多线程访问(有写操作)同一个shared\_ptr需要同步，否则会有race condition发生。因此多线程读写shared\_ptr所指向的同一个对象，需要加锁保护。智能指针里的数据不保证是线程安全的，只是保证了指针正确释放的线程安全，也就是引用计数器的线程安全

std::lock\_guard<std::mutex> lock(g\_i\_mutex);

shared\_ptr<int> sp = std::make\_shared<int>(100);

\*sp = \*sp+1;

智能指针简单实现：

template <typename T>

class SmartPtr

{

Public:

SmartPtr(T\* p);

~SmartPtr();

SmartPtr(const SmartPtr<T>& orig);

SmartPtr<T>& operator=(const SmartPtr<T>& rhs);

Private:

T\* ptr;

int\* use\_count;

};

template<typename T>

SmartPtr<T>::SmartPtr(T\* p): ptr(p)

{

try

{

use\_count = new int(1);

}

catch(…)

{

delete ptr;

ptr = nullptr;

use\_count = nullptr;

exit(1);

}

}

template<typename T>

SmartPtr<T>::~SmartPtr()

{

If (--(\*use\_count) == 0)

{

delete ptr;

delete use\_count;

ptr = nullptr;

use\_count = nullptr;

}

}

template<typename T>

SmartPtr<T>::SmartPtr(const SmartPtr<T>& orig)

{

ptr = orig.ptr;

use\_count = orig.use\_count;

++(\*use\_count); //所有引用这个内存的智能指针共用这块use\_count内存

}

template<typename T>

SmartPtr<T>& SmartPtr<T>::operator=(const SmartPtr<T>& rhs)

{

++(\*rhs.use\_count);

if (--(\*use\_count) == 0)

{

delete ptr;

delete use\_count;

}

ptr = rhs.ptr;

use\_count = rhs.use\_count;

return \*this;

}

智能指针如何实现？什么时候改变引用计数？

构造函数中计数初始化为值1;拷贝构造函数中计数值加1;将shared\_ptr作为函数参数传递时计数加一；当作为函数的返回值时加一； 赋值运算符中，左边的对象引用计数减一(减1后可能导致左边析构)，右边的对象引用计数加一；析构函数中引用计数减一 ;在赋值运算符和析构函数中，如果减一后为0，则调用delete释放对象

weak\_ptr是一种弱引用指针，不影响引用计数，从而解决循环引用的问题。它不指向对象的共享内存，但是可以检测到所管理的对象是否已经被释放，避免非法访问。和shared\_ptr之间相互转化，shared\_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared\_ptr(即不能通过weak\_ptr直接访问对象),或者use\_count()来判断是否有效。

unique\_ptr:在异常发生时可以帮助避免资源泄露的智能指针。这个指针实现了独占拥有的概念,意味着可以确保一个对象和其相应资源同一时间只被一个指针拥有。一旦拥有者被销毁或变成空,或开始拥有另一个对象,先前拥有的那个对象就会被销毁，其任何资源也会被释放。

unique\_ptr<string>p3 (new string(‘auto’)) unique\_ptr<string> p4(p3); p4 = p3; //编译报错,unique\_ptr不支持拷贝，赋值操作。而是通过reset,release操作释放所有权

unique\_ptr所有权转移： u.release()//u放弃对指针的控制权，返回指针，并将u置空；u.reset()//释放u指向的对象；u.reset(q)//将u指向q这个裸指针。

虽然不能拷贝赋值unique\_ptr,但是可以通过调用release或者reset将指针的所有权从一个unique\_ptr转移给另一个unique\_ptr;

unique\_ptr<Test> p1(new Test(“case1”));

unique\_ptr<Test> p2(p1.release()); //将所有权从p1转移到p2,p1现在指向nullptr。

unique\_ptr<Test> p3(new Test(“case2”));

p2.reset(p3.release()); //p2释放了原来指向的内存，指向了p3指向的内存。

返回unique\_ptr:不能拷贝unique\_ptr的规则有一个例外：可以拷贝或者赋值一个将要被销毁的unique\_ptr。其本质是调用了移动拷贝和移动赋值；最常见的例子就是从函数返回一个unique\_ptr(一个临时对象)

unique\_ptr<Test>retDying(string param)

{return unique\_ptr<Test>(new Test(param));} //返回即将被销毁的uniptr

unique\_ptr<Test>retTemp(string param)

{unique\_ptr<Test>pTemp(new Test(param)); return pTemp;}//返回局部对象

unique\_ptr<Test>ret1 = retDying(“dying”);

unique\_ptr<Test>ret2 = retTemp(“temp”));

auto\_ptr(程序中应该用unique\_ptr替换)。特征：对象的所有权是独占性的，不能共享所有权，不能放入容器不能通过复制初始化。

不能将auto\_ptr放入到标准容器中。标准容器的拷贝行为，会导致auto\_ptr内的对象被释放，造成难以发觉的错误。同理，unique\_ptr也是这样。但可通过技巧：vec.push\_back(std::move(unique\_ptr))移动资源至容器而不是复制资源至容器

总结:unique\_ptr和auto\_ptr。auto\_ptr可以随便赋值，但赋值后原来的对象就报废了。unique\_ptr干脆不让随便赋值，复制。硬要复制赋值，只能std::move，这样原来对象同样报废。可移动但是不可拷贝使得我们可以保证，同一时刻，只有一个对象与资源相关联，可以在对象之间传递所有权。

应用场景：

使用shared\_ptr明确说明这个对象会被多个指针引用。当不确定是否会被多个指针引用时，使用unique\_ptr

vector:动态数组,元素存储在一块连续的存储空间中，与常规数组不同的是:可以自动增长或缩小存储空间；与其他标准顺序容器相比,vector更有效访问容器内的元素和在末尾添加和删除元素;而在其他位置添加和删除元素(insert,erase),效率不及其他顺序容器；

