1. 对象object和变量variable一般可以相互使用，c++中每个变量都有其数据类型，数据类型决定着变量所占内存空间的大小和布局方式、该空间能储存的值的范围，以及变量能参与的运算。
2. tip试试这样想，所谓指向常量的指针或引用（const & or const \*），不过是指针或引用“自以为是”罢了，它们觉得自己指向了常量（也可能变量），所以不自觉的不去改变所指向的值。
3. const指针，常量指针，int \* const。
4. 顶层const(top-level const)表示指针本身是个常量，底层const(low-level const)表示指针所指对象是一个常量。

Int i=0;

Int \*const p1=&I;//不能改变p1的值，这是一个顶层const

Const int ci=42;//不能改变ci的值，这是一个顶层const

Const int \*p2=&ci;//可以改变p2的值，这是一个底层const

1. Constexpr 指针的初始值必须是nullptr或者0，或者是储存于某个固定地址的对象。
2. 类型别名，typedef double wages; 别名申明，using SI=Sales\_item;
3. Const是对指定类型的修饰，typedef char \*pstring; const pstring cstr=0; const pstring \*ps; pstring 实际上是指向char的指针类型，因此，const pstring 就是指向char的常量指针，而非指向常量字符的指针。

如果把类型别名替换成它本来的样子，以理解语句含义：const char\* cstr=0;//是对const pstring cstr的错误理解//p61(87)。

1. Decltype类型指示符，有时希望从表达式的类型推断出要定义的变量类型，但是不想用该表达式的值初始化变量，c++11新标准引入第二种类型说明符decltype，它的作用是选择并返回操作数的数据类型，decltype((f())) sum=x;//sum的类型就是函数f的返回类型（编译器并不实际调用f）。

切记：decltype((variable))（注意双层括号）的结果永远是引用，而decltype(variable)结果只有当variable本身就是一个引用时才是引用。

1. 直接初始化和拷贝初始化，如果使用=初始化一个变量，实际上执行的是拷贝初始化，编译器会把等号右侧的初始值拷贝到新创建的对象中去，如果不使用等号，执行的是直接初始化。
2. WARNING:由于历史原因，也是为了兼容C，所以C++中的字符串字面值并不是标准库string的对象，切记，字符串字面值与string是不同的类型。
3. 某些编译器可能仍需以老式的声明语句来处理元素为vector的vector对象，如vector<vector<int> >，注意最后一个空格。
4. WARNING:vector对象（以及string对象）的下标运算符可用于访问已存在的元素，而不能用于添加元素。
5. C++语言定义了箭头运算符->，箭头运算符将解引用和成员访问两个操作结合在一起，也就是说，it->mem和(\*it).mem表达的意思相同。
6. WARNING:但凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素。
7. WARNING:不允许拷贝和赋值，不能将数组的内容拷贝给其他数组作为其初始值，也不能用数组为其他数组赋值。一些编译器支持数组的赋值，这就是所谓的编译器扩展(compiler extension)，但一般来说最好避免，因为含有非标准特性的程序很可能在其他编译器上无法正常哦工作。
8. 复杂数组声明，int arr[10]; int \*ptrs[10]; //ptrs是含有10个整型指针的数组，int (\*parray)[10]=&arr；//parray指向一个含有10个整数的数组，int (&arr\_ref)[10]=arr;//arr\_ref引用一个含有10个整数的数组。Tips：想要理解数组声明的含义，最好的办法就是从数组的名字开始按照由内向外的顺序阅读，例：int \*(&array)[10]=ptrs;//array是数组的引用，该数组含有10个指针。。
9. Note:string nums[]={“one”,”tow”,”three”}; string \*p=&nums[0]; //p指向nums的第一个元素，数组还有一个特性，在很多用到数组名字的地方，编译器都会自动将其替换为一个指向数组首元素的指针，string \*p2=nums;//等价于p2=&nums[10]。在大多数表达式中，使用数组类型的对象其实是一个指向该数组首元素的指针。
10. Int ia[]={0,1,2,3}//ia是一个含有34个整数的数组，auto ia2(ia)//等价于auto ia2(&ia[0])，ia2是一个整型指针，指向ia的第一个元素，单使用decltype不会发生，decltype(ia) ia3={0,1,2,3}//ia3是一个含有4个整数的数组。
11. 指针也是迭代器，int \*e=&arr[4]//指向array尾元素下一个位置的指针，重写之前的循环，for(int \*b=arr;b!=e;++b){cout<<\*b<<endl;//输出arr元素}
12. 混用string对象和C风格字符串，如果程序某处需要一个C风格字符串，无法直接使用string对象来代替它。例如，不能用string 对象直接初始化指向字符的指针，为了完成该功能，string专门提供了一个名为c\_str的成员函数，char \*str=s;//错误：不能用string对象初始化char\*，const char \*str=s.c\_str()；//正确，我们无法保证c\_str函数返回数组一直有效，如果后续操作改变了s的值可能让之前返回的数组失去效用，如果执行完c\_str()函数后程序想一直都用其返回的数组，最好将该数组重新拷贝一份。
13. 使用数组初始化vector对象，int int\_arr[]={0,1,2,3}，vector<int>ivec(begin(int\_arr),end(int\_arr))。
14. For(const auto &row:ia){for(auto col:row)};//正确

for(auto row:ia{for(auto col:row)})//无法通过编译，因为row不是引用类型，所以编译器初始化row时会自动将数组形式的元素转换成指向该数组内首元素的指针，这样得到的row类型就是int\*，显然内层的循环就不合法了，编译器试图在一个int\*内遍历，这显然和程序的初衷相去甚远。

Note:要使用范围for语句处理多维数组，除了最内层的循环外，其他所有循环的控制变量都应该是引用类型。

1. 运算对象operand，对表达式求值将得到一个结果result，字面值和变量是最简单的表达式expression，其结果就是字面值和变量的值，把一个运算符operator和一个或多个运算对象组合起来可以生成较复杂的表达式。
2. 左值和右值，简单归纳，当一个对象被用作右值的时候，用的是对象的值（内容），当对象被用作左值的时候，用的是对象的身份（在内存中的位置）；当一个左值被当成右值使用的时候，实际使用的是它的内容（值）。

* 赋值运算符需要一个（非常量）左值作为其左侧运算对象，得到的结果仍然是一个左值
* 取地址符作用于一个左值运算对象，返回一个指向该运算对象的指针，这个指针是一个右值
* 内置解引用运算符、下标运算符、迭代器解引用运算符、string和vector下标运算符的求值结果都是左值
* 内置类型和迭代器的递增递减运算符作用于左值运算对象，其前置版本（之前章节所用的形式）所得到的结果是右值

使用decltype时候，左值和右值有所不同，如果表达式求值结果是左值，decltype作用于该表达式（非变量）得到一个引用类型。例如：假定p的类型是int\*，因为解引用运算符生成左值，所以decltype(\*p)的结果是int&，另外，因为取地址运算符生成右值，所以decltype(&p)结果是int\*\*，结果是一个指向整型指针的指针。

1. 除了-m 导致溢出的特殊情况，其他时候(-m)/n和m/(-n)都等于-(m/n)，m%(-n)等于m%n，(-m)%n等于-(m%n)。
2. ++i，前置版本，对象本身作为左值返回，i++，后置版本将对象原始值的副本作为右值返回，后置版本将原始值储存下来以便返回这个未修改的内容（建议使用前置版本）。
3. Sizeof(type),sizeof expr，sizeof并不实际计算其运算对象的值。

对于数组执行sizeof得到整个数组所占空间的大小，等价于对数组中所有元素各执行一次sizeof并将结果求和，注意sizeof不会将数组转换成指针处理，对string对象或vector对象，sizeof只返回该类型固定部分大小，不会计算对象中元素占用了多少空间。

