拓跋阿秀：

一、首先我会问你关于C++三大特性（继承、封装、多态），你了解多少

1、比如用自己的话介绍面向对象的三大特性，并且举例说明？

三大特性：继承、封装和多态

（1）继承

让某种类型对象获得另一个类型对象的属性和方法。

它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展

常见的继承有三种方式：

1. 实现继承：指使用基类的属性和方法而无需额外编码的能力
2. 接口继承：指仅使用属性和方法的名称、但是子类必须提供实现的能力
3. 可视继承：指子窗体（类）使用基窗体（类）的外观和实现代码的能力（C++里好像不怎么用）

例如，将人定义为一个抽象类，拥有姓名、性别、年龄等公共属性，吃饭、睡觉、走路等公共方法，在定义一个具体的人时，就可以继承这个抽象类，既保留了公共属性和方法，也可以在此基础上扩展跳舞、唱歌等特有方法

（2）封装

数据和代码捆绑在一起，避免外界干扰和不确定性访问。

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏，例如：将公共的数据或方法使用public修饰，而不希望被访问的数据或方法采用private修饰。

（3）多态

同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为（重载实现编译时多态，虚函数实现运行时多态）。

多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单一句话：允许将子类类型的指针赋值给父类类型的指针

实现多态有二种方式：覆盖（override），重载（overload）。覆盖：是指子类重新定义父类的虚函数的做法。重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。例如：基类是一个抽象对象——人，那教师、运动员也是人，而使用这个抽象对象既可以表示教师、也可以表示运动员。

二、其次我会问你寄到构造函数和析构函数的问题来考察一下你的基础

比如

2、构造函数、析构函数的执行顺序？构造函数和拷贝构造的内部都干了啥？

1) 构造函数顺序

① 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。

② 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。

③ 派生类构造函数。

2) 析构函数顺序

① 调用派生类的析构函数；

② 调用成员类对象的析构函数；

③ 调用基类的析构函数。

3 、虚析构函数的作用，父类的析构函数是否要设置为虚函数?

1) C++中基类采用virtual虚析构函数是为了防止内存泄漏。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。

假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。

那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

2) 纯虚析构函数一定得定义，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调用其每一个虚基类以及上一层基类的析构函数。

因此，缺乏任何一个基类析构函数的定义，就会导致链接失败，最好不要把虚析构函数定义为纯虚析构函数。

4、构造函数析构函数可否抛出异常

1) C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数中发生异常，控制权转出构造函数之外。

因此，在对象b的构造函数中发生异常，对象b的析构函数不会被调用。因此会造成内存泄漏。

2) 用auto_ptr对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危机，不再需要在析构函数中手动释放资源；

3) 如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用terminate函数让程序结束；

4) 如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是没有完成他应该执行的每一件事情。

5、构造函数一般不定义为虚函数的原因

（1）创建一个对象时需要确定对象的类型，而虚函数是在运行时动态确定其类型的。在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型

（2）虚函数的调用需要虚函数表指针vptr，而该指针存放在对象的内存空间中，若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表vtable地址用来调用虚构造函数了

（3）虚函数的作用在于通过父类的指针或者引用调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类或者引用去调用，因此就规定构造函数不能是虚函数

（4）析构函数一般都要声明为虚函数，这个应该是老生常谈了，这里不再赘述

6 、构造函数或者析构函数中可以调用虚函数吗

简要结论：

· 从语法上讲，调用完全没有问题。

· 但是从效果上看，往往不能达到需要的目的。

《Effective C++》的解释是： 派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。 同样，进入基类析构函数时，对象也是基类类型。

举个例子：

```
#include<iostream>

using namespace std;



class Base

{

public:

  Base()

  {

•    Function();

  }



  virtual void Function()

  {

•    cout << "Base::Fuction" << endl;

  }

  ~Base()

  {

•    Function();

  }

};



class A : public Base

{

public:

  A()

  {

   Function();

  }



  virtual void Function()

  {
```

```
    cout << "A::Function" << endl;
  }
  ~A()
  {
    Function();
  }
};

int main()
{
  Base* a = new Base;
  delete a;
  cout << "-------------------------" <<endl;
  Base* b = new A;*//语句1*
  delete b;
}
```

//输出结果

//Base::Fuction

//Base::Fuction

//------------------------

//Base::Fuction

//A::Function

//Base::Fuction

语句1讲道理应该体现多态性，执行类A中的构造和析构函数，从实验结果来看，语句1并没有体现，执行流程是先构造基类，所以先调用基类的构造函数，构造完成再执行A自己的构造函数，析构时也是调用基类的析构函数，也就是说构造和析构中调用虚函数并不能达到目的，应该避免。

三、再然后我会问你一些偏底层的问题，看你是否具有探究问题的想法

比如

7、this指针调用成员变量时，堆栈会发生什么变化?

