**关键字**

**const**

顶层const表示指针本身是一个常量，底层const表示指针所指对象是一个常量。

std::string isbn() const{}中的const关键字的作用是修改隐式this指针的类型。这个成员函数不能改变调用它的对象的内容。

**constexpr**

constexpr size\_t scale(size\_t cnt){return new\_sz() \* cnt;} scale(2)是常量表达式；int i = 2;scale(i)不是常量表达式。

**extern**

如果想声明一个变量而非定义它，就在变量名前添加关键字extern，而且不要显示地初始化变量。

**变量类型：**

有两种方法用于定义类型别名，传统使用typedef，新方法使用using。

auto decltype

**类型转换**

类型转换，不包含底层const，使用static\_cast。

包含底层const使用const\_cast。

**机制：**

**指针与引用**

指针与引用相比有很多不同点：

其一，指针本身就是一个对象，允许对指针赋值和拷贝，而且在指针的生命周期内它可以先后指向几个不同的对象。

其二，指针无须在定义时赋初值。和其他内置类型一样，在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值。

**IO**

iostream定义了用于读写流的基本类型，fstream定义了读写命名文件类型，sstream定义了读写内存string对象的类型

istream，ostream，iostream

ifstream，ofstream，fstream

istringstream，ostringstream，stringstream

4种IO库条件状态：strm::badbit,failbit,eofbit,goodbit

**顺序容器**

六种顺序容器：vector,deque,list,forward\_list,array,string

string和vector将元素保存在连续的内存空间中。因此，由下标来计算地址是非常快速的，但是在中间位置添加或删除元素就会非常耗时。

list和forward\_list设计目的是令容器任何位置的添加和删除操作都很快速，代价是不支持元素的随机访问。与其他容器相比额外内存开销大。

deque是一个更为复杂的数据结构，支持快速的随机访问。在deque的中间位置添加或删除元素代价很高。但是，在deque的两端添加或删除元素很快。

array是一种更安全，更容易使用的数组类型，array是固定大小的

除了顺序容器外，标准库还定义了三个顺序容器适配器：stack,queue和priority\_queue。

**Lambda和bind**

四种可调用对象：函数，函数指针，重载了函数调用运算符的类和lambda表达式。

一个lambda表达式的形式：[capture list](parameter list) -> return type{function body}

值捕获：被捕获的变量的值是在lambda创建时拷贝，而不是调用时拷贝。

引用捕获：auto f2 = [&v1]{return v1;}这个v1是可以随时变化的。

隐式捕获：可以让编译器根据lambda体中的代码来推断我们使用哪些变量，&告诉编译器采用捕获引用方式，=表示采用值捕获方式。

如果只有一个return语句可以让编译器推断返回值类型，反之需要自己指定返回值类型。

[](int i) -> int {if(i < 0) return -i; else return i;}

auto g = bind(f, a, b, \_2, c, \_1);调用g( \_1, \_2)映射为f( a, b, \_2, c, \_1)

**关联容器**

两个主要的关联容器：map和set。

按关键字有序保存元素：map,set,multimap和multiset

无序集合：unordered\_map,unordered\_set,unordered\_multimap和unordered\_multiset

**智能指针**

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。weak\_ptr指向shared\_ptr所管理的对象。

每个shared\_ptr都有一个关联的计数器，通常称其为引用计数。一旦一个shared\_ptr的计数器变为0，它就会自动释放自己所管理的对象。即使有weak\_ptr指向对象。