容器的大小(vector::size())和容器的容量(vector::capacity())是有区别的:size是指容器内元素的个数，capacity是容器预分配的空间大小(分配新存储空间之前可以存储的元素总数)。capacity一般大于等于size。容量多余容器大小的部分用于以防容器大小的增加使用。每次重新分配内存都会很影响程序的性能，所以一般分配的容量都大于容器的大小；若要自己指定分配的容量的大小，则可以使用vector::reserve()，vector::resize() 和容器的size息息相关。调用resize(n)后,容器的size即为n,至于是否影响capacity，取决于调整后的容器的size是否大于capacity; reserve()函数和vector的capacity息息相关。调用reserve(n)后，若容器的capacity<n,则重新分配内存空间，把之前vector中的对象通过copy constructor复制到新的内存空间，销毁之前的内存。从而使得capacity等于n.如果capacity>=n呢？vector会忽略它，这个调用什么也不做

resize和reserve的区别：vector调用resize()后,所有的空间都已经初始化了,所以可以直接访问。而reserve()函数与分配的空间没有被初始化,所以不可访问。调用reserve永远也不会减少容器占用的内存空间(只会增大，或者不变)

vector和list的区别，应用，越详细越好：

vector:连续存储的容器,动态数组，在堆上分配空间；底层实现:数组；

两倍容量增长:vector插入新元素时，未超过当时容量，那么直接添加到最后，然后调整迭代器。如果没有剩余空间了，则会重新配置原有元素个数的两倍空间，然后将原空间元素通过赋值的方式初始化新空间，再向新空间增加元素，最后释放原空间。为什么两倍空间扩容，而不是每次增长一个固定大小的容量呢？2倍增长：平均每次push\_back操作的时间复杂度为常量时间。若是增长固定值：平均每次push\_back操作的时间复杂度为O(n)。因此使用2倍方式扩容。

list: 动态双端链表，在堆上分配空间，每插入一个元素都会分配空间，每删除一个元素都会释放空间。底层数据结构:双向链表 每个节点有信息快，前驱指针，后驱指针

性能:随机访问性能很差，只能快速访问头，尾节点；插入，删除很快(常数开销)，适用于经常插入删除大量数据。相对于vector占用内存多(两个指针)

forward\_list:单向链表

array：固定大小的数组

deque(double-ended queue双端队列):可以进行下标访问的顺序容器，它允许在其首尾两端快速插入及删除元素。另外，在deque任一端插入或删除元素不会让当前正在使用的迭代器失效。插入或删除元素-线性O(n)

与vector相反，deque的元素不一定是相邻存储的：典型实现是采用单独分配的固定大小的相邻元素数组，外加一个额外的数组(索引数组)去存储这些空间的首地址，这表示访问必须进行二次指针解引用，与之相比vector的下标访问只需要进行一次。

deque的存储按需自动扩展及收缩。扩张deque比扩展vector的消耗少，因为它不涉及复制当前的元素到新内存位置。deque相比vector有较大的最小内存开销，就算只保存一个元素的deque也必须为其分配至少一个内存数组

使用reserve()函数提前设定容量大小:对于有大量的数据需要进行push\_back，应当使用reserve()函数提前设定其容量大小,否则会出现许多次容量扩充操作导致效率低下；通常两种情况使用reserve来避免不必要的重新分配。1)知道有多少元素将出现在容器中,可以提前reserve适当数量的空间;2)保留可能需要的最大空间,然后添加了所有数据后,再修整掉多余的容量

用swap方法强行释放vector所占内存: vector<int>().swap(vec); 或者 vec.swap(vector<int>())，即和空的容器交换;

在C++11中可以调用shrink\_to\_fit()来要求deque,string,vectir退回不需要的内存空间(使容器的size大小等于capacity),此函数指出不再需要任何多余的内存空间(标准库并不保证一定退回内存空间)

函数clear()删除存储在容器中的元素。如果元素是对象(object)，stl将为每个元素调用各自析构函数。如果存储的是指向对象的指针，stl不会调用元素对应的析构函数，应该在循环中依次主动调用元素的析构函数

调用clear()后，vector的size变成zero,但它的容量(capacity)并不发生变化，因此clear本身并不释放内存

map: template <class Key, class T, class Compare = less<Key>> class map;

std::map<int, string, less<string>> ;

struct cmp{ bool operator() (const string& s1, const string& s2) return s1 < s2;}

map<string, int, cmp> m ; //通过仿函数对key定义顺序

struct Key{

Int id;

bool operator< (const Key& ref)const { return id < ref.id;}

};

map<Key,string>m; //通过自定义关键字排序顺序 bool operator<

若想按照map的value排序: 由于map不能利用std::sort()直接排序(std::sort()只能用在线性容器vector,list,deque)，所以间接进行。

惯用套路:typedef std::pair<string,int> PAIR;

Struct CMP{

Bool operator() (const& PAIR& p1, const PAIR& p2) {return p1.second < p2.second;}

};

std::vector<PAIR>vec(m.begin(),m.end());

std::sort(vec.begin(),vec.end(),CMP());

构造函数中能否抛出异常？不能

构造函数中抛出异常，会导致析构函数不能被调用，但对象本身已申请到的内存资源会被系统释放。因为析构函数不能被调用，所以可能会造成内存泄露或系统资源未被释放。

析构函数能抛出异常吗？不能

不要在析构函数中抛出异常。虽然c++并不禁止析构函数抛出异常，但这样会导致程序过早结束或出现不明确的行为 。

如果一个析构函数可能抛出异常，析构函数应该捕捉任何异常，然后吞下它们(不传播)或结束程序

void\*可以存储任意类型的指针，使用时强制转化成对应类型的指针： void\* p1; char\* p2 = “hellp”; p1 = p2;但void\*的类型并不能无需类型转换直接赋值给其他类型；因此: char\* p = (char\*)malloc(4); void\* 类型不能做运算： void\* p; p++;是不对的