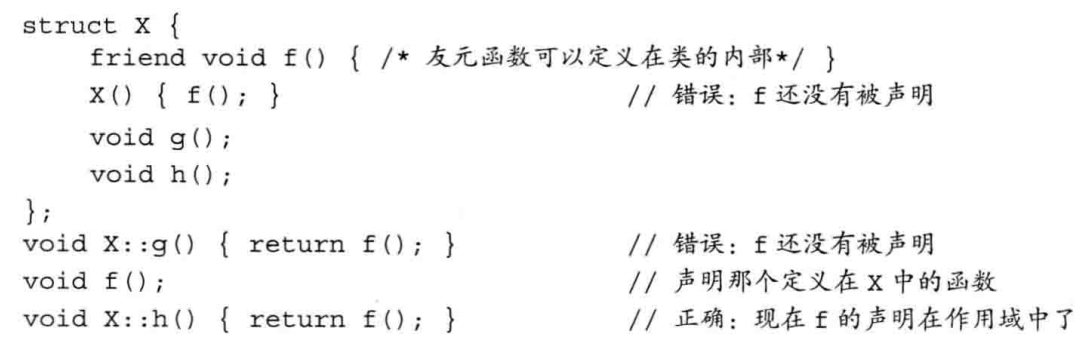
1. 命名的强制类型转换，cast-name<type>(expression);其中，type是转换的目标类型而expression是要转换的值，如果type是引用类型，则结果是左值，cast-name是static\_caset，dynamic\_cast，const\_cast，reinterpret\_cast中一种，dynamic\_cast支持运行时类型识别。

Static\_cast，只要不包含底层const都可以使用，double slope=static\_cast<double>(j)/I;把一个较大的算术类型赋值给较小的类型时，static\_cast非常有用，这告诉编译器我们知道且不在乎潜在精度损失，void \*p=&d;//任何非常量对象地址都能存入void\*，double \*dp=static\_cast<double\*>(p)；//将void\*转换回初始的指针

Const\_cast，const\_cast只能改变运算对象底层const，const char \*pc; char \*p=const\_cast<char\*>(pc);//正确，但是通过p写值是未定义的行为

Reinterpret\_cast危险

1. C++局部对象，名字有作用域，对象有生命周期，这两个概念非常重要。自动对象，把只存在于块执行期间的对象称为自动对象（automatic object），当块的执行结束后，块中创建的自动对象的值就变成未定义的了，形参是一种自动对象。
2. Const形参和实参，顶层const作用于对象本身，const int ci=42;//不能改变ci，const是顶层的，int i=ci;//正确，当拷贝ci时，忽略了它的顶层const，int\* const p=&i;//const是顶层的，不能给p赋值，\*P=0;//正确，通过p改变对象的内容是允许的，现在i变成了0
3. 因为数组不能被拷贝，所以函数不能返回数组，但是可以返回数组的指针或引用。
4. 返回数组指针的函数形式如下：Type(\*function(parameter\_list))[dimention]
5. 尾置返回类型，trailing return type，auto func(int i)->int(\*)[10];
6. 函数重载和const形参，Record lookup(Account&);//作用于Account的引用，Record lookup(const Account&);//新函数，作用于常量引用，Record lookup(Account\*);//新函数，作用于指向Account的指针，Record lookup(const Account\*);//新函数，作用于指向常量的指针
7. C++中，名字查找发生在类型检查之前。
8. 通常，应该在函数声明中指定默认实参，并将该声明放在合适的头文件中,string screen(std::string::size\_type ,std::string::size\_type,char=’’); string screen(std::string::size\_type ,std::string::size\_type,char=’\*’);//错误：重复声明，string screen(std::string::size\_type=24,std::string::size\_type=80,char);//正确，添加默认实参。
9. Constexpr函数是指能用于常量表达式的函数。其中，函数的返回类型及所有形参类型都得是字面值类型，且函数体中必须只有一条return语句，constexpr int new\_sz(){return 42;} constexpr int foo=new\_sz()。Constexpr函数被隐式指定为内联函数。Constexpr函数不一定返回常量表达式。
10. 返回指向函数的指针，想要声明一个返回函数指针的函数，最简答的办法是使用类型别名：using F=int(int\*,int);//F是函数类型，不是指针，using PF=int(\*)(int\*,int);//PF是指针类型，另一种声明方式，int(\*f1(int))(int\*,int);//f1有形参列表，所以f1是个函数，f1前面有个\*，所以f1返回一个指针，进一步指针类型本身也包含形参列表，因此指针指向函数，该函数返回类型是int。也可，auto f1(int)->int(\*)(int\*,int)
11. 类的基本思想：数据抽象（data abstration）和封装（encapsulation）。数据抽象是一种依赖于接口（interface）和实现（implementation）分离的办成技术。类的接口包括用户所能执行的操作；类的实现包括类的数据成员、负责接口实现的函数体以及定义类所需的各种私有函数。封装实现了类的接口和实现的分离。封装后的类隐藏了它的实现细节，也就是说，类的用户只能使用接口而无法访问实现部分。
12. 成员函数通过一个名为this的额外的隐式参数来访问调用它的那个对象，当调用一个成员函数时，用请求该函数的对象地址初始化this。因为this的目的总是指向“这个”对象，所以this是一个常量指针，我们不允许改变this中保存的地址。
13. 引入const成员函数，string isbn()const{return this->bookNo;}，const的作用是修改隐式this指针的类型。默认情况下，this的类型是指向类类型非常量版本的常量指针，例如Sales\_data成员函数中，this的类型是Sales\_data\*const。尽管this是隐式的，但仍遵循初始化规则，我们不能把this绑定到一个常量对象上。把this声明成const Sales\_data\*const，毕竟，在isbn函数体内不会改变this所指的对象。C++把const放在成员函数的参数列表后，此时，紧跟在参数列表后的const表示this是一个指向常量的指针，像这样使用const的成员函数称为常量成员函数(const member function)。Note:常量对象，以及常量对象的引用或指针都只能调用常量成员函数。
14. 类的作用域和成员函数，类本身就是一个作用域。编译器分两步处理类：首先编译成员的声明，然后才轮到成员函数体（如果有的话），因此，成员函数体可以随意使用类的其他成员而无需在意成员出现的次序。
15. IO类属于不能被拷贝的类型，因此我们只能通过引用来传递它们，且，因为读取和写入操作会改变流的内容，所以两个函数接受的都是普通引用，而非对常量的引用。
16. 不同于其他成员函数，构造函数（constructor）不能被声明成const的，当创建类的一个const对象时，直到构造函数完成初始化过程，对象才能真正取得其“常量”属性。因此，构造函数在const对象的构造过程中可以向其写值。
17. 编译器创建的构造函数又称为合成的默认构造函数(synthesized default constructor)，默认构造函数在很多方面都有其特殊性，其一为如果我们的类没有显示定义构造函数，那么编译器就会隐式地定义一个默认构造函数。Note:只有当类没有声明任何构造函数时，编译器才会自动生成默认构造函数。
18. 使用class和struct定义类的唯一区别就是默认访问权限，class为private，struct为public。
19. 友元：既然Sales\_data数据成员是private的，我们的read\print\add函数就无法正常编译了，因为尽管这些函数是类接口一部分，但它们不是类成员。
20. 类可以允许其他类或者函数访问它的非公有成员，方法是令其他类或者函数称为它的友元（friend）。如果类想把一个函数作为它的友元，只需要增加一条以friend关键字开始的函数声明语句即可。Note:最好在类定义开始或结束前的位置集中声明友元。
21. 右值引用rvalue reference：所谓右值引用就是必须绑定到右值的引用。用&&而不是&来获取右值引用，右值引用重要性质，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。一般而言，一个左值表达式表示的是一个对象的身份，而一个右值表达式表示的是一个对象的值。Int i=42; int &r=I;//正确：r左值引用i，int &&rr2=i\*42;//正确：将rr2绑定到乘法结果上。
22. 左值持久，右值短暂。右值要么是字面常量，要么是在表达式求值过程中创建的临时对象；由于右值引用只能绑定到临时变量中，我们得知，所引用的对象将要被销毁，该对象没有其他用户，这两个特性意味着：使用右值引用的代码可以自由接管所引用对象的资源。Note：右值引用指向将要被销毁的对象，因此可以从绑定到右值引用的对象“窃取”状态。
23. 变量是左值。变量可以看作只有一个运算对象而没有运算符的表达式。Int&&rr1=42;//正确：字面常量是右值，int&&rr2=rr1//错误：表达式rr1是左值！move函数，显示的将一个左值转换为对应的右值引用类型，move函数可以获得绑定到左值上的右值引用，int&&rr3=std::move(rr1);//ok,note:我们可以销毁一个移后原对象，也可以赋予它新值，但不能使用一个移后源对象的值。
24. 可变数据成员（mutable data member）永远不会是const，即便它是const对象的成员，因此，一个const成员函数可以改变一个可变成员的值，变量声明引入mutable关键字。
25. Note:当提供一个类内初始值时，必须以符号=或者花括号表示。
26. Note:一个const成员函数如果以引用的形式返回\*this，那么它的返回类型将是常量引用。
27. 因为非常量版本的函数对于常量对象是不可用的，所以我们只能在一个常量对象上调用const成员函数，另一方面，虽然可以在非常量对象上调用常量版本和非常量版本，但显然此时非常量版本是一个更好的匹配。
28. 类类型。对于一个类来说，在我们创建它的对象之前该类必须被定义过，而不能仅仅被声明，否则，编译器就不知道这样的对象需要多少储存空间。然而，一个类的名字出现后，它就被认为是声明过了（但尚未定义），因此类允许包含指向它自身类型的指针或引用。Class link{link\* next; link\* prev;}
29. 友元再探。友元函数能定义在类的内部，这样的函数是隐式内联的。如果一个类指定了友元类，则友元类的成员函数可以访问此类包括非公有成员在内的所有成员。
30. 友元关系不存在传递性，每个类负责控制自己的友元类或友元函数。
31. 想要某个成员函数作为友元，必须按照如下方式：1.首先定义Window\_mgr类，其中声明clear函数，但不定义。在clear使用Screen成员前必须先声明Screen.2.接下来定义Screen，包括对clearn友元声明。3.最后定义clear，此时才可以使用Screen成员。
32. 类和非成员函数的声明不是必须在它们的友元声明之前。当一个名字第一次出现在一个友元声明中时，我们隐式假定该名字在当前作用域中是可见的。然而，友元本身不一定真的声明在当前作用域中。甚至就算在类的内部定义该函数，我们也必须在类的外部提供相应声明从而使函数可见，换句话说，即使我们仅仅是用声明友元的类的成员调用该友元函数，它也必须是被声明过的：



