1、什么是 STL？

C++ STL 从广义来讲包括了三类：算法，容器和迭代器。

- 算法包括排序，复制等常用算法，以及不同容器特定的算法。
- 容器就是数据的存放形式，包括序列式容器和关联式容器，序列式容器就是 list，vector 等，关联式容器就是 set，map 等。
- 迭代器就是在不暴露容器内部结构的情况下对容器的遍历。

2、解释一下什么是 trivial destructor

“trivial destructor”一般是指用户没有自定义析构函数，而由系统生成的，这种析构函数在《STL 源码解析》中成为“无关痛痒”的析构函数。

反之，用户自定义了析构函数，则称之为“non-trivial destructor”，这种析构函数**如果申请了新的空间一定要显式的释放，否则会造成内存泄露**

对于 trivial destructor，如果每次都进行调用，显然对效率是一种伤害，如何进行判断呢？

《STL 源码解析》中给出的说明是:

首先利用 value_type()获取所指对象的型别，再利用__type_traits<T>判断该型别的析构函数是否 trivial，若是(__true_type)，则什么也不做，若为(__false_type)-则去调用 destory()函数。

也就是说，在实际的应用当中，STL 库提供了相关的判断方法**__type_traits**，感兴趣的读者可以自行查阅使用方式。除了 trivial destructor，还有 trivial construct、trivial copy construct 等，如果能够对是否 trivial 进行区分，可以采用内存处理函数 memcpy()、malloc()等更加高效的完成相关操作，提升效率。

3、使用智能指针管理内存资源，RAII 是怎么回事？

1. RAII 全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。

因为 C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。所以，在 RAII 的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定。

2. 智能指针（std::shared_ptr 和 std::unique_ptr）即 RAII 最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记 delete 造成的内存泄漏。

毫不夸张的来讲，有了智能指针，代码中几乎不需要再出现 delete 了。

4、迭代器：++it、it++哪个好，为什么

1. 前置返回一个引用，后置返回一个对象

```
// ++i 实现代码为
int& operator++()
{
  *this += 1;
  return *this;
}
```

1. 前置不会产生临时对象，后置必须产生临时对象，临时对象会导致效率降低

```
//i++实现代码为
int operator++(int)
{int temp = *this;
   ++*this;
   return temp;
}
```

5、说一下 C++左值引用和右值引用

C++11 正是通过引入右值引用来优化性能，具体来说是通过移动语义来避免无谓拷贝的问题，通过 move 语义来将临时生成的左值中的资源无代价的转移到另外一个对象中去，通过完美转发来解决不能按照参数实际类型来转发的问题（同时，完美转发获得的一个好处是可以实现移动语义）。

1.在 C++11 中所有的值必属于左值、右值两者之一，右值又可以细分为纯右值、将亡值。在 C++11 中可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值（将亡值或纯右值）。

2.举个例子，int a = b+c, a 就是左值，其有变量名为 a，通过&a 可以获取该变量的地址；表达式 b+c、函数 int func()的返回值是右值，在其被赋值给某一变量前，我们不能通过变量名找到它，＆(b+c)这样的操作则不会通过编译。

C++11 对 C++98 中的右值进行了扩充。在 C++11 中右值又分为纯右值（prvalue，Pure Rvalue）和将亡值（xvalue，eXpiring Value）。

其中纯右值的概念等同于我们在 C++98 标准中右值的概念，指的是临时变量和不跟对象关联的字面量值；将亡值则是 C++11 新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用 T&&的函数返回值、std::move 的返回值，或者转换为 T&&的类型转换函数的返回值。将亡值可以理解为通过“盗取”其他变量内存空间的方式获取到的值。在确保其他变量不再被使用、或即将被销毁时，通过“盗取”的方式可以避免内存空间的释放和分配，能够延长变量值的生命期。

3.左值引用就是对一个左值进行引用的类型。右值引用就是对一个右值进行引用的类型，事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。