四种类型转换：

const\_cast 用于将const变量转为非const

static\_cast：用于基本数据类型之间的转换(int转char,int转enum, 指针类型转为void\*;char\* c = new char; void\* p = static\_cast<void\*>(c))用于类层次结构中基类和派生类之间指针或引用的转换：进行上行转换(把派生类的指针或引用转换为基类是安全的)Base\* pbase = static\_cast<Base\*>(pDerived);，用于下行转换(把基类指针或引用转换为派生类时，由于没有动态类型检查，所以是不安全的)

dynamic\_cast:

dynamic\_cast<type\*>(e):e是一个有效的指针,dynamic\_cast<type&>(e):e是一个有效的引用；如果dynamic\_cast的转换目标是指针类型并且失败了，则结果为0；如果转换目标是引用类型并且失败了，则dynamic\_cast运算符抛出一个std::bad\_cast异常；若e是一个空指针，那么返回空指针。dynamic\_cast主要用于类层次间的上行转换和下行转换。在类层次上行转换时,dymanic\_cast和static\_cast的效果是一样的;在进行下行转换时,dynamic\_cast具有类型检查功能,比static\_cast更安全,耗费重大运行成本。

reinterpret\_cast: (重新解释)几乎什么都能转，比如将int转指针；将一种对象类型转换为另一种对象类型，不管它们是否相关

为什么不用C的强制转换？转换不够明确，不能进行错误检查，容易出错。

内联函数与宏：

宏定义在预编译时进行宏替换；内联函数在编译阶段，在调用内联函数的地方进行替换，减少了函数的调用过程(减少函数进栈出栈)，但使得编译文件变大。适合简单函数；内联函数比宏定义安全，内联函数可以检查参数，宏定义只是文本替换。

内联函数与虚函数的讨论：内联函数不能是虚函数

函数的inline属性是在编译时确定的(静态行为)，而virtual的性质是在运行时确定的(动态行为)，两者不能同时存在，只能有一个选择。

inline函数表示该函数是内联的，它建议编译程序在调用该函数的地方直接将函数的代码展开并插入到caller的代码中。

inline的函数内部结构:一般不包含循环。没有复杂的函数调用。

虚函数肯定不能被内联毋庸置疑，因为虚函数只有到了runtime才能被识别到底是哪一个被调用，而内联是编译时期就会将代码展开并安插。这种特性阻止了虚函数的内联。但在语言规则上，虚函数可以用inline修饰。只是这样的修饰不产生作用而已。

new和malloc的区别：

new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要头文件支持

使用new操作符申请内存无需指定内存块的大小，编译器会根据类型信息自行计算。malloc需要显示指出内存的尺寸。返回类型不同：new分配内存成功返回的是对象类型的指针，类型严格与对象匹配，无需进行类型转换。malloc内存分配成功返回的是void\*,需要通过强制类型转换将void\*指针转换成需要的类型。自定义类型：new调用构造函数,delete调用析构函数；malloc和free无法对自定义类型对象构造和析构。

重载:c++允许重载new/delete操作符，而malloc不允许重载。内存区域：new操作符从自由存储区(free store)上为对象动态分配内存，而malloc函数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是OS的术语，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的内存。自由存储区不仅可以是堆，还可以是静态存储区。

内存分配失败时返回值：new失败时，会抛出bad\_alloc异常，不返回NULL;malloc失败时返回NULL。int\* a = new int;if (nullptr == a) {...} else{...}这样没有意义。因为new不会返回nullptr。如果失败一定抛出异常。应该使用异常机制。

try{ int\* a = new int();}catch (bad\_alloc){...}

使用new 操作符分配对象内存步骤：

调用operator new函数(operator new [])分配一块足够大，原始的内存空间。

编译器运行相应的构造函数以构造对象。

对象构造完成后，返回一个指向对象的指针。

使用delete操作符释放对象内存：

调用对象的析构函数

编译器调用operator delete(或operator delete[])函数释放内存空间

new和delete中调用了malloc和free。反之，不行。operator new delete可以被重载,malloc/free不允许被重载。

realloc: 使用malloc分配内存后，如果发现使用过程中内存不足，可以realloc函数进行内存重新分配实现内存的扩充。realloc先判断当前的指针所指向的内存是否有足够的连续内存空间，如果有，原地扩大可分配的内存地址，并且返回原来的地址指针。如果空间不足，先按指定大小重新分配内存，将原数据拷贝到新分配的内存区域，而后释放原来的内存区域。

堆和栈的区别:

空间大小：栈的内存空间是连续的，空间大小通常是系统预先规定好的，即栈顶地址和最大空间是确定的。而堆的内存空间是不连续的。

管理方式:栈由编译器自动分配地和释放；堆由程序员手动分配释放

碎片问题:栈的内存空间连续，先进后出保证无碎片；堆每次在空闲链表遍历寻找分配节点，频繁new会产生大量碎片。

分配效率：栈由专门指令执行效率高；堆需要额外处理，效率低

栈效率高的原因:栈是操作系统提供的数据结构，计算机底层对栈提供了一系列支持:分配专门的寄存器存储栈的地址，压栈入栈有专门的指令执行；而堆是C++函数库提供的，需要一系列分配内存，合并内存和释放内存的算法，因此效率低。

malloc函数实现:肯定是从堆中分配内存；malloc函数的实质是它有一个可用内存块的空闲内存块链表(OS有一个记录空闲内存地址的链表)。调用malloc函数时，沿着链表寻找到一个足够大的内存块。将内存块一分为二(一块的大小与用户申请的大小相等，另一块的大小就是剩下的字节)，将分配给用户的那块内存传给用户，并将剩下的那块(如果有的话)返回到连接表上。调用free函数时,将用户释放的内存块连接到空闲链表上。到最后，空闲链表被切成很多小内存块。于是，malloc()函数请求延时，并开始在空闲链表上检查各内存片段，对它们进行内存整理，将相邻的小空闲块合并成较大的内存块。