关于这段代码最重要的是理解友元声明的作用是影响访问权限，它本身并非普通意义上的声明。Note:有的编译器并不强制执行上述关于友元的限定规则。

1. Note:编译器处理完类中的全部声明后才会处理成员函数的定义。
2. Note:如果成员是const、引用，或者属于某种未提供默认初始构造函数的类类型，我们必须通过构造函数初始值列表为这些成员提供初值。建议：使用构造函数初始值，在很多类中，初始化和赋值的区别事关底层效率问题：前者直接初始化数据成员，后者则先初始化再赋值。除了效率问题外更重要的是，一些数据成员必须被初始化。
3. 委托构造函数（delegating constructor），一个委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把它自己的一些（或者全部）职责委托给了其他构造函数。Warning:对于c++新手来说有一种常见的错误，它们试图以如下形式声明一个用默认构造函数初始化的对象：Sales\_data obj();//错误：声明了一个函数而非对象，Sales\_data obj2;//正确：obj2是一个对象而非函数。
4. 隐式的类类型转换：如果构造函数只接受一个是实参，则它实际上定义了转换为此类类型的隐式转换机制，有时我们把这种构造函数称作转换构造函数（converting constructor）。Note：能通过一个实参调用的构造函数定义了一条从构造函数的参数类型向类类型隐式转换的规则。

String null\_book=”9-999-99999-9”;

//构造一个临时的Sales\_data对象

//该对象的units\_sold和revenue等于0，bookNo等于null\_book

Item.combine(null\_book);

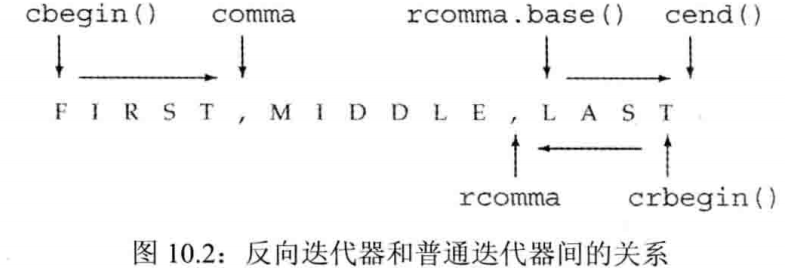
这里我们用一个string实参调用了Sales\_data的combine成员。该调用是合法的，编译器用给定的string自动创建一个Sales\_data对象。新生成的这个（临时）Sales\_data对象被传递给combine。因为combine的参数是一个常量引用，所以我们可以给该参数传递一个临时量。

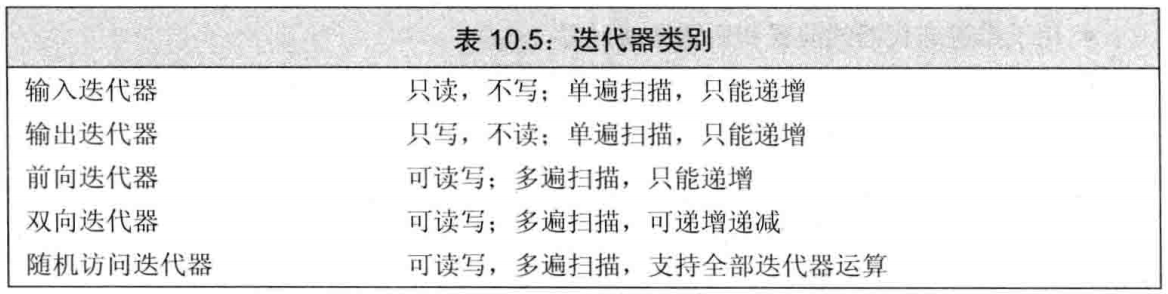
**只允许一步类类型转换：**//错误：需要用户定义两种转换（1）把“9-999-99999-9“转换成string（2）再把这个（临时的）string转换成Sales\_data

Item.combine(“9-999-99999-9”);//正确：显示地转换成string，隐式地转换成Sales\_data item.combine(string(“9-999-99999-9”));//正确：隐式转换成string，显示地转换成Sales\_data item.combine(Sales\_data(“9-999-99999-9”));

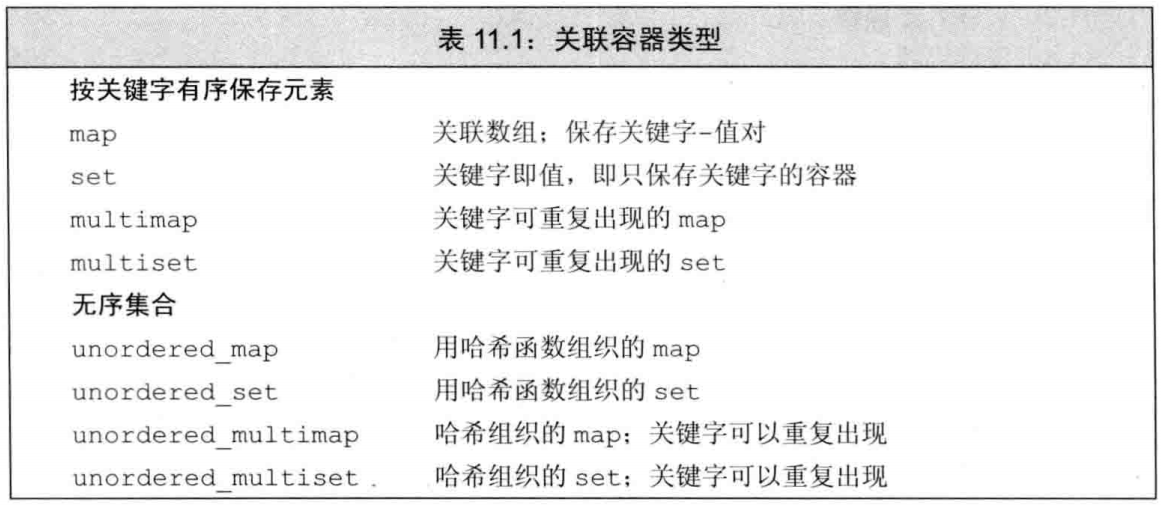
1. 聚合类（aggregate class）：满足条件（1）所有成员都是public的（2）没有定义任何构造函数（3）没有类内初始值（4）没有基类，也没有virtual函数；例如：struct data{int ival; string s;}
2. 类的静态成员：有时候类需要它的一些成员与类本身直接相关，而不是与类的各个对象保持联系。例如：一个银行账户类可能需要一个数据成员来表示当前的基准利率。在此例中，我们希望利率与类关联，而非与类的每个对象关联。从实现效率的角度看，没必要每个对象都储存利率信息。而更加重要的是，一旦利率浮动，我们希望所有的对象都能使用新值。Note:和类的所有成员一样，当我们指向类外部的静态成员时，必须指明成员所属的类名。Static关键字则只出现在类内部的声明语句中。