当在类的非静态成员函数访问类的非静态成员时，编译器会自动将对象的地址传给作为隐含参数传递给函数，这个隐含参数就是this指针。

即使你并没有写this指针，编译器在链接时也会加上this的，对各成员的访问都是通过this的。

例如你建立了类的多个对象时，在调用类的成员函数时，你并不知道具体是哪个对象在调用，此时你可以通过查看this指针来查看具体是哪个对象在调用。This指针首先入栈，然后成员函数的参数从右向左进行入栈，最后函数返回地址入栈。

8、是否可以说一下STL的两级空间配置器是怎么回事?

1、首先明白为什么需要二级空间配置器?

我们知道动态开辟内存时，要在堆上申请，但若是我们需要

频繁的在堆开辟释放内存，则就会在堆上造成很多外部碎片，浪费了内存空间;

每次都要进行调用malloc、free函数等操作，使空间就会增加一些附加信息，降低了空间利用率;

随着外部碎片增多，内存分配器在找不到合适内存情况下需要合并空闲块，浪费了时间，大大降低了效率。

于是就设置了二级空间配置器，当开辟内存<=128bytes时，即视为开辟小块内存，则调用二级空间配置器。

关于STL中一级空间配置器和二级空间配置器的选择上，一般默认选择的为二级空间配置器。 如果大于128字节再转去一级配置器器。