右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名。

左值引用通常也不能绑定到右值，但常量左值引用是个“万能”的引用类型。它可以接受非常量左值、常量左值、右值对其进行初始化。不过常量左值所引用的右值在它的“余生”中只能是只读的。相对地，非常量左值只能接受非常量左值对其进行初始化。

4.右值值引用通常不能绑定到任何的左值，要想绑定一个左值到右值引用，通常需要 std::move()将左值强制转换为右值。

左值和右值

左值：表示的是可以获取地址的表达式，它能出现在赋值语句的左边，对该表达式进行赋值。但是修饰符 const 的出现使得可以声明如下的标识符，它可以取得地址，但是没办法对其进行赋值

```
const int& a = 10;
```

右值：表示无法获取地址的对象，有常量值、函数返回值、lambda 表达式等。无法获取地址，但不表示其不可改变，当定义了右值的右值引用时就可以更改右值。

左值引用和右值引用

左值引用：传统的 C++中引用被称为左值引用

右值引用：C++11 中增加了右值引用，右值引用关联到右值时，右值被存储到特定位置，右值引用指向该特定位置，也就是说，右值虽然无法获取地址，但是右值引用是可以获取地址的，该地址表示临时对象的存储位置

这里主要说一下右值引用的特点：

- 特点 1：通过右值引用的声明，右值又“重获新生”，其生命周期与右值引用类型变量的生命周期一样长，只要该变量还活着，该右值临时量将会一直存活下去
- 特点 2：右值引用独立于左值和右值。意思是右值引用类型的变量可能是左值也可能是右值

- 特点 3：T&& t 在发生自动类型推断的时候，它是左值还是右值取决于它的初始化。

举个例子：

```
#include <bits/stdc++.h>using namespace std;
template<typename T>void fun(T&& t){
        cout << t << endl;
}
int getInt(){
        return 5;
}
int main() {

        int a = 10;
        int& b = a;  //b 是左值引用
        int& c = 10;  //错误，c 是左值不能使用右值初始化
        int&& d = 10;  //正确，右值引用用右值初始化
        int&& e = a;  //错误，e 是右值引用不能使用左值初始化
        const int& f = a; //正确，左值常引用相当于是万能型，可以用左值或者右值初始化
        const int& g = 10;//正确，左值常引用相当于是万能型，可以用左值或者右值初始化
        const int&& h = 10; //正确，右值常引用
        const int& aa = h;//正确
        int& i = getInt();  //错误，i 是左值引用不能使用临时变量（右值）初始化
        int&& j = getInt();  //正确，函数返回值是右值
        fun(10); //此时 fun 函数的参数 t 是右值
        fun(a); //此时 fun 函数的参数 t 是左值
        return 0;
}
```

6、STL 中 hashtable 的实现？

STL 中的 hashtable 使用的是**开链法**解决 hash 冲突问题。

hashtable 中的 bucket 所维护的 list 既不是 list 也不是 slist，而是其自己定义的由 hashtable_node 数据结构组成的 linked-list，而 bucket 聚合体本身使用 vector 进行存储。hashtable 的迭代器只提供前进操作，不提供后退操作

在 hashtable 设计 bucket 的数量上，其内置了 28 个质数[53, 97, 193,...,429496729]，在创建 hashtable 时，会根据存入的元素个数选择大于等于元素个数的质数作为 hashtable 的容量（vector 的长度），其中每个 bucket 所维护的 linked-list 长度也等于 hashtable 的容量。

如果插入 hashtable 的元素个数超过了 bucket 的容量，就要进行重建 table 操作，即找出下一个质数，创建新的 buckets vector，重新计算元素在新 hashtable 的位置。