BestPractice:即便是一个常量静态数据成员在类内部被初始化了，通常情况下也应该在类的外部定义一下该成员。

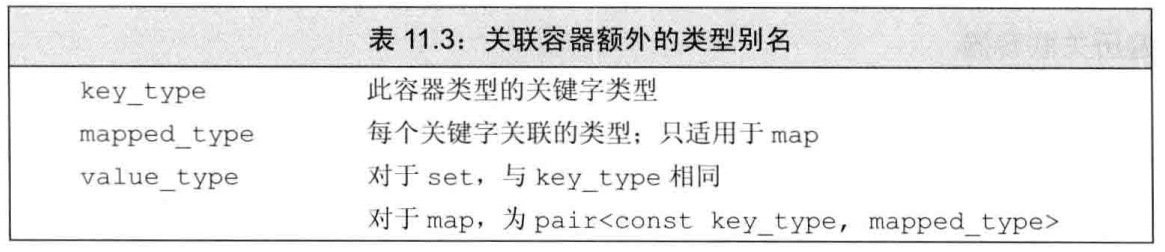
1. Note:当将一个容器初始化为另一个容器的拷贝时，两个容器的容器类型和元素类型都必须相同。但当传递迭代器参数来拷贝一个范围时，就不要求容器类型是相同的了。而且，新容器和原容器中的元素类型也可以不同，只要能将要拷贝的元素转换为要初始化的容器的元素类型即可。
2. Warning:赋值相关运算会导致指向左边容器内部的迭代器、引用和指针失效。而swap操作将容器内容交换不会导致指向容器的迭代器、引用和指针失效(容器类型为array和string的情况除外)。
3. Embrace操作：构造而不是拷贝元素。当调用push或insert成员函数时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器中。而当我们调用一个embrace成员函数时，则是将参数传递给元素类型的构造函数。Embrace成员使用这些参数在容器管理的内存中直接构造元素。//在c的末尾构造一个Sales\_data对象，c.embrace\_back(“987-123”,25,15.99)，//创建一个临时的Sales\_data对象传递给push\_back，c.push\_back(Sales\_data(“987-123”,25,15.99))。
4. 每个适配器都定义两个构造函数：默认构造函数创建一个空对象，接受一个容器的构造函数拷贝该容器来初始化适配器。Stack<int>stk(deq)；//从deq拷贝元素到stk。
5. WARNING:那些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。
6. 谓词（predicate）：是一个可调用的表达式，其返回结果是一个能用作条件的值。标准库算法所使用的谓词分为两种：一元谓词（只接受单一参数）和二元谓词（两个参数），接受谓词参数的算法对输入序列中的元素调用谓词。因此，元素类型必须能转换为谓词的参数类型。
7. Lambda：[capture list](parameter list)->return type{function boty}，其中，capture list是一个lambda所在函数中定义的局部变量的列表（通常为空），return type、parameter和function body与任何普通函数一样，但是lambda必须使用尾置返回来指定返回类型。可以忽略参数列表和返回类型，但必须永远包含捕获列表和函数体。Auto f=[]{return 42;} cout<<f()<<endl;//42
8. Note：捕获列表只用于局部非static变量，lambda可以直接使用局部static变量和在它所在函数之外声明的名字。
9. 可变lambda，默认情况，对于一个值被拷贝的变量，lambda不会改变其值。如果希望改变一个被捕获变量的值，就必须在参数列表首加上关键字mutable。
10. Bind，functional头文件中，可以看作一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来“适应”原对象的参数列表。Auto newcallable=bind(callable,arg\_list)。函数ref返回一个对象，包含给定的引用，此对象是可以拷贝的，标准库还有一个cref函数，生成一个保存const应用的类。
11. 插入迭代器（insert iterator），是一种迭代器适配器，它接受一个容器，生成一个迭代器，能实现向给定容器添加元素。当通过一个插入迭代器进行赋值时，该迭代器调用容器操作来向给定容器的指定位置插入一个元素。



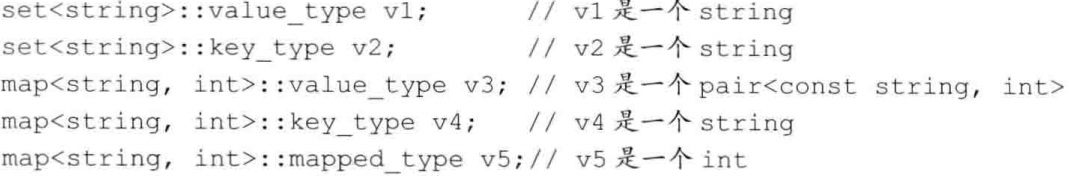
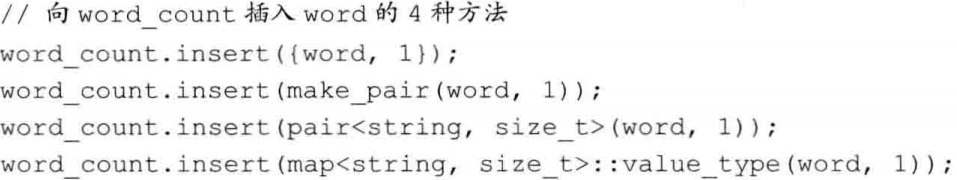
1. 关联容器：

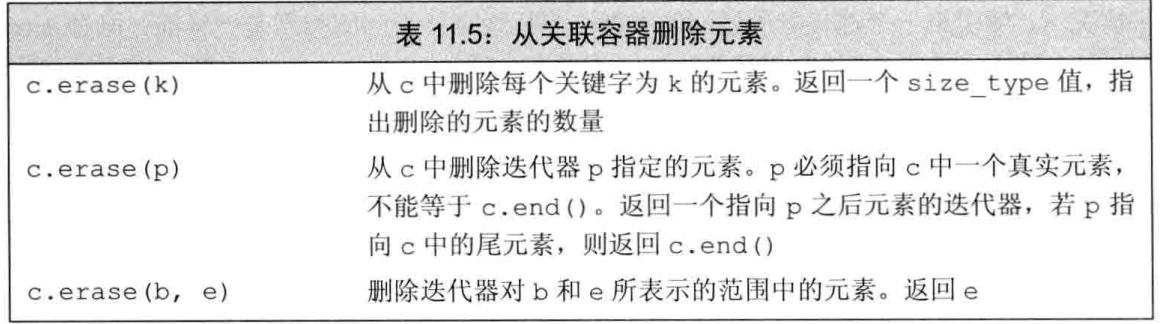


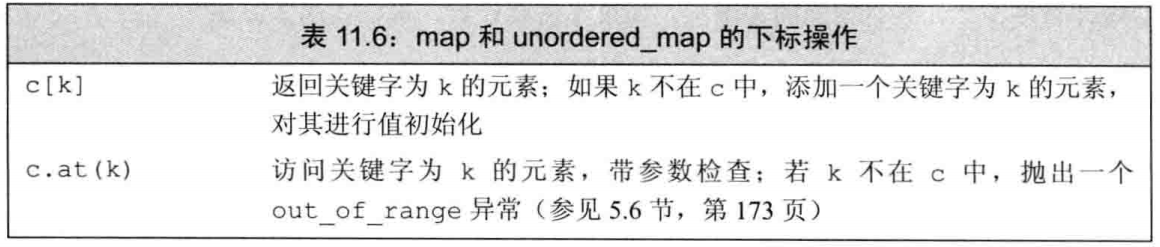
1. 关联容器操作：

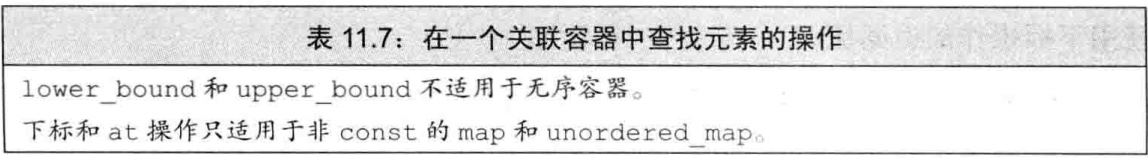


对于set类型，key\_type和value\_type是一样的，set中保存的值就是关键字。在一个map中，元素是关键字-值对，即每个元素是一个pair对象，包含一个关键字和一个关联的值，我们不能改变一个元素的关键字，因此pair的关键字部分是const的。