与shared\_ptr不同，某个时刻只能有一个unique\_ptr指向一个给定对象。当unique\_ptr被销毁时，它指向的对象也被销毁。

**拷贝，复制与销毁**

**拷贝构造函数**

如果一个构造函数的第一个参数是自身类类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数。

拷贝初始化是依靠拷贝构造函数或移动构造函数来完成的。

拷贝初始化不仅在我们用=定义变量时会发生，下列情况也会发生：

1，将一个对象作为实参传递给一个非引用类型的形参；2，从一个返回类型为非引用类型的函数返回一个对象；

3，用花括号列表初始化一个数组中的元素或一个聚合类的成员；4，标准库容器调用insert或push成员

**拷贝运算符**

重载运算符本质上是函数，赋值运算符就是一个名为operator=的函数。

例子 Sales\_data& Sales\_data::operator=(const Sales\_data &rhs){}

**析构函数**

析构函数有一个函数体和一个析构部分。

在一个析构函数中，首先执行函数体，然后销毁成员。成员按初始化顺序的逆序销毁。

需要析构函数的类也需要拷贝和赋值操作。需要拷贝操作的类也需要赋值操作，反之亦然。

我们可以通过将拷贝控制成员定义为=default来显示地要求编译器生成合成版本。

在函数的参数列表后面加上=delete来指出我们希望将它定义为删除的。

析构函数不能是删除的成员。

**移动构造函数，移动赋值运算符**

为了支持移动操作，新标准引入了一种新的引用类型--右值引用。

我们通过&&而不是&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摧毁的对象。

int &&rr1 = 42;//正确 int &&rr2 = rr1;//错误：表达式rr1是左值。

我们可以通过调用一个名为move的新标准库函数来获得绑定到左值上的右值引用。

StrVec::StrVec(StrVec &&s) noexcept:elements(s.elements),first\_free(s.first\_free),cap(s.cap){

s.elements = s.first\_free = s.cap = nullptr;

}

如果一个类定义了自己的拷贝构造函数，拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符了。

只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非static数据成员都可以移动时，编译器才会为它合成移动构造函数或移动赋值运算符。

**重载运算符**

重载的运算符是具有特殊名字的函数，它们的名字由关键字operator和其后要定义的运算符号共同组成。

如果一个运算符函数是成员函数，则它的第一个（左侧）运算对象绑定到隐式的this指针上，因此成员运算符函数的（显式）参数数量比运算符的运算对象总数少一个。

具有对称性的运算符可能转换任意一端的运算对象，例如算术，相等性，关系和位运算等，因此它们通常应该是普通的非成员函数。

为了与其他输出运算符保持一致，operator<<一般要返回它的ostream形参。

输入输出运算符必须是非成员函数，而不能是类的成员函数。否则，它们的左侧运算对象将是我们的类的一个对象。

通常情况下，我们把算术和关系运算符定义成非成员函数以允许对左侧或右侧的运算对象进行转换。

如果类定义了算术运算符，则它一般也会定义一个对应的复合复制运算符。

关联容器和一些算法要用到小于运算符，所以定义operator<会比较有用。

**虚函数**

对于某些函数，基类希望它的派生类各自定义适合自身的版本，此时基类就将这些函数声明为虚函数。

virtual double net\_price(std::size\_t n) const;

派生类可以在这样的函数之前加上virtual关键字，但是并不是非得这么做。

C++11新标准允许派生类显示地注明它将使用哪个成员函数改写基类的虚函数，使用ovveride关键字。

在C++语言中，当我们使用基类的引用或指针调用一个虚函数时将发生动态绑定。

基类通常都应该定义一个虚析构函数，即使该函数不执行任何实际操作也是如此。

成员函数如果没有被声明为虚函数，则其解析过程发生在编译时而非运行时。

在类名后面跟一个关键字final可以防止继承的发生。

我们可以将基类的指针或引用绑定到派生类对象上。

和内置指针一样，智能指针类也支持派生类类向基类的类型转换。

动态绑定只有当我们通过指针或引用调用虚函数时才会发生。

我们可以把某个函数指定为final，则之后的任何常识覆盖该函数的操作都将引发错误。

和普通的虚函数不一样，一个纯虚函数无须定义。

我们通过在函数体的位置书写=0就可以将一个虚函数说明为纯虚函数。其中=0只能出现在类内部的虚函数说明语句处。

含有纯虚函数的类是抽象基类。

**模版**

template <typename T>

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

编写泛型代码的两个重要原则：1，模版中的函数参数是const的引用；2，函数体中的条件判断仅使用<比较运算

template <typename T> class Blob{}

就像我们能为函数参数提供默认实参一样，我们也可以提供默认模版实参。

在大型系统中，在多个文件中实例化相同模板的额外开销可能非常严重。

在新的标准中，可以通过显示实例化来避免这种开销。

extern template class Blob<string>;

template int compare(const int&, const int&);//