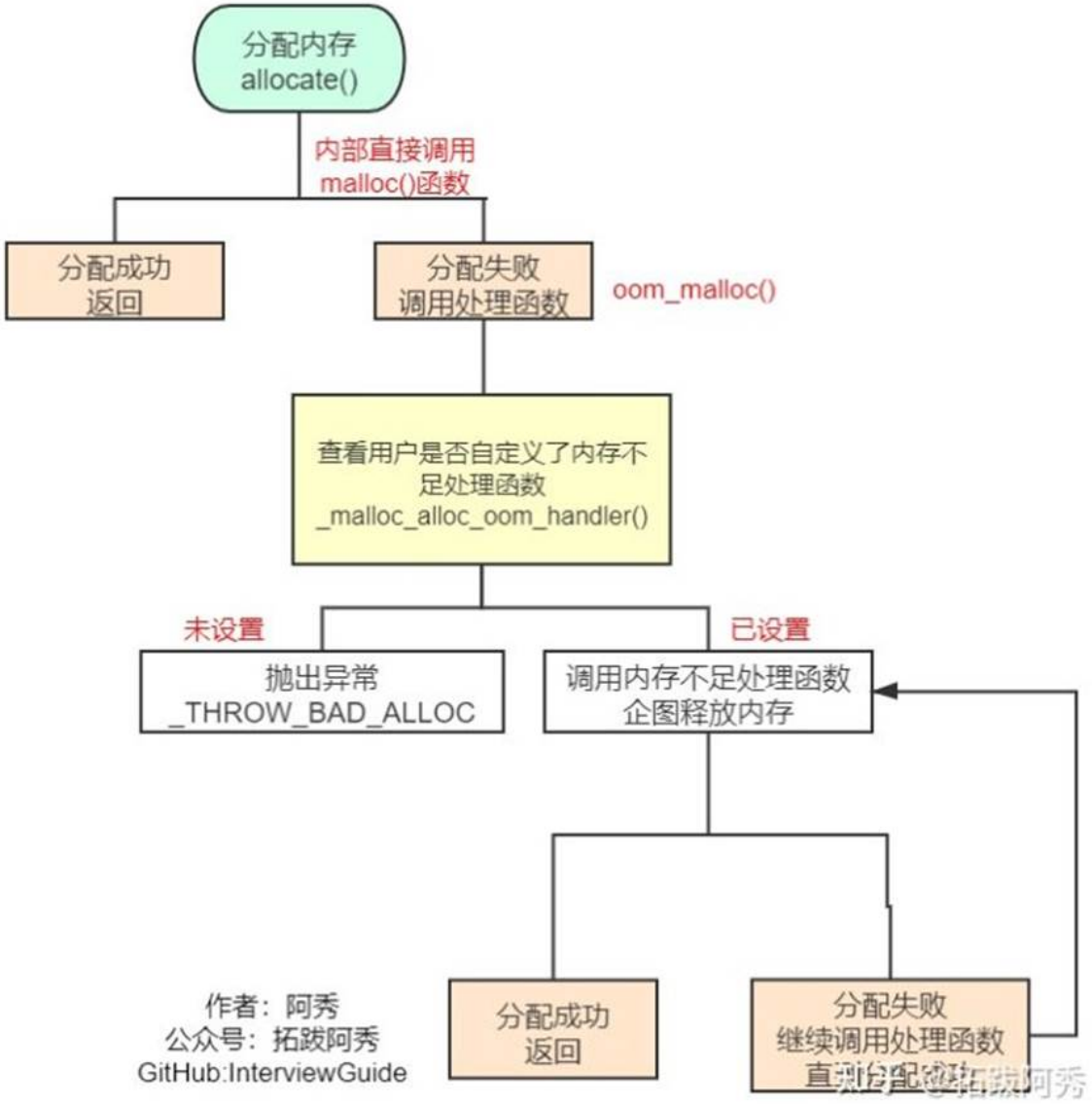
一级配置器

一级空间配置器中重要的函数就是allocate、deallocate、reallocate 。 一级空间配置器是以malloc()，free()，realloc()等C函数执行实际的内存配置 。大致过程是:

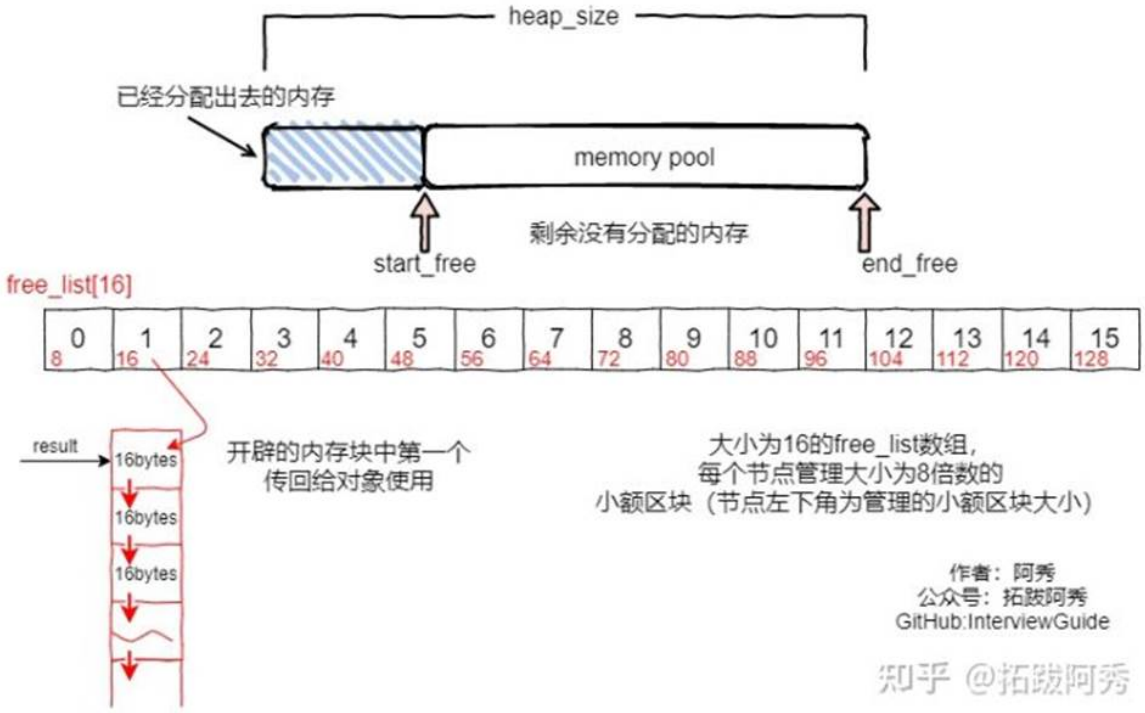
1、直接allocate分配内存，其实就是malloc来分配内存，成功则直接返回，失败就调用处理函数

2、如果用户自定义了内存分配失败的处理函数就调用，没有的话就返回异常

3、如果自定义了处理函数就进行处理，完事再继续分配试试


分配内存
allocate()
内部直接调用
malloc()函数
分配成功
返回
分配失败
调用处理函数
oom_malloc()
查看用户是否自定义了内存不
足处理函数
_malloc_alloc_oom_handler()
未设置
已设置
抛出异常
_THROW_BAD_ALLOC
调用内存不足处理函数
企图释放内存
作者：阿秀
公众号：拓跋阿秀
GitHub:InterviewGuide
分配成功
返回
分配失败
继续调用处理函数
直到分配成功
知乎 @拓跋阿秀