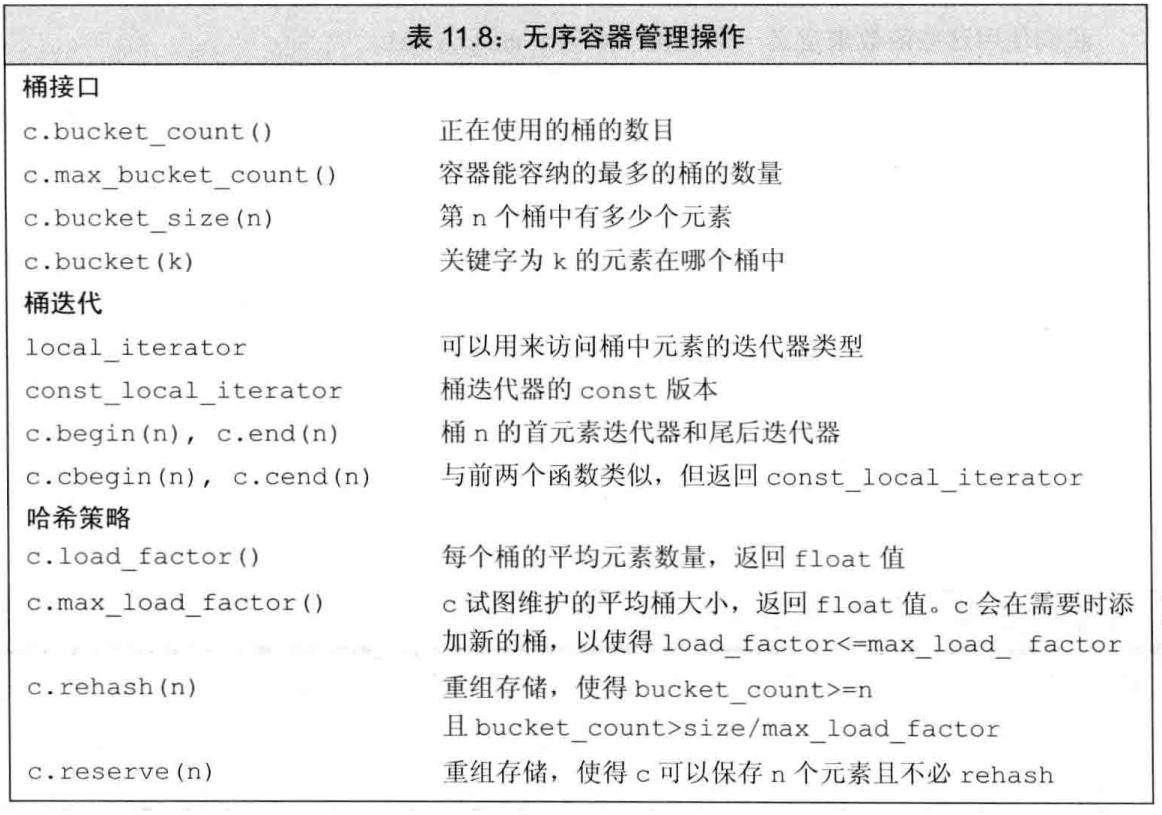




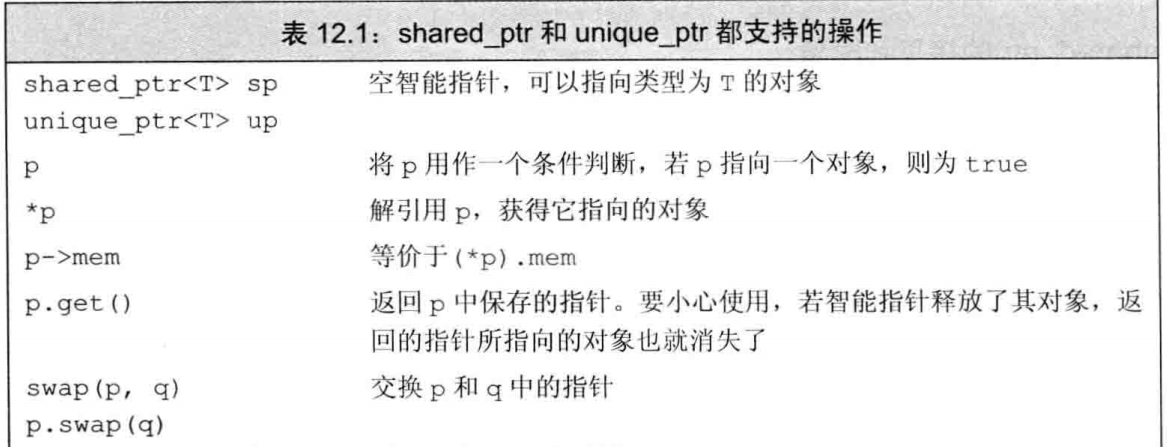


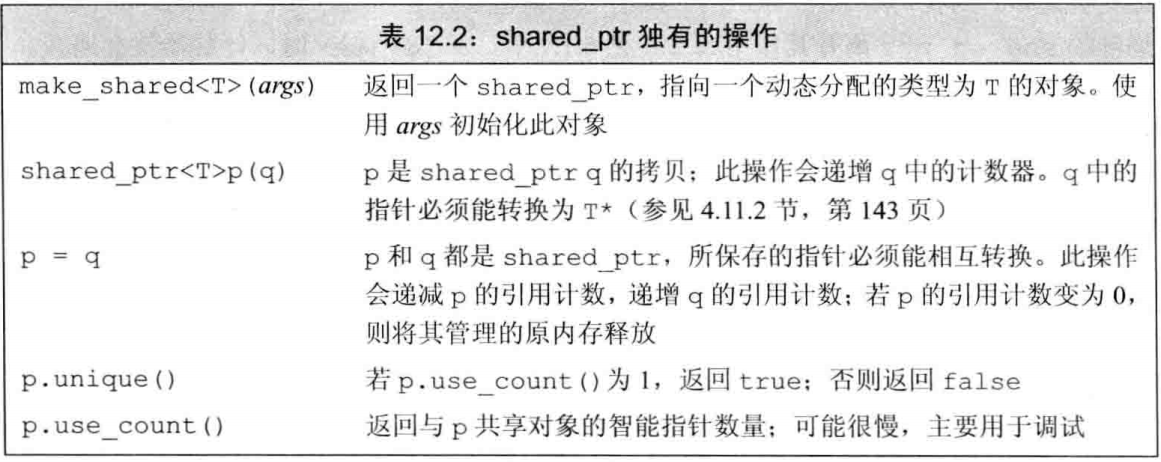




Equal\_range函数，接受一个关键字，返回一个迭代器pair，若关键字存在，则第一个迭代器指向第一个与关键字匹配的原色，第二个迭代器指向最后一个匹配元素之后的位置。若未找到匹配元素，则两个迭代器都指向关键字可以插入的位置。

1. 动态内存，除了自动和static对象外，c++还支持动态内存对象。动态内存的对象的生存期与它们在哪里创建是无关的，只有当显示地被释放时，这些对象才会销毁。我们程序目前为止只用到静态内存和栈内存。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。分配在静态或栈内存中的对象由编译器自动创建和销毁。
2. 为了更容易和安全的使用动态内存，标准库使用两种智能指针类型管理动态内存。智能指针行为类似常规指针，重要区别是它负责自动释放所指向的对象。两种智能指针的区别在于管理底层指针的方式：shared\_ptr允许多个指针指向同一个对象，unique\_ptr则“独占”所指向的对象。标准库还定义了一个名为weak\_ptr的伴随类，它是一种弱引用，指向shared\_ptr所管理的对象。三种类型都定义在memory头文件中。
3. 智能指针：



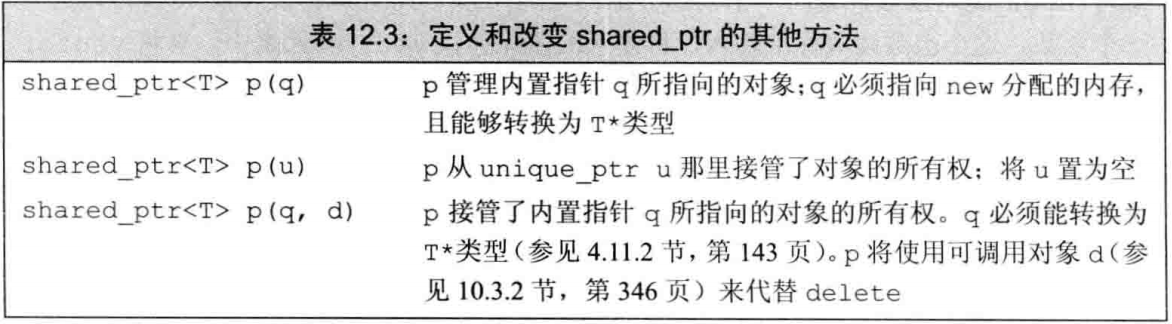


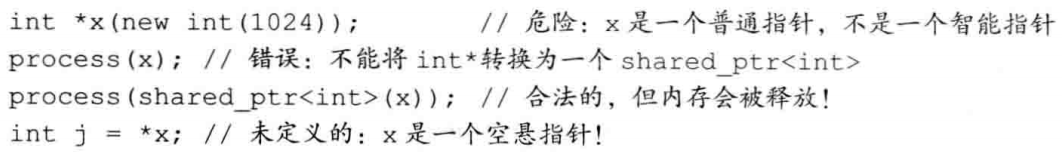
最安全的分配和使用动态内存的方法是调用一个名为make\_shared的标准库函数。此函数在动态内存中分配一个对象并初始化它，返回指向此对象的shared\_ptr。