一般在堆的头部用一个字节存放堆的大小。这样delete[]时可以用到。

函数调用过程：在函数调用时，第一个进栈的是主函数的下一条指令(被调用函数的下一条指令)，然后是函数的各个参数，参数由右往左进栈，然后是函数中的局部变量(静态变量不入栈)，当本次调用结束后，局部变量先出栈，然后是参数出栈，最后栈顶指针指向最开始的地址(下一条指令的地址)，程序由该位置继续运行。

STL中内存池实现：

内存池出现原因：内存碎片。每次new T时，1)内存分配(malloc),2)构造函数初始化内存。内存分配耗时，因此一次性分配一大块内存(一次分配，多次使用，提高了效率)，而内存池就是管理这一大块内存的数据结构。频繁使用new导致系统内存空间碎片化严重，容易导致很难找到一块连续的大块内存，造成内存碎片，空间利用率低。

内存申请流程:小于等于128K的用第二级分配器；大于等于128K的用第一级分配器

第一级分配器：第一级采用malloc, free

第二级分配器：使用自由链表(free-list),有16个free-lists，各自管理大小分别为8K，16K，24K，32K，40K，48K，56K，64K，72K…….120K,128K bytes的小额区块

内存池分配方法:

1)使用allocate向内存池请求size大小的内存空间，如果需要请求的内存大小大于128bytes直接使用malloc

2)如果需要的内存小于128Kbytes,allocate根据size找到最合适的自由链表(free-lists) a)如果链表不为空,返回第一个node,链表头改为第二个node b)如果链表为空，使用blockAlloc请求分配node; 如果内存池中有大于一个node的空间,分配尽可能多的(最多20个)的node,将一个node返回，其它的node添加到链表中。如果内存池没有一个node,再次向操作系统分配内存(找不到空间，就抛出异常)

3)用户调用deallocate释放内存空间，如果要求释放的内存空间大于128bytes，直接调用free；小于128bytes则加入相应的自由链表中而不是直接还给操作系统

函数模板与模板函数 类模板与模板类

模板函数是函数模板的实例化；模板类是类模板的实例化

模板特化：

template<> ; 模板特例分为全特化(函数模板特化，类模板特化)和偏特化。模板特化的目的就是对于某一种变量类型具有不同的实现，因此需要特化版本。偏特化(仅支持类模板的偏特化)：偏特化本质上是指定部分类型，还是一个模板，全特化本质上是一个实例

函数模板的特化：每个模板参数都要提供参数

函数重载与模板特化：模板特化本质上是一个实例，而非函数名的一个重载版本。

类模板的特化：分为全特化和部分特化。部分特化不必为所有模板参数提供实参，可以只指定一部分而非所有模板参数。

为什么模板的声明和实现都放在头文件中？

对于普通cpp文件，编译器编译时，看到函数声明即可编译(链接到)。对于模板文件(模板实现放在cpp文件)，由于不能决定模板参数的类型，所以编译器不能生成模板的实例，此时调用模板失败。而实现放在头文件中，实例化时，既能看到声明，又能找到模板实现。

int& int::operator++() //++I {\*this += 1; return \*this;} //i++ int int::operator (int){ int oldvalue = \*this; ++(\*this); return oldvalue;}

static关键字作用:

最重要一点：隐藏(static变量，static函数)同时编译多个文件时，所有未加static的全局变量和全局函数都具有全局可见性，在其他源文件中可被访问。加了static，只在当前文件中可访问，可以避免命名冲突; static变量的记忆功能和全局生存期：程序刚开始运行时完成初始化，唯一一次初始化，生存期为整个源程序；static变量默认初始化为0;类成员声明为static:是类的成员不是对象的成员，因为没有this指针

数组指针区别:数组一旦声明，就不能再赋值，但指针可以赋值；指针所指的字符串被理解为常量，不能修改，数组定义的字符串可以修改。sizeof可以计算数组变量的数组大小，但sizeof(char\*)是指针大小。

static成员函数为什么不能是虚函数:

虚函数依靠vptr和vtable来处理：类对应vtable，每个对象有自己的vptr,(vtable和vptr都是)在类的构造函数中创建生成，并且只能用this指针来访问，因为vptr是类的一个成员，并且vptr指向保存虚函数地址的vtable.; 静态成员函数没有this指针,所以无法访问vptr,因此static函数不能为virtual; 虚函数调用关系: this->vptr->vtable->virtual function

拷贝构造函数使用场合：

定义变量并且初始化(2种)：AA a1; AA a2 = a1; 或者AA a2(a1);

将对象作为参数传给函数参数(非引用): void func(AA a1); AA a;func(a);

将对象作为返回值: AA func(){ return AA a1;return a1;}

拷贝构造函数为什么要用引用？

实例化一个对象，会调用拷贝构造函数，拷贝构造函数又会调用自己，形成无限递归。

构造函数不能是虚函数？

调用虚函数都是通过虚指针找到虚表中的虚函数；在虚指针没有正确初始化之前，不能调用虚函数。虚指针是在构造函数中初始化，虚表的创建是在编译阶段。

构造函数中不要调用虚函数：vptr的初始化在构造函数中，在调用到基类的构造函数时，派生类对象还不是一个完整的对象，vptr没有完全初始化，无法读到vtbl中的虚函数。此时调用的是不起virtual作用的函数

RTTI(运行时类型识别):通过typeid运算符和dynamic\_cast运算符实现

C++通过两个操作符提供RTTI: typeid运算符:返回其表达式或类型名的实际类型；dynamic\_cast运算符，该运算符将 基类的指针或引用安全的转换为派生类类型的指针或引用