二级配置器


heap_size
已经分配出去的内存
memory pool
剩余没有分配的内存
start_free
end_free
free_list[16]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 128
result
16bytes
16bytes
16bytes
开辟的内存块中第一个
传回给对象使用
大小为16的free_list数组，
每个节点管理大小为8倍数的
小额区块（节点左下角为管理的小额区块大小）
作者：阿秀
公众号：拓跋阿秀
GitHub:InterviewGuide
知乎 @拓跋阿秀

1、维护16条链表，分别是0-15号链表，最小8字节，以8字节逐渐递增，最大128字节，你传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表（如需要13bytes空间，我们会给它分配16bytes大小），在找到第n个链表后查看链表是否为空，如果不为空直接从对应的free_list中拔出，将已经拨出的指针向后移动一位。

2、对应的free_list为空，先看其内存池是不是空时，如果内存池不为空： （1）先检验它剩余空间是否够20个节点大小（即所需内存大小(提升后) * 20），若足够则直接从内存池中拿出20个节点大小空间，将其中一个分配给用户使用，另外19个当作自由链表中的区块挂在相应的free_list下，这样下次再有相同大小的内存需求时，可直接拨出。 （2）如果不够20个节点大小，则看它是否能满足1个节点大小，如果够的话则直接拿出一个分配给用户，然后从剩余的空间中分配尽可能多的节点挂在相应的free_list中。 （3）如果连一个节点内存都不能满足的话，则将内存池中剩余的空间挂在相应的free_list中（找到相应的free_list），然后再给内存池申请内存，转到3。 3、内存池为空，申请内存 此时二级空间配置器会使用malloc()从heap上申请内存，（一次所申请的内存大小为2 * 所需节点内存大小（提升后）* 20 + 一段额外空间），申请40块，一半拿来用，一半放内存池中。 4、malloc没有成功 在第三种情况下，如果malloc()失败了，说明heap上没有足够空间分配给我们了，这时，二级空间配置器会从比所需节点空间大的free_list中一一搜索，从比它所需节点空间大的free_list中拔除一个节点来使用。如果这也没找到，说明比其大的free_list中都没有自由区块了，那就要调用一级适配器了。

释放时调用deallocate()函数，若释放的n>128，则调用一级空间配置器，否则就直接将内存块挂上自由链表的合适位置。

STL二级空间配置器虽然解决了外部碎片与提高了效率，但它同时增加了一些缺点：

1.因为自由链表的管理问题，它会把我们需求的内存块自动提升为8的倍数，这时若你需要1个字节，它会给你8个字节，即浪费了7个字节，所以它又引入了内部碎片的问题，若相似情况出现很多次，就会造成很多内部碎片；

2.二级空间配置器是在堆上申请大块的狭义内存池，然后用自由链表管理，供现在使用，在程序执行过程中，它将申请的内存一块一块都挂在自由链表上，即不会还给操作系统，并且它的实现中所有成员全是静态的，所以它申请的所有内存只有在进程结束才会释放内存，还给操作系统，由此带来的问题有：1.即我不断的开辟小块内存，最后整个堆上的空间都被挂在自由链表上，若我想开辟大块内存就会失败；2.若自由链表上挂很多内存块没有被使用，当前进程又占着内存不释放，这时别的进程在堆上申请不到空间，也不可以使用当前进程的空闲内存，由此就会引发多种问题。

一级分配器

GC4.9之后就没有第一级了，只有第二级

二级分配器

——default_alloc_template 剖析

有个自动调整的函数：你传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表（0-15号链表，最小8字节 最大128字节）

allocate函数：如果要分配的内存大于128字节，就转用第一级分配器，否则也就是小于128字节。那么首先判断落在第几号链表，定位到了，先判断链表是不是空，如果是空就需要充值，（调节到8的倍数，默认一次申请20个区块，当然了也要判断20个是不是能够申请到，如果只申请到一个那就直接返回好了，不止一个的话，把第2到第n个挨个挂到当前链表上，第一个返回回去给容器用,n是不大于20的，当然了如果不在1-20之间，那就是内存碎片了，那就先把碎片挂到某一条链表上，然后再重新malloc了，malloc 2*20个块）去内存池去拿或者重新分配。不为空的话

9、vector的增加删除都是怎么做的？为什么是1.5或者是2倍?