1. Note:如果将shared\_ptr存在一个容器中，而后不再需要全部元素，而只使用其中一部分，要记得用erase删除不再需要的那些元素。

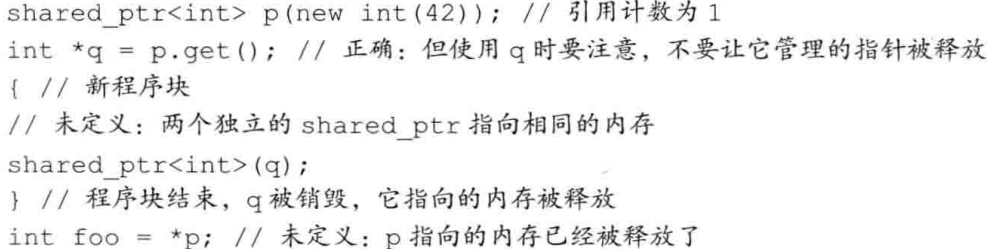
程序使用动态内存出于以下三种原因之一：1.程序不知道自己需要使用多少对象，2.程序不知道所需对象的准确类型，3.程序需要在多个对象间共享数据。Note:使用动态内存的一个常见原因是允许多个对象共享相同的状态。

1. 直接内存管理：在自由空间分配的内存是无名的，因此new无法为其分配的对象命名，而是返回一个指向该对象的指针。Int \*pi=new int;//pi指向一个动态分配的、未初始化的无名对象。
2. 定位new(placement new)，int \*p1=new int;//若分配失败，new抛出std::bad\_alloc，int \*p2=new (nothrow)int;//若分配失败，new返回一个空指针。与new类型类似，delete表达式也执行两个动作：销毁给定的指针指向的对象；释放对应的内存。
3. Delete之后重置指针值。当我们delete一个指针后，指针值就变为无效了。虽然指针已经无效，但在很多机器上指针任然保持着（已经释放了的）动态内存的地址。在delete之后，指针就变成了所谓的空悬指针（dangling pointer），即，指向一块曾经保存数据对象但现在已经无效的内存的指针。如果任需要保留指针，可以在delete之后将nullptr赋予nullptr，这样就清除地指出不指向任何对象。
4. 动态内存的一个基本问题是可能有多个指针指向相同的内存。在delete内存之后重置指针的方法只对这个指针有效，对其他任何任指向（已经释放的）内存的指针是没有作用的。
5. Shared\_ptr和new结合使用。若我们不初始化一个智能指针，它就会被初始化为一个空指针。我们还可以用new返回的指针来初始化智能指针：shared\_ptr<double>p1;//shared\_ptr可以指向一个double，shared\_ptr<int>p2(new int(42));//p2指向一个值为42的int。
6. 接受指针参数的智能指针构造函数是explicit的。因此，不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式来初始化一个智能指针。Shared\_ptr<int>p1=new int(1024);//错误：必须使用直接初始化方式，shared\_ptr<int>p2(new int(1024));//正确。出于相同的原因，一个返回shared\_ptr的函数不能在其返回语句中隐式转换一个普通指针：shared\_ptr<int>clone(int p){return new int(p);};//错误：隐式转换为shared\_ptr<int>。
7. 默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用delete释放它所关联的对象。我们可以将智能指针绑定到一个指向其他类型的资源的指针上，但是为了这么做，必须提供自己的操作来代替delete。