C++程序构建(build)4个过程：预编译，编译，汇编，链接

预处理:头文件，宏；编译:将预处理后的文件转为汇编代码，生成汇编文件

汇编：将汇编文件生成机器码,即目标文件；链接：将目标文件和库连接生成可执行文件

extern 作用：

extern置于变量或者函数前，标识变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其它模块中寻找其定义。

c++中使用C代码：在C++中调用C的代码必须把原来的C语言声明放到extern "C"中,否则在C++中无法编译通过；原因:C和C++具有完全不同的编译和链接方式。C语言编译器编译函数时不带函数类型和作用域信息，只包含函数符号名字。C++为了实现函数重载，在编译时会带上函数的类型和作用域信息；使用extern C是为了告诉编译器按照C的翻译规则。

例如:int func(int a, int b);c编译器把函数翻译成类似\_Func的符号，C链接器只要找到这个符号就可以成功。在C++中编译器会检查参数类型，会被翻译成\_Z\_Func\_int\_int的符号

STL基本组成(六大组件):容器，迭代器，仿函数，算法，分配器，适配器

之间关系:分配器给容器分配存储空间，算法通过迭代器获取容器中的内容，仿函数协助算法完成各种操作，适配器用来适配仿函数

迭代器模式(iterator):运用于聚合对象的一种模式，通过运用该模式，使得可以在不清楚对象内部表示的情况下，按一定顺序访问聚合对象中的各个元素。

仅支持底层聚合类，如list,vector,stack

迭代器和指针区别：

迭代器不是指针，是类模板，表现的像指针。通过重载指针的一些操作符:->,\*,++,--等。迭代器返回的是对象引用而不是对象的值

迭代器失效：

对于序列容器(vector,deque)来说，使用erase(itertor)后，erase返回下一个有效的迭代器；对于list:erase返回下一个有效的迭代器；对于关联容器map,set来说，返回下一个元素的迭代器。

STL unique: 其功能是去除相邻的重复元素(只保留一个，若不相邻要先sort)，返回去重之后的尾地址。所以使用前需要对数组进行排序。它并没有实现将重复的元素删除，而是将不重复的元素放在了最前面，后半部分保持排序后的元素。

重载前置++:

Test& operator++(){}

重载后置++ Test operator++(int){}

C++11：

allocator(内存分配器)

=default显示地要求编译器生成合成的版本。

=delete阻止拷贝(声明为private并且不定义)：对某些类来说，拷贝构造函数和拷贝赋值运算符没有意义。可以通过将拷贝构造函数和拷贝赋值运算符定义为删除的函数来阻止拷贝，在声明函数后面加=delete来指出希望将它定义为删除的。析构函数不能是删除的成员。

final关键字:

修饰类声明(放在类声明后，表示这个类不能被继承)

修饰虚函数(放在虚函数声明后，表示这个虚函数不能被继承)

移动构造函数(避免内存拷贝:新标准一个最主要的特性是可以移动而非拷贝对象的能力很多情况下会发生对象拷贝，在其中某些情况下，对象拷贝后就立即被销毁了。因此移动而非拷贝对象会大幅提升性能。另一个原因是IO类或者unique\_ptr这样的类包含不能共享的资源，这些类禁止复制。 旧标准中，容器保存的类必须是可以拷贝的，但在新标准中，容器也可以保存不可拷贝的类型，前提是它们能被移动)：

左值右值的区别？

左值是持久对象，存在于内存中；右值要么是字面常量，要么是临时创建的对象

引入右值的目的是为了效率避免内存拷贝。

左值引用(传统的引用)：一个引用指向一个左值;int &a = b;因为b是左值;例外:const引用。const int& a = 10; 10是右值，但必须是const

右值引用(c++11,指向右值的引用，要被销毁的对象，字面常量)：消除两个对象交互时不必要的对象拷贝；不能将右值引用直接绑定到一个变量上，即使这个变量是右值引用类型也不行。

std::move：右值引用必须绑定到右值上，但可以通过std::move绑定左值

虽然不能将一个右值引用直接绑定到一个左值上，但可以调用std::move将一个左值转换为右值引用类型: int &&r2 = std::move(r1);//r1转换后就不能使用了

要实现对象的move,类要支持移动构造函数和移动复制运算

移动构造函数和移动赋值运算符(窃取资源而不是拷贝资源,不分配任何资源)：

移动构造函数的第一个参数是该类型的一个右值引用，任何额外的参数都必须有默认实参。StrVec(StrVec&&);

移动赋值运算符:StrVec& StrVec::operator=(StrVec&& rhs);

如果一个类有一个拷贝构造函数而没有移动构造函数，则其对象是通过拷贝构造函数来”移动”的。拷贝赋值运算符和移动赋值运算符也是这样。

std::bind : 返回一个可调用对象

auto newCallble = bind(callable, arg\_list)

当调用newCallble时，newCallble会调用callble

auto check6 = bind(check\_size,std::placeholders::\_1,6); check6(“aaa”);

若类重载了调用运算符，则该类的对象称为函数对象

std::function: 用来存储可调用对象，这些可调用对象必须有相同的调用形式(返回类型，形参表);function是一个模板，使用时必须为模板提供类型function<int(int,int)> fp; fp(“aaa”);//调用

可变参数模板：

atomic原子操作:用于多线程互斥操作。类似于mutex，但原子操作更加接近底层，因而效率更高。C11中atomic\_bool, atomic\_int，多线程中对这些类型的共享资源进行操作，编译器保证这些操作都是原子性的。从而避免了使用锁，提高了效率(效率比mutex高)

std::atomic对int,char,bool，也可以atomic<T>生成一个T类型的原子对象，等数据结构进行原子性封装，利用std::atomic可以实现数据结构的无锁设计。

原子操作的实现原理(处理器如何实现原子操作)：

使用总线锁保证原子性：所谓总线锁就是使用处理器提供的一个lock信号。多个处理器同时对共享变量进行读写操作。当一个处理器在总线上输出lock信号时，其它处理器的请求将被阻塞，那么该处理器可以独占共享内存。