7、简单说一下 traits 技法

traits 技法利用"内嵌型别"的编程技巧与**编译器的 template 参数推导功能**，增强 C++未能提供的关于型别认证方面的能力。常用的有 iterator_traits 和 type_traits。

iterator_traits

被称为**特性萃取机**，能够方便的让外界获取以下 5 种型别：

- value_type：迭代器所指对象的型别
- difference_type：两个迭代器之间的距离
- pointer：迭代器所指向的型别
- reference：迭代器所引用的型别
- iterator_category：三两句说不清楚，建议看书

type_traits

关注的是型别的**特性**，例如这个型别是否具备 non-trivial defalt ctor（默认构造函数）、non-trivial copy ctor（拷贝构造函数）、non-trivial assignment operator（赋值运算符） 和 non-trivial dtor（析构函数），如果答案是否定的，可以采取直接操作内存的方式提高效率，一般来说，type_traits 支持以下 5 中类型的判断：

```
__type_traits<T>::has_trivial_default_constructor
__type_traits<T>::has_trivial_copy_constructor
__type_traits<T>::has_trivial_assignment_operator
__type_traits<T>::has_trivial_destructor
__type_traits<T>::is_POD_type
```

由于编译器只针对 class object 形式的参数进行参数推到，因此上式的返回结果不应该是个 bool 值，实际上使用的是一种空的结构体：

```
struct __true_type{};struct __false_type{};
```

这两个结构体没有任何成员，不会带来其他的负担，又能满足需求，可谓一举两得

当然，如果我们自行定义了一个 Shape 类型，也可以针对这个 Shape 设计 type_traits 的特化版本

```
template<> struct __type_traits<Shape>{
        typedef __true_type has_trivial_default_constructor;
        typedef __false_type has_trivial_copy_constructor;
        typedef __false_type has_trivial_assignment_operator;
        typedef __false_type has_trivial_destructor;
        typedef __false_type is_POD_type;
};
```

感谢微信好友“铁锤哥哥”勘误：“特性萃取机处” 方面->方便 ， 中->种 -2021.06.28

8、STL 的两级空间配置器

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

二级配置器

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；

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 与 list 的区别与应用？怎么找某 vector 或者 list 的倒数第二个元素

1. vector 数据结构 vector 和数组类似，拥有一段连续的内存空间，并且起始地址不变。因此能高效的进行随机存取，时间复杂度为 o(1);但因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度为 o(n)。

另外，当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝。连续存储结构：vector 是可以实现动态增长的对象数组，支持对数组高效率的访问和在数组尾端的删除和插入操作，在中间和头部删除和插入相对不易，需要挪动大量的数据。

它与数组最大的区别就是 vector 不需程序员自己去考虑容量问题，库里面本身已经实现了容量的动态增长，而数组需要程序员手动写入扩容函数进形扩容。

1. list 数据结构 list 是由双向链表实现的，因此内存空间是不连续的。只能通过指针访问数据，所以 list 的随机存取非常没有效率，时间复杂度为 o(n);但由于链表的特点，能高效地进行插入和删除。非连续存储结构：list 是一个双链表结构，支持对链表的双向遍历。每个节点包括三个信息：元素本身，指向前一个元素的节点（prev）和指向下一个元素的节点（next）。因此 list 可以高效率的对数据元素任意位置进行访问和插入删除等操作。由于涉及对额外指针的维护，所以开销比较大。

区别：

- vector 的随机访问效率高，但在插入和删除时（不包括尾部）需要挪动数据，不易操作。

- list 的访问要遍历整个链表，它的随机访问效率低。但对数据的插入和删除操作等都比较方便，改变指针的指向即可。
- 从遍历上来说，list 是单向的，vector 是双向的。
- vector 中的迭代器在使用后就失效了，而 list 的迭代器在使用之后还可以继续使用。

int mySize = vec.size();vec.at(mySize -2);

list 不提供随机访问，所以不能用下标直接访问到某个位置的元素，要访问 list 里的元素只能遍历，不过你要是只需要访问 list 的最后 N 个元素的话，可以用反向迭代器来遍历：

10、STL 中 vector 删除其中的元素，迭代器如何变化？为什么是两倍扩容？释放空间？

size()函数返回的是已用空间大小，capacity()返回的是总空间大小，capacity()-size()则是剩余的可用空间大小。当 size()和 capacity()相等，说明 vector 目前的空间已被用完，如果再添加新元素，则会引起 vector 空间的动态增长。