1) 新增元素：vector通过一个连续的数组存放元素，如果集合已满，在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的内存，在插入新增的元素；

2) 对vector的任何操作，一旦引起空间重新配置，指向原vector的所有迭代器就都失效了；

3) 初始时刻vector的capacity为0，塞入第一个元素后capacity增加为1；

4) 不同的编译器实现的扩容方式不一样，VS2015中以1.5倍扩容，GCC以2倍扩容。

对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到O(n)的时间复杂度，因此，使用成倍的方式扩容。

1) 考虑可能产生的堆空间浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式有两种，以2二倍的方式扩容，或者以1.5倍的方式扩容。

2) 以2倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和，导致之前分配的内存不能再被使用，所以最好倍增长因子设置为(1,2)之间：

3) 向量容器vector的成员函数pop_back()可以删除最后一个元素.

4) 而函数erase()可以删除由一个iterator指出的元素，也可以删除一个指定范围的元素。

5) 还可以采用通用算法remove()来删除vector容器中的元素.

6) 不同的是：采用remove一般情况下不会改变容器的大小，而pop_back()与erase()等成员函数会改变容器的大小。

10、最后再让你解释一下STL中stack和queue的实现

stack

stack（栈）是一种先进后出（First In Last Out）的数据结构，只有一个入口和出口，那就是栈顶，除了获取栈顶元素外，没有其他方法可以获取到内部的其他元素，其结构图如下：

stack这种单向开口的数据结构很容易由双向开口的deque和list形成，只需要根据stack的性质对应移除某些接口即可实现，stack的源码如下：

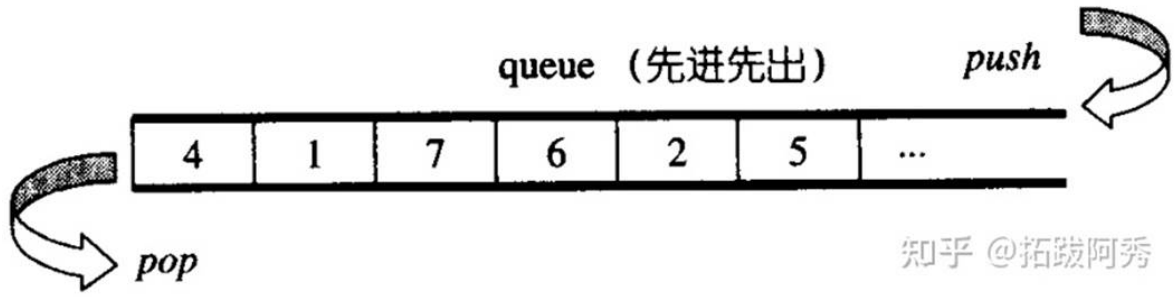
```
template  T, class Sequence = deque >
class stack
{
  …
protected:
  Sequence c;
public:
  bool empty(){return c.empty();}
  size_type size() const{return c.size();}
  reference top() const {return c.back();}
  const_reference top() const{return c.back();}
  void push(const value_type& x){c.push_back(x);}
  void pop(){c.pop_back();}
};
```

从stack的数据结构可以看出，其所有操作都是围绕Sequence完成，而Sequence默认是deque数据结构。stack这种“修改某种接口，形成另一种风貌”的行为，成为adapter(配接器)。常将其归类为container adapter而非container

stack除了默认使用deque作为其底层容器之外，也可以使用双向开口的list，只需要在初始化stack时，将list作为第二个参数即可。由于stack只能操作顶端的元素，因此其内部元素无法被访问，也不提供迭代器。

queue

queue（队列）是一种先进先出（First In First Out）的数据结构，只有一个入口和一个出口，分别位于最底端和最顶端，出口元素外，没有其他方法可以获取到内部的其他元素，其结构图如下：



类似的，queue这种“先进先出”的数据结构很容易由双向开口的deque和list形成，只需要根据queue的性质对应移除某些接口即可实现，queue的源码如下：

```
template  T, class Sequence = deque >
class queue
{
  ...
protected:
  Sequence c;
public:
  bool empty(){return c.empty();}
  size_type size() const{return c.size();}
  reference front() const {return c.front();}
  const_reference front() const{return c.front();}
  void push(const value_type& x){c.push_back(x);}
  void pop(){c.pop_front();}
};
```

从queue的数据结构可以看出，其所有操作都也都是是围绕Sequence完成，Sequence默认也是deque数据结构。queue也是一类container adapter。

同样，queue也可以使用list作为底层容器，不具有遍历功能，没有迭代器。