使用缓存锁保证原子性：在同一时刻，只需要保证对某个内存地址的操作是原子性即可，但总线锁把CPU和内存之间的通信锁住了(使得其它地址的数据CPU也无法处理)，开销比较大。目前处理器在某些场合下使用缓存锁定来进行优化。

缓存锁定是指内存区域如果被缓存在处理器的缓存行中(如果操作数据没有缓存在处理器中，处理器使用总线锁定)，并且在lock操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不再总线上发出lock信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性。

lambda表达式：定义一个匿名函数，并且可以捕获一定范围内的变量;使用临时的匿名函数可以减少函数数量，让代码变得清晰。

[capture list](parameter list) -> return type {function body}

capture list:一个lambda只有在其捕获列表中捕获一个它所在函数中的局部变量，才能在函数体中使用该变量；值捕获[var]，引用捕获[&var],[=]以值形式捕获所有外部变量,[&]以引用形式捕获所有外部变量,捕获this指针的目的是可以在lambda中使用当前类的成员函数和成员变量。变量在lambda表达式中被捕获，构成了一个闭包。  
指定lambda返回类型：默认情况下，如果一个lambda函数体包含return之外的任何语句,则编译器假设此lambda返回void，即此时lambda不能返回值。所以函数体中多个语句时要指定返回类型。

auto关键字:自动推导类型，但不能用于函数传参以及数组类型推导

for (auto it = vec.begin();it != vec.end();++it){} ; auto ptr = new auto(10);

auto不能用于函数传参 : int add(auto x, auto y);不能用于推导数组类型

auto与decltype的区别：处理const的方式不同 对引用变量的推导结果不同

auto会忽略顶层const(最左边的const),底层const则会保留下来；decltype则会返回该变量的完整类型

const int ci = i, &cr = ci;

auto b = ci;//整数 auto c = cr;//整数 auto d = &i;//整形指针 auto e = &ci;//指向整形常量的指针 decltype(ci) x = 0;//const int型 decltype(cr) y;//错误y是引用，引用必须初始化

对引用变量的不同：auto将引用变量赋给变量后，变量类型为引用变量所对应变量的类型。而decltype则是为引用类型

int i = 0, &r=i; auto c = r;//c整形 decltype(r)d = r;//d为int&引用类型

decltype(返回操作数的数据类型。编译器分析表达式并得到它的类型，却不实际计算表达式的值):从表达式中推断出要定义变量的类型。但不想用表达式的值去初始化变量

decltype(f()) sum = x; //sum的类型就是函数f的返回类型

int i = 42, a, &r = i; decltype(r) b = a; //b必须初始化，因为b的类型是引用

int\* p , i=42; decltype(\*p) c = i; //如果表达式的内容是解引用指针，则decltype将得到引用类型

decltype((variable)) //双层括号的结果永远是引用；而decltype(variable)结果只有当variable本身是一个引用时才是引用

for循环：

int numbers [] = {1,2,3,4,5};

for (auto it : numbers){

cout<<it<<” ”;

}

std::array:

std::array<int,4>arr={1,2,3,4};

for (auto it : arr) cout<<it<<” ”;

std::forward\_list(单向链表): list:双向链表

std::unordered\_map(内部哈希表实现,理论上查找效率为o(1),需要增加哈希表的内存开销)

std::unordered\_set(内部哈希表实现)

=defaulted函数：指示编译器生成该函数的默认实现(少敲键盘,性能更好)

=deleted函数：防止对象拷贝

C11多线程：

线程间通信：std::promise std::future std::async() std::package\_task std::call\_once()

std::future: 提供了一个异步操作结果的机制。字面理解表明未来，因为这个异步操作不是马上就能获得操作结果的，只能在未来某个时候获取。获取future结果的三种方式:get(等待异步操作结束并返回结果),wait(等待异步操作完成，没有返回值),wait\_for(超时等待返回结果)

std::promise:promise对象可以保存某一类型T的值，该值可以被future对象读取。通过get\_future来获取与该promise对象相关联的future对象，对该future对象做get()。在线程函数中给外面传进来的promise赋值，当线程函数执行完，就可以通过promise获取该值。通过调用futureObj = promiseobj.get\_future(); futureObj.get();

std::promise<int> pr;

std::thread t ([](std::promise<int>& p) {p.set\_value(9);}, std::ref(pr))

std::future<int> f = pr.get\_future();

auto r = f.get();

std::package\_task:它包装了一个可调用目标(函数,lambda,std::bind),它有点类似于promise,promise保存了一个共享状态值，而packaged\_task保存的是一个函数

std::packaged\_task<int()> task([](){return 7;})

std::thread t1(std::ref(task));

std::future<int> f1 = task.get\_future();

auto r1 = f1.get(); // 7

std::async:创建一个线程执行传递的任务，这个任务是可调用对象，然后返回一个std::future。这样可以方便获取线程函数的执行结果。(std::async会自动创建一个线程去调用线程函数，它返回一个std::future)。提供了线程的创建策略，可以延迟加载线程。

future存储多线程函数的返回值(return值)，当调用future.get时会阻塞直到绑定的task执行完毕。

当线程启动后，一定要在和线程相关联的thread销毁前，确定以何种方式等待线程执行结束。两种方式：

std::thread ::join() 等待(会阻塞)启动的线程完成，才会继续往下执行。

std::thread ::detach() 启动的线程自主在后台运行，当前线程继续往下执行，不等待新线程结束。

C++11中的锁：

互斥锁(mutex)：某一时刻只能一个线程获得这个锁，其他线程都不能获取该互斥锁。需要调用lock()和unlock()。C++提供了std::lock\_guard类模板，实现了RAII