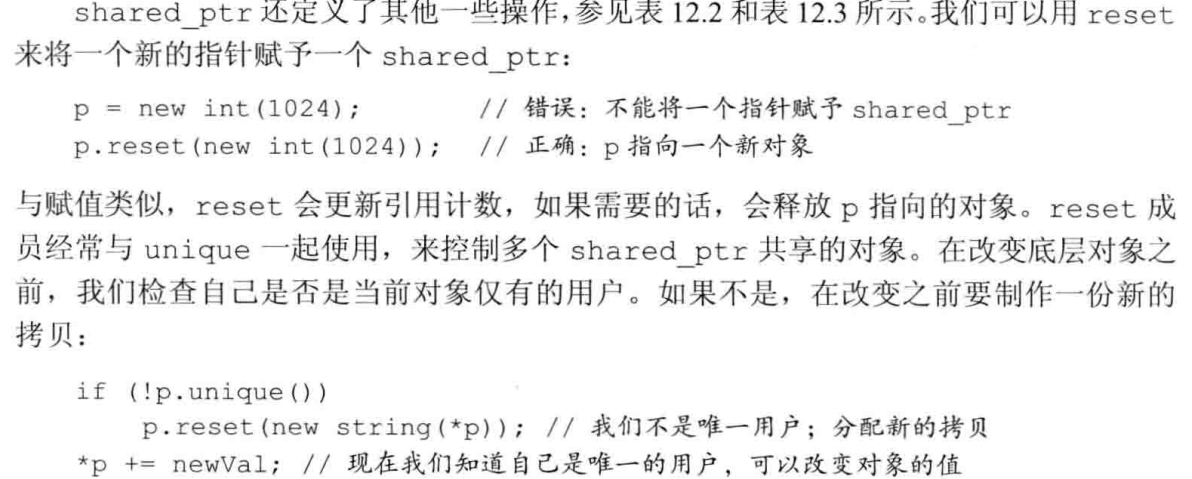




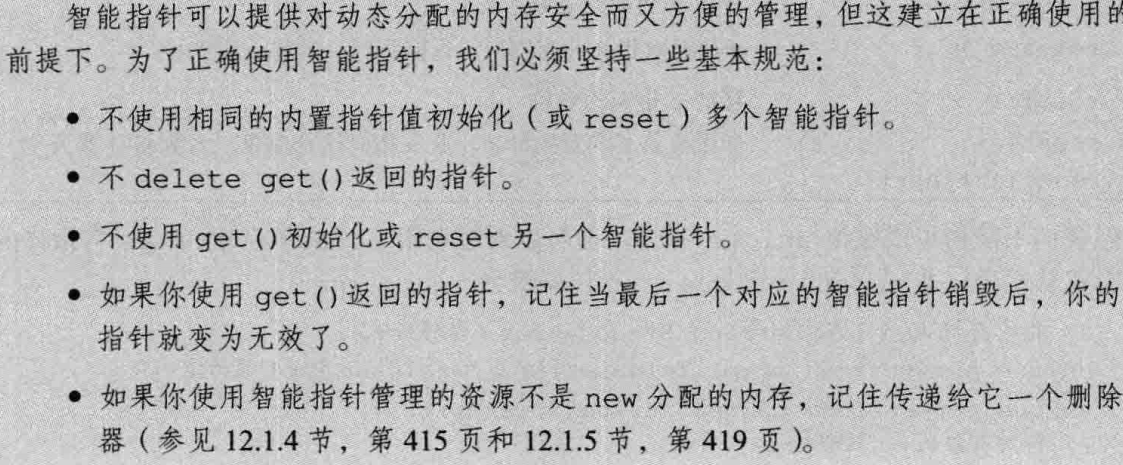
也不要使用get初始化另一个智能指针或为智能指针赋值。



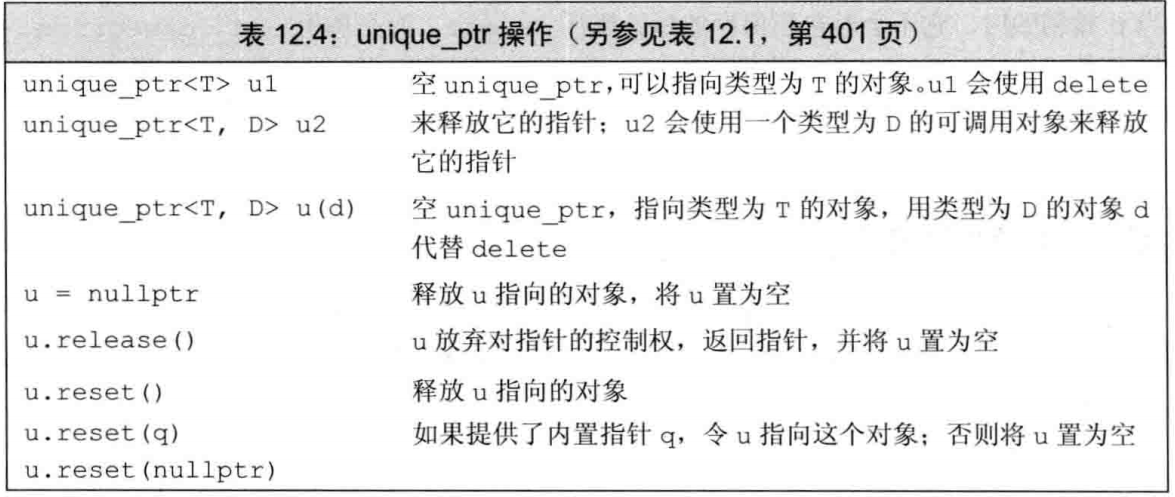
Warning:get用来将指针的访问权限传递给代码，你只有在确定代码不会delete指针的情况下，才能使用get。特别是，永远不要用get初始化另一个智能指针或者为另一个智能指针赋值。



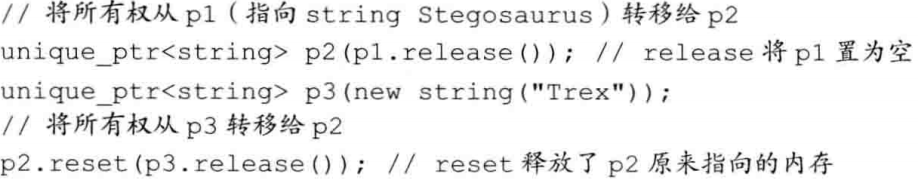
Warning：智能指针陷阱



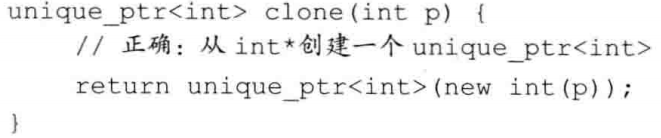
1. Unique\_ptr



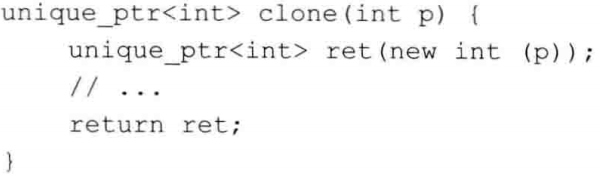
虽然我们不能拷贝或赋值unique\_ptr，但可以通过调用release或reset将指针的所有权从一个（非const）unique\_ptr转移到另一个unique：



传递unique\_ptr参数和返回unique\_ptr，不能拷贝unique\_ptr的规则有一个例外：我们可以拷贝或赋值一个将要被销毁的unique\_ptr。最常见的例子是从函数返回一个unique\_ptr：

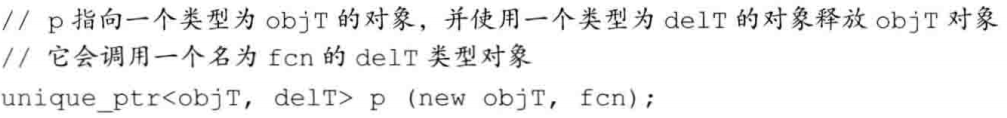


还可以返回一个局部对象的拷贝：

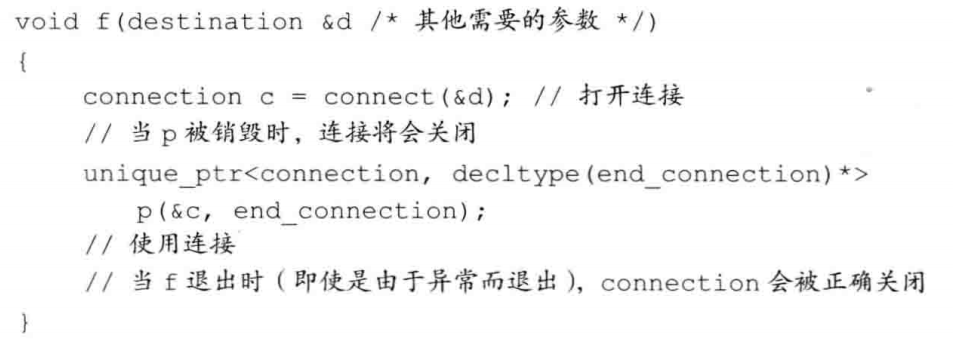


对于两段代码，编译器都知道要返回的对象将要被销毁。在次情况下，编译器执行一种特殊的“拷贝”，将在13.6.2节中介绍。

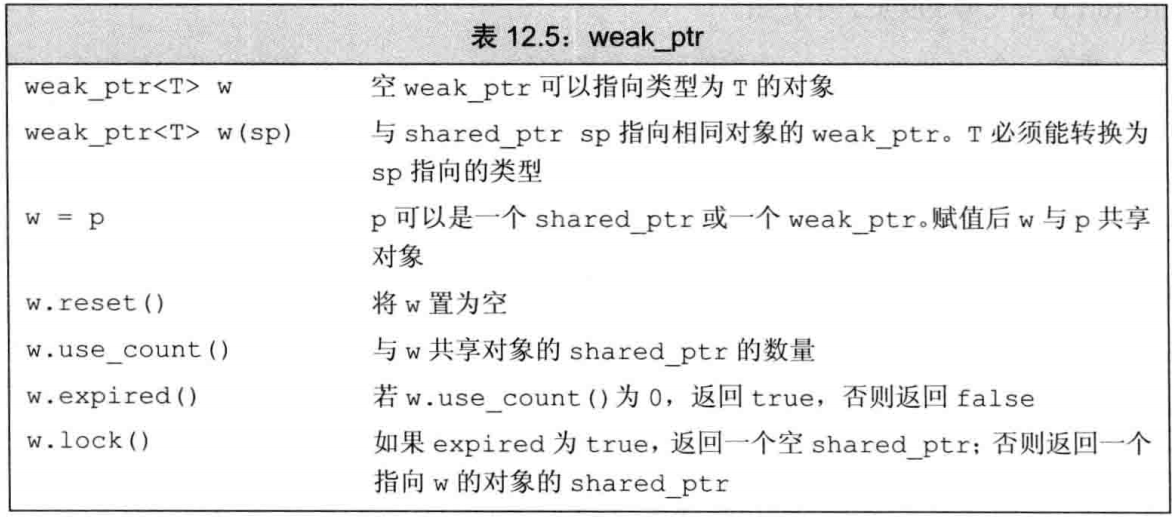
向unique\_ptr传递删除器：



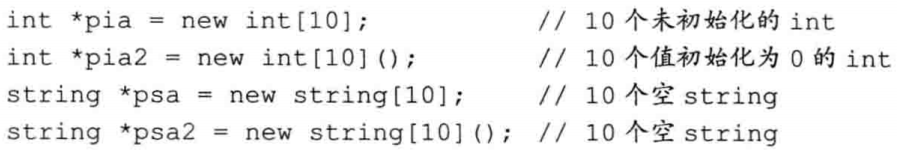
重写连接程序，用unique\_ptr来代替shared\_ptr：



1. Weak\_ptr：是一种不控制所指向对象生存期的智能指针，它指向一个shared\_ptr管理的对象。将一个weak\_ptr绑定到一个shared\_ptr不会改变shared\_ptr的引用计数。一旦最后一个指向对象的shared\_ptr被销毁，对象就会被释放。即使有weak\_ptr指向对象，对象也还是会被释放，因此，weak\_ptr的名字抓住了这种智能指针“弱”共享对象的特点。

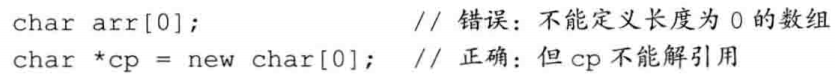


1. New和数组，int\*pia=new int[get\_size()];//pia指向第一个int，方括号中的大小必须是整型，但不必是常量。也可以，typedef int arr[42]; int\*p=new arr。分配一个数组会得到一个元素类型的指针，所谓的动态数组并不是数组类型，这是很重要的。
2. 初始化动态分配对象的数组，默认情况下，new分配的对象，不管是单个分配还是数组中的，都是默认初始化的，可以对数组中的元素进行值初始化，方法是在大小之后跟一对空括号。