由于动态增长会引起重新分配内存空间、拷贝原空间、释放原空间，这些过程会降低程序效率。因此，可以使用 reserve(n)预先分配一块较大的指定大小的内存空间，这样当指定大小的内存空间未使用完时，是不会重新分配内存空间的，这样便提升了效率。只有当 n>capacity()时，调用 reserve(n)才会改变 vector 容量。

resize()成员函数改变元素的数目，至于空间的的变化需要看具体情况去分析，如下：

```
void resize(size_type __new_size, const _Tp& __x) {
      if (__new_size < size())
            erase(begin() + __new_size, end());
      else
            insert(end(), __new_size - size(), __x);
      }
```

1、空的 vector 对象，size()和 capacity()都为 0

2、当空间大小不足时，新分配的空间大小为原空间大小的 2 倍。

3、使用 reserve()预先分配一块内存后，在空间未满的情况下，不会引起重新分配，从而提升了效率。

4、当 reserve()分配的空间比原空间小时，是不会引起重新分配的。

5、resize()函数只改变容器的元素数目，未改变容器大小。

6、用 reserve(size_type)只是扩大 capacity 值，这些内存空间可能还是“野”的，如果此时使用“[ ]”来访问，则可能会越界。而 resize(size_type new_size)会真正使容器具有 new_size 个对象。

不同的编译器，vector 有不同的扩容大小。在 vs 下是 1.5 倍，在 GCC 下是 2 倍；

空间和时间的权衡。简单来说， 空间分配的多，平摊时间复杂度低，但浪费空间也多。

使用 k=2 增长因子的问题在于，每次扩展的新尺寸必然刚好大于之前分配的总和，也就是说，之前分配的内存空间不可能被使用。这样对内存不友好，最好把增长因子设为(1, 2)，也就是 1-2 之间的某个数值。

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。

11、Vector 如何释放空间?

由于 vector 的内存占用空间只增不减，比如你首先分配了 10,000 个字节，然后 erase 掉后面 9,999 个，留下一个有效元素，但是内存占用仍为 10,000 个。所有内存空间是在 vector 析构时候才能被系统回收。

empty()用来检测容器是否为空的，clear()可以清空所有元素。但是即使 clear()，vector 所占用的内存空间依然如故，无法保证内存的回收。

如果需要空间动态缩小，可以考虑使用 deque。

如果使用 vector，可以用 swap()来帮助你释放多余内存或者清空全部内存。

```
vector(Vec).swap(Vec); //将 Vec 中多余内存清除；
vector().swap(Vec); //清空 Vec 的全部内存；
```

实例

```
#include <iostream>
#include <vector>
using namespace std;
int main (){
    vector<int> vec (100,100);
    // three ints with a value of 100
    vec.push_back(1);
    vec.push_back(2);
    cout <<"vec.size(): " << vec.size() << endl;
    cout <<"vec.capasity(): " << vec.capacity() << endl;

    vector<int>(vec).swap(vec);
    //清空vec中多余的空间，相当于vec.shrink_to_fit();

    cout <<"vec.size(): " << vec.size() << endl;
    cout <<"vec.capasity(): " << vec.capacity() << endl;

    vector<int>().swap(vec); //清空vec的全部空间

    cout <<"vec.size(): " << vec.size() << endl;
    cout <<"vec.capasity(): " << vec.capacity() << endl;

    return 0;
}
/*
    运行结果:
    vec.size(): 102
    vec.capasity(): 200
    vec.size(): 102
    vec.capasity(): 102
    vec.size(): 0
    vec.capasity(): 0
 */
```

12、容器内部删除一个元素

1. 顺序容器（序列式容器，比如 vector、deque）

erase 迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list 除外)，所以不能使用 erase(it++)的方式，但是 erase 的返回值是下一个有效迭代器；

It = c.erase(it);

2. 关联容器(关联式容器，比如 map、set、multimap、multiset 等)

erase 迭代器只是被删除元素的迭代器失效，但是返回值是 void，所以要采用 erase(it++)的方式删除迭代器；

c.erase(it++)

13、STL 迭代器如何实现

1、 迭代器是一种抽象的设计理念，通过迭代器可以在不了解容器内部原理的情况下遍历容器，除此之外，STL 中迭代器一个最重要的作用就是作为容器与 STL 算法的粘合剂。

