例化。

编写类型无关的代码：：

我们compare函数虽然简单，但它说明了编写泛型代码的两个重要原则：

（1）模板中的函数参数是const的引用。

（2）函数体中的条件判断仅适用<运算比较。

注意：函数模板和类模板成员函数的定义通常放在头文件中。

4.类模板

类模板用来生成类。与函数模板不同的是，编译器不能为类模板推断模板参数类型，如我们已经多次看到的，为了使用类模板，我们必须在模板名后的尖括号中提供额外信息，用来代替模板参数的模板实参列表。

类模板的成员函数：

我们可以在类模板内部，也可以在类模板外部为其定义成员函数，且定义在类模板内的成员函数被隐式声明为内联函数。

（1） 类模板的成员函数本身是一个普通函数，但是，类模板的每个实例都有其自己版本的成员函数，因此，类模板的成员函数具有和模板相同的模板参数，因而，定义在类模板之外的成员函数就必须以关键字template开始，后解类模板列表。

template<typename T>

返回类型 Blob<T>::member\_name(parm\_list);

（2）默认情况下， 一个类模板的成员函数只有当程序用到它时才进行实例化。p587。

在类代码内中，即类模板自己的作用域中，我们可以直接使用模板名而不提供实参。

在类模板外定义成员时，必须记住，我们并不在类的作用域中，知道遇到类名才表示进入类的作用域。

template<typename T>

Blobptr<T> Blobptr<T>::operator++(int){

Blobptr ret=\*this;//这时在函数体内，已经进入类的作用域，所以在这定义ret时不用重复模板实参。

++\*this;

return ret;

}

5.类模板和友元

当一个类包含一个友元声明时，类与友元各自是否是模板是相互无关的，如果一个类模板包含一个非模板友元，则友元被授予可以访问所有模板实例，如果友元自身是模板，类可以授权给所有友元模板实例，也可以只授权给特定实例。

（1）一对一友好关系。p589：他们使用同一个typename。将一个模板的特定实例声明为友元要将友元前置声明。

（2）通用和特定的模板友好关系-一个类可以将另一个模板的每个实例都声明为自己的友元。或者限定特定的实例为友元。

6.在模板内不能重用模板参数名。

7.p593，使用类的类型成员

c++假定通过作用域运算符访问的名字不是类型。因此，如果我们希望使用一个模板类型参数的类型成员，就必须显示告诉编译器该名字是一个类型。我们通过使用关键字typename来实现这一点。

template<typename T>

template T::value\_type top(const T& c){

}

我们的top函数期待一个容器类型的实参，它使用typename知名其返回类型并在c中没有元素时生成一个值初始化的元素返回给调用者。

当通知编译器一个名字表示类型时，必须使用关键字typename而不能使用class。

8.成员模板

一个类（无论是普通类还是模板类）可以包含本身是模板的成员函数，这种成员被称为成员模板，成员模板不能是虚函数。

9.p597控制实例化

extern template declaration;//实例化声明

例子：external template class Blob<string>;

template declaration;//实例化定义

例子: template int compare(const int&,const int&);

当编译器遇到一个extern模板声明时，它不会再本文件中生成实例化代码，将一个实例化声明为extern就表示承诺在程序其他位置有该实例化的一个非extern声明（定义），对于一个给定的实例化版本，可能有多个extern声明，但必须只有一个定义。

特别注意：和普通类模板的普通实例化不同，实例化定义一个类模板时会实例化该类的所有成员，即使我们不使用某个成员。

10.shared\_ptr和unique\_ptr可以很容易的重载删除器的方式，我们可以很容易的重载一个shared\_ptr的删除器，只要在创建或reset指针时传递给它一个可调用对象即可。与之相反，删除器的类型是一个unique\_ptr对象的类型的一部分，用户必须在定义unique\_ptr时以显示模板实参的形式提供删除器的类型。对于unique\_ptr来说，提供自己的删除器更为复杂。

shared\_ptr在运行时绑定删除器:

我们可以推断出shared\_ptr必须能直接访问其删除器，即删除器必须保存为一个指针或一个封装了指针的类。(如function）。我们可以确定shared\_ptr不是将删除器直接保存为一个成员，因为删除器的类型知道运行时才会知道。实际上在一个shared\_ptr的生存期中，我们可以随时飞边其删除器的类型，我们可以使用一种类型的删除器构造一个shared\_ptr，虽有使用reset赋予此shared\_ptr另一种类型的删除器。而与之相反，类成员的类型在运行的时通常不能改变，（所以，不能用类成员直接保存删除器。）