条件锁：即所谓条件变量。使用场景线程池。std::condition\_variable(和std::mutex一起工作)

std::unique\_lock与std::lock\_guard(自动锁定解锁)区别：

std::lock\_guard是RAII模板类的简单实现：在构造函数中进行加锁，在析构函数中进行解锁。但互斥锁对性能有影响，所以要尽可能减小锁定的区域，也就是使用细粒度的锁。这一点lock\_guard不够灵活，lock\_guard只能保证构造加锁，析构解锁，本身不提供加锁解锁的接口。

std::unique\_lock是一个更加通用的互斥包装器。可以延迟锁定(defer\_lock)，递归锁定，所有权转移，和条件变量一起使用，提供了lock()和unlock()功能。在析构时，可以根据当前的状态决定是否进行解锁(lock\_guard一定会解锁)。更灵活，更强大，但损失性能。

读写锁：

boost::shared\_lock

std::atomic(原子数据类型：不发生数据竞争，能直接用在多线程中而不必对其进行添加互斥资源锁的类型；可以理解为这些原子类型内部自己加了锁)

tuple容器(元祖)：元祖容器

tuple<T1,T2,T3,…..>tup = make\_tuple(…,….,….) tuple<T1,…>tup(….,….)可直接初始化，也可以用make\_tuple初始化

若希望将一些数据结构组合成单一对象，但又不想定义一个新数据结构(struct)来表示这些数据时,tuple很有用

访问元素用get<pos>(tup);获取元素数量tuple\_size<decltype(tup)>::value

常用tuple作为返回值，返回多个值(函数返回一个tuple)

bitset

bitset<32> bitvec(1U) //32位；低位为1，剩余位为0 bitvec.all();//如果所有位为1，返回true bitvec.count()返回其中1的个数 bitvec.any()只要有一个为1就返回true

提取bvitset的值：bitset.to\_ulong()

随机数库：生成不同范围内的随机数 #include <random>通过随机数引擎类和随机数分布类协作

引擎类生成unsigned随机数序列，分布类使用引擎类生成指定类型，在给定范围内的，服从特定概率分布的随机数 两者组合使用。

设置随机数发生器种子：相同的种子会生成相同的序列。要每次生成不同的种子需要每次设置不同的种子。通常种子是通过调用系统函数time(#include <ctime>) default\_random\_engine e(time(0));//设置随机种子

default\_random\_engine e(time(0)); uniform\_int\_distribution<unsigned>u(0,9) cout<<u(e);

STL 算法：

iter search(first1,last1,first2,last2)在序列(first1,last2)的区间中，查找序列(first2, last2)的首次出现点。如果序列一内不存在与序列二完全匹配的子序列，返回序列一的last1

transform(转换运算)：将指定范围内元素应用于给定的操作，并将结果存储在指定的另一个范围内。主要作用是省去了自己写for循环

一元操作：outputIterator transform(InputIterator first1, InputIterator last1,

OutputIterator result, op);//op应用于[first1,last1]范围内的每个元素，并将操作返回的值存储在result开头的范围内，op是函数对象。string str(“hello”); transform(str.begin(),str.end(),::tolower);

二元操作：outputIterator transform(InputIterator first1, InputIterator last1,

InputIterator2 first2, OutputIterator result,op);//实现二元函数的调用

unique:移除相邻的重复元素(必须先排序),返回一个迭代器指向新区间的尾端，新区间内不包含相邻的重复元素。会有残余数据留下来，用erase函数去除

rotate(循环移位函数):rotate(first,middle,last);效果是交换[middle,last)和[first,middle)部分的位置

replace:将[first,last)区间内的所有old\_value替换成new\_value

void replace(first,last, const T&old\_value, const T& new\_value);

line = line.replace(line.find(“&”),1,”1”);//将line中的第一个&替换成1 replace(pos,len,str);

line =line.replace(line.begin(),line.begin()+5,”1”);//将line从begin往后6个位置替换成1

remove(并不真正删除) 序列{0,1,0,2,0,3,0,4},移除所有0值，元素结果是{1,2,3,4,0,3,0,4},1,2,3,4被分别拷贝到第1,2,3,4位置，第四个位置以后不动。

merge(将两个排序后的集合，合并置于另一空间,得到另一个有序空间):

mergesort(first,last){

int mid = (first + last) / 2;

mergesort(first, mid);mergesort(mid+1,last);

inplace\_merge(first,mid,last);(应用于有序区间)

}

lower\_bound(beg,end,k): 返回大于等于k值的第一个位置

upper\_bound(beg,end,k):返回大于K值的第一个位置

equal\_range(beg,end,k):返回一个pair对象,first是lower\_bound,second是upper\_bound

binary\_search()

next\_permutation()

multimap:包含重复的键 multimap::lower\_bound(param)返回一个迭代器,它指向键值和参数相等或大于参数的第一个元素，或指向迭代器末尾multimap::upper\_bound(param),指向键值大于函数参数的第一个元素 multimap::equal\_range(param)返回pair对象,pair中的first指向键关联的第一个实例,second指向键关联的最后一个实例的下一个位置,即equal\_range返回的是[lower\_bound,upper\_bound)

适配器(adapters):在stl组件中，扮演轴承，转换器角色。适配器是一种设计模式(将一个class的接口转换为另一个class的接口，使原本因接口不兼容而不能合作的classes，可以一起运行)

迭代器适配器：插入迭代器，流迭代器，反向迭代器

插入迭代器:back\_inserter(使用push\_back()在尾端插入元素，只有提供了push\_back操作的容器才能使用:vector,deque,list);front\_inserter(内部使用了push\_front()在最前端插入,只有提供了push\_front操作的容器才能使用:deque,list)

vector<int>vec1={1,2,3,4,5},vec2; copy(vec1.begin(),vec1.end(),back\_inserter(vec2));

equal:如果两个序列在[first,last)区间内相等,equal()返回true。如果第二序列的元素比较多，多出来的元素不予考虑。

因此判断两个序列是否完全相等：if (vec1.size()==vec2.size() && equal(vec1.begin(),vec1.end(),vec2.begin())); 或者使用vec1 == vec2

set相关用法：set\_union,set\_intersection,set\_difference, symmetric 参数可以用set或multiset,即集合中的元素可以重复，元素要排序的