2、 迭代器的作用就是提供一个遍历容器内部所有元素的接口，因此迭代器内部必须保存一个与容器相关联的指针，然后重载各种运算操作来遍历，其中最重要的是*运算符与->运算符，以及++、--等可能需要重载的运算符重载。

这和 C++中的智能指针很像，智能指针也是将一个指针封装，然后通过引用计数或是其他方法完成自动释放内存的功能。

3、最常用的迭代器的相应型别有五种：value type、difference type、pointer、reference、iterator catagoly;

14、map、set 是怎么实现的，红黑树是怎么能够同时实现这两种容器？ 为什么使用红黑树？

1. 他们的底层都是以红黑树的结构实现，因此插入删除等操作都在 O(logn 时间内完成，因此可以完成高效的插入删除；
2. 在这里我们定义了一个模版参数，如果它是 key 那么它就是 set，如果它是 map，那么它就是 map；底层是红黑树，实现 map 的红黑树的节点数据类型是 key+value，而实现 set 的节点数据类型是 value
3. 因为 map 和 set 要求是自动排序的，红黑树能够实现这一功能，而且时间复杂度比较低。

15、如何在共享内存上使用 STL 标准库？

1. 想像一下把 STL 容器，例如 map, vector, list 等等，放入共享内存中，IPC 一旦有了这些强大的通用数据结构做辅助，无疑进程间通信的能力一下子强大了很多。

我们没必要再为共享内存设计其他额外的数据结构，另外，STL 的高度可扩展性将为 IPC 所驱使。STL 容器被良好的封装，默认情况下有它们自己的内存管理方案。

当一个元素被插入到一个 STL 列表(list)中时，列表容器自动为其分配内存，保存数据。考虑到要将 STL 容器放到共享内存中，而容器却自己在堆上分配内存。

一个最笨拙的办法是在堆上构造 STL 容器，然后把容器复制到共享内存，并且确保所有容器的内部分配的内存指向共享内存中的相应区域，这基本是个不可能完成的任务。

2. 假设进程 A 在共享内存中放入了数个容器，进程 B 如何找到这些容器呢？

一个方法就是进程 A 把容器放在共享内存中的确定地址上（fixed offsets），则进程 B 可以从该已知地址上获取容器。另外一个改进点的办法是，进程 A 先在共享内存某块确定地址上放置一个 map 容器，然后进程 A 再创建其他容器，然后给其取个名字和地址一并保存到这个 map 容器里。

进程 B 知道如何获取该保存了地址映射的 map 容器,然后同样再根据名字取得其他容器地址。

16、map 插入方式有哪几种？

1. **用 insert 函数插入 pair 数据，**

```
mapStudent.insert(pair<int, string>(1, "student_one"));
```

2. **用 insert 函数插入 value_type 数据**

```
mapStudent.insert(map<int, string>::value_type (1, "student_one"));
```

3. **在 insert 函数中使用 make_pair()函数**

```
mapStudent.insert(make_pair(1, "student_one"));
```

4. **用数组方式插入数据**

```
mapStudent[1] = "student_one";
```

17、STL 中 unordered_map(hash_map)和 map 的区别，hash_map 如何解决冲突以及扩容

1. unordered_map 和 map 类似，都是存储的 key-value 的值，可以通过 key 快速索引到 value。不同的是 unordered_map 不会根据 key 的大小进行排序，
2. 存储时是根据 key 的 hash 值判断元素是否相同，即 unordered_map 内部元素是无序的，而 map 中的元素是按照二叉搜索树存储，进行中序遍历会得到有序遍历。
3. 所以使用时 map 的 key 需要定义 operator<。而 unordered_map 需要定义 hash_value 函数并且重载 operator==。但是很多系统内置的数据类型都自带这些，
4. 那么如果是自定义类型，那么就需要自己重载 operator<或者 hash_value()了。
5. 如果需要内部元素自动排序，使用 map，不需要排序使用 unordered_map
6. unordered_map 的底层实现是 hash_table;
7. hash_map 底层使用的是 hash_table，而 hash_table 使用的开链法进行冲突避免，所有 hash_map 采用开链法进行冲突解决。
8. **什么时候扩容：**当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即当前数组的长度乘以加载因子的值的时候，就要自动扩容啦。
9. **扩容(resize)**就是重新计算容量，向 HashMap 对象里不停的添加元素，而 HashMap 对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。