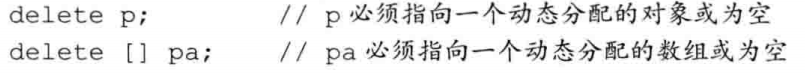


动态分配一个空数组是合法的：

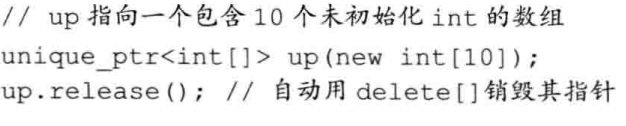


当我们用new分配一个大小为0的数组时，new返回一个合法的非空指针。对于零长度的数组来说，此指针就像尾后指针一样，我们可以像使用尾后迭代器一样使用这个指针。可以向此指针加上（减去）0，也可以从此指针减去自身从而得到0.但此指针不能解引用-毕竟它不指向任何元素。

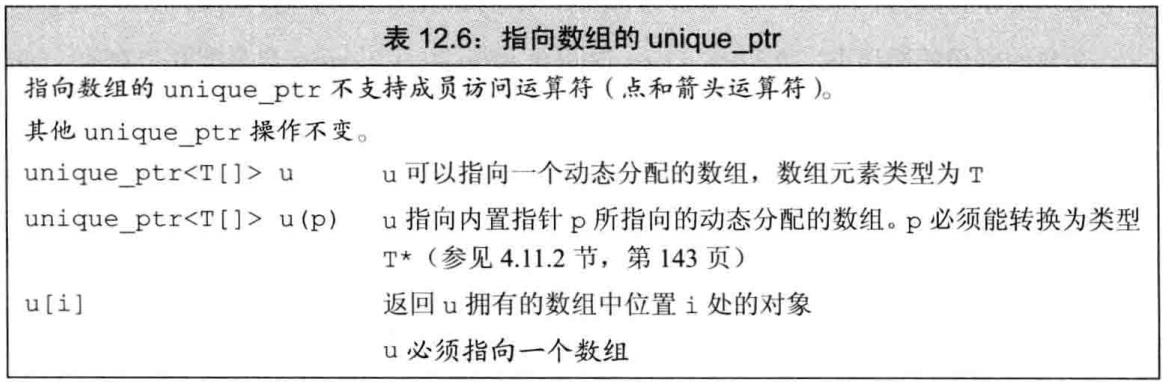
释放动态数组：



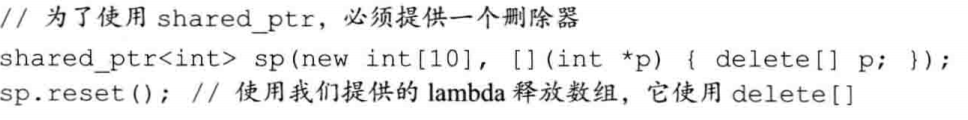
智能指针和动态数组：

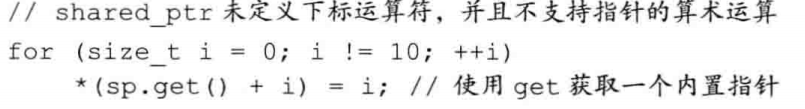


当一个unique\_ptr指向一个数组时，我们不能使用点和箭头成员运算符。毕竟unique\_ptr指向的是一个数组而不是单个对象，因此这些运算符是无意义的。



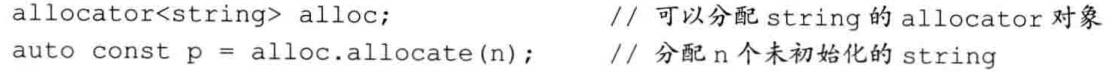
与unique\_ptr不同，shared\_ptr不直接支持管理动态数组。如果希望使用shared\_ptr管理一个动态数组，必须提供自己定义的删除器。

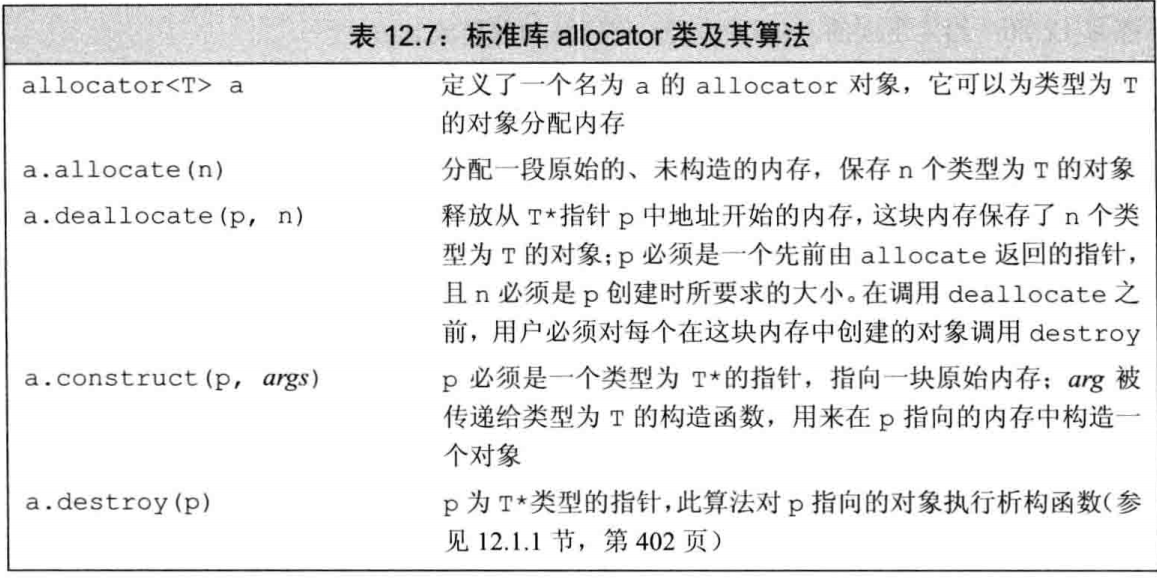




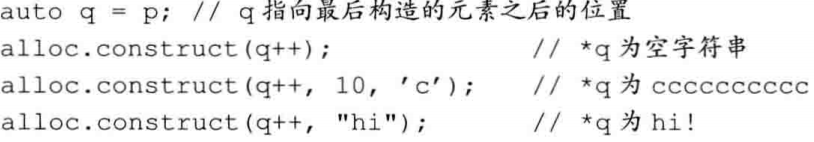
Shared\_ptr未定义下标运算符，而且智能指针类型不支持指针算术对象。因此，为了访问数组中的元素，必须用get获取一个内置指针，然后用它来访问数组元素。

1. Allocator类。New有一些灵活性上的局限性，其中一方面表现在它将内存分配和对象构造组合在一起了。类似，delete将对象析构和内存释放组合在一起了。分配单个对象时，通常希望将内存分配和对象初始化组合在一起。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。在此情况下，我们希望将内存分配和对象构造分离。意味着可以分配大块内存，但只在真正需要时才真正执行对象创建操作（同时付出一定开销）。
2. 标准库allocator定义在头文件memory中，它帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。Allocator是一个模板，为了定义一个allocator对象，必须指明这个allocator可以分配的对象类型。当一个allocator对象分配内存时，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。

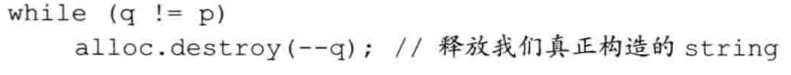




Allocator分配未构造的内存。Allocator分配的内存是未构造的(unconstructed)。我们需要在此内存中构造对象。Construct成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。额外参数用来初始化构造的对象。



Warning：为了使用allocate返回的内存，我们必须用construct构造对象。使用未构造的内存，其行为是未定义的。



Warning：只能对真正构造了的元素进行destroy操作。

一旦元素被销毁后，就可以重新使用这部分内存来保存其他string，也可以将其归还给系统。释放内存通过调用deallocate完成，alloc.deallocate(p,n)，传递给deallocate的指针不能为空，它必须指向由allocate分配的内存，而且传递给deallocate的大小参数必须与调用allocated分配内存时提供的大小参数具有一样的值。