C++内存模型：堆，栈，全局/静态存储区，代码段(存放函数可执行代码，)，内存映射段(动态库/静态库，文件映射)，常量区(字符串常量)。

自由存储区和堆有什么区别？malloc在堆上分配内存块，使用free释放内存。new申请的内存在自由存储区上，使用delete来释放。

C++对象模型：

非静态成员变量配置在每一个类对象内部；静态数据成员存放在所有的类对象之外；静态成员函数和普通成员函数放在所有的类对象之外

class Point

{

public:

Point(float val);

virtual ~Point();

float x() const;

static int PointCount();

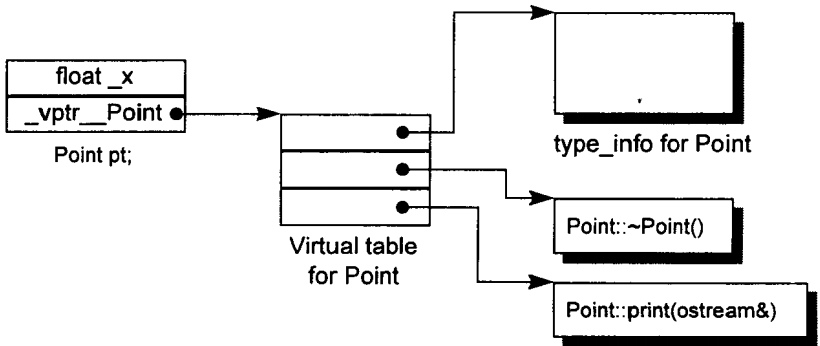
protected:

virtual ostream& print(ostream& os) const;

float \_x;

static int \_point\_count;

};



虚函数：

每个类产生一堆指向virtual function的指针，放在表格中，这个表格成为virtual table(vtbl)虚表在编译时生成。虚表存放在进程的只读数据段。

虚表中的内容：第一行：RTTI(run time type information)运行时类型信息。第二行：虚函数指针的偏移。第三行：虚函数的入口地址。

每个类对象被添加了一个指针，指向相关的virtual table,通常这个指针被成为vptr ，vptr在实例生成构造函数中初始化。

多态性和虚函数表(编译时生成)：

C++如何实现多态？

多态分为静态多态和动态多态。静态多态是通过重载和模板实现，在编译时确定。动态多态通过虚函数和继承关系来实现，执行动态绑定，在运行时确定。

动态多态实现有几个条件：1)虚函数2)一个基类的指针或引用指向派生类的对象。基类指针在调用成员函数(虚函数)时，就会去查找该对象的虚函数表。vptr在每个对象的首地址。查找该虚函数表中该函数的指针进行调用。

每个对象中保存的只是一个虚函数表的指针，c++内部为每一个类维持一个虚函数表，该类的对象都指向这同一个虚函数表

虚函数的实现： 在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址。当子类继承了父类的时候也会继承其虚函数表。当子类重写父类中虚函数的时候，会将继承到的虚函数表中的地址替换成重新写的函数地址。

每一个含有虚函数的类，都会生成虚表。虚表中的虚函数决定了执行此对象的虚成员函数的时候，真正执行的那一个成员函数。

对于有多个基类的类对象，会有多个虚表。每个基类对应一个虚表。同时，虚表的顺序和继承时的顺序相同。

在每一个类对象所占用的内存中，虚指针位于最前边，每个虚指针指向对应的虚表(有多个虚表时)。

当派生类继承含有虚函数的子类时，会复制一份虚函数表，派生类如果有与基类中虚函数同名的虚函数，会在虚函数表中覆盖原来基类的虚函数；如果虚函数不重名，只会在虚函数表中增加一个函数入口。这种机制实现了多态。

class A{public: virtual void funa(){}};

class B{public: virtual void funb(){}};

class C: public A, public B{public virtual void func(){}};

sizeof(C): 8。因为类C多继承，同时继承了类A，类B，因此就复制了两张虚函数表。也就有了两个vptr，所以大小为8。类C也有自己的虚函数，也会放在其中一张虚函数表当中，不会再增加对象大小。

多重继承：继承一个以上的base classes，但若这些base classes又有更高级的base classes，会导致致命的”菱形继承”

虚继承(解决c++重复继承问题的手段，从不同途径继承来的同一基类，会在子类中存在多份拷贝。这将存在两个问题：其一，浪费存储空间；其二，存在二义性问题):

class File{…}; class InputFile: public File {…}; class OutputFile: public File {….}; class IOFile: public InputFile, public OutputFile{…..};

IOFile对象只该有一个File,所以它继承自两个base classes而来的filename不该重复；因此必须令那个带有此数据的class(File)成为一个virtual base class(虚基类：多条继承路径上有一个公共的基类)让所有继承File的class采用virtual继承：class InputFile: virtual public File{…} class OutputFile: virtual public File{…}

菱型继承时，sizeof(类)大小是多少？虚基类指针？虚基类表？

Hashmap为什么占用内存比map多(桶)

Exception(什么情况下用exception,exception对性能有什么影响)

1. 多态实现的原理

a. 传值，非引用或者非指针

b.rtti 原理

c. vptr, 虚表在什么时候构建， vptr在什么时候设置， vptr

2. stl

vector<const int> container;

vector<int&> ref\_vec;

class A: public vector<int>

{

};

vector<int> a;

a.push\_back(1), a.push\_back(2), a.push\_back(3), a.push\_back（4）;

vector<A> a;

扩增的细节