18、vector 越界访问下标，map 越界访问下标？vector 删除元素时会不会释放空间？

1. 通过下标访问 vector 中的元素时会做边界检查，但该处的实现方式要看具体 IDE，不同 IDE 的实现方式不一样，确保不可访问越界地址。
2. map 的下标运算符[]的作用是：将 key 作为下标去执行查找，并返回相应的值；如果不存在这个 key，就将一个具有该 key 和 value 的某人值插入这个 map。
3. erase()函数，只能删除内容，不能改变容量大小;

erase 成员函数，它删除了 itVect 迭代器指向的元素，并且返回要被删除的 itVect 之后的迭代器，迭代器相当于一个智能指针;clear()函数，只能清空内容，不能改变容量大小;如果要想在删除内容的同时释放内存，那么你可以选择 deque 容器。

19、map 中[]与 find 的区别？

1. map 的下标运算符[]的作用是：将关键码作为下标去执行查找，并返回对应的值；如果不存在这个关键码，就将一个具有该关键码和值类型的默认值的项插入这个 map。
2. map 的 find 函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器。

20、 STL 中 list 与 queue 之间的区别

1. list 不再能够像 vector 一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在；
2. list 不仅是一个双向链表，而且还是一个环状双向链表，所以它只需要一个指针；
3. list 不像 vector 那样有可能在空间不足时做重新配置、数据移动的操作，所以插入前的所有迭代器在插入操作之后都仍然有效；
4. deque 是一种双向开口的连续线性空间，所谓双向开口，意思是可以在头尾两端分别做元素的插入和删除操作；
5. deque 和 vector 最大的差异，一在于 deque 允许常数时间内对起头端进行元素的插入或移除操作；二在于 deque 没有所谓容量概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，deque 没有所谓的空间保留功能。

21、STL 中的 allocator、deallocator

1. 第一级配置器直接使用 malloc()、free()和 relloc()，第二级配置器视情况采用不同的策略：当配置区块超过 128bytes 时，视之为足够大，便调用第一级配置器；当配置器区块小于 128bytes 时，为了降低额外负担，使用复杂的内存池整理方式，而不再用一级配置器；
2. 第二级配置器主动将任何小额区块的内存需求量上调至 8 的倍数，并维护 16 个 free-list，各自管理大小为 8~128bytes 的小额区块；
3. 空间配置函数 allocate()，首先判断区块大小，大于 128 就直接调用第一级配置器，小于 128 时就检查对应的 free-list。如果 free-list 之内有可用区块，就直接拿来用，如果没有可用区块，就将区块大小调整至 8 的倍数，然后调用 refill()，为 free-list 重新分配空间；
4. 空间释放函数 deallocate()，该函数首先判断区块大小，大于 128bytes 时，直接调用一级配置器，小于 128bytes 就找到对应的 free-list 然后释放内存。

22、STL 中 hash table 扩容发生什么？

1. hash table 表格内的元素称为桶（bucket),而由桶所链接的元素称为节点（node),其中存入桶元素的容器为 stl 本身很重要的一种序列式容器——vector 容器。之所以选择 vector 为存放桶元素的基础容器，主要是因为 vector 容器本身具有动态扩容能力，无需人工干预。
2. 向前操作：首先尝试从目前所指的节点出发，前进一个位置（节点），由于节点被安置于 list 内，所以利用节点的 next 指针即可轻易完成前进操作，如果目前正巧是 list 的尾端，就跳至下一个 bucket 身上，那正是指向下一个 list 的头部节点。

23、常见容器性质总结？

1. vector 底层数据结构为数组 ，支持快速随机访问
2. list 底层数据结构为双向链表，支持快速增删
3. deque 底层数据结构为一个中央控制器和多个缓冲区，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问

deque 是一个双端队列(double-ended queue)，也是在堆中保存内容的.它的保存形式如下:

[堆 1] --> [堆 2] -->[堆 3] --> ...

每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是 list 和 vector 的结合品.

4. stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时
5. queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时（stack 和 queue 其实是适配器,而不叫容器，因为是对容器的再封装）
6. priority_queue 的底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现
7. set 底层数据结构为红黑树，有序，不重复
8. multiset 底层数据结构为红黑树，有序，可重复
9. map 底层数据结构为红黑树，有序，不重复
10. multimap 底层数据结构为红黑树，有序，可重复
11. unordered_set 底层数据结构为 hash 表，无序，不重复
12. unordered_multiset 底层数据结构为 hash 表，无序，可重复
13. unordered_map 底层数据结构为 hash 表，无序，不重复
14. unordered_multimap 底层数据结构为 hash 表，无序，可重复

24、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。

25、说一下 STL 每种容器对应的迭代器

容器	迭代器
vector、deque	随机访问迭代器
stack、queue、priority_queue	无
list、(multi)set/map	双向迭代器
unordered_(multi)set/map、forward_list	前向迭代器

26、STL 中迭代器失效的情况有哪些？

以 vector 为例：

插入元素：

1、尾后插入：size < capacity 时，首迭代器不失效尾迭代失效（未重新分配空间），size == capacity 时，所有迭代器均失效（需要重新分配空间）。

2、中间插入：中间插入：size < capacity 时，首迭代器不失效但插入元素之后所有迭代器失效，size == capacity 时，所有迭代器均失效。

删除元素：

尾后删除：只有尾迭代失效。

中间删除：删除位置之后所有迭代失效。

deque 和 vector 的情况类似,

而 list 双向链表每一个节点内存不连续, 删除节点仅当前迭代器失效,erase 返回下一个有效迭代器;

map/set 等关联容器底层是红黑树删除节点不会影响其他节点的迭代器, 使用递增方法获取下一个迭代器 mmp.erase(iter++);

unordered_(hash) 迭代器意义不大, rehash 之后, 迭代器应该也是全部失效.

27、STL 中 vector 的实现

vector 是一种序列式容器，其数据安排以及操作方式与 array 非常类似，两者的唯一差别就是对于空间运用的灵活性。

众所周知，array 占用的是静态空间，一旦配置了就不可以改变大小，如果遇到空间不足的情况还要自行创建更大的空间，并手动将数据拷贝到新的空间中，再把原来的空间释放。

vector 则使用灵活的动态空间配置，维护一块**连续的线性空间**，在空间不足时，可以自动扩展空间容纳新元素，做到按需供给。

其在扩充空间的过程中仍然需要经历：**重新配置空间，移动数据，释放原空间**等操作。这里需要说明一下动态扩容的规则：以原大小的两倍配置另外一块较大的空间（或者旧长度+新增元素的个数），源码：

```
const size_type len  = old_size + max(old_size, n);
```

Vector 扩容倍数与平台有关，在 Win + VS 下是 1.5 倍，在 Linux + GCC 下是 2 倍

测试代码：

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    //在Linux + GCC 下
        vector<int> res(2,0);
        cout << res.capacity() <<endl; //2
        res.push_back(1);
        cout << res.capacity() <<endl;//4
        res.push_back(2);
        res.push_back(3);
    cout << res.capacity() <<endl;//8
        return 0;


    //在 win 10 + VS2019 下
    vector<int> res(2,0);
        cout << res.capacity() <<endl; //2
        res.push_back(1);
        cout << res.capacity() <<endl;//3
        res.push_back(2);
        res.push_back(3);
    cout << res.capacity() <<endl;//6

}
```

运行上述代码，一开始配置了一块长度为 2 的空间，接下来插入一个数据，长度变为原来的两倍，为 4，此时已占用的长度为 3，再继续两个数据，此时长度变为 8，可以清晰的看到空间的变化过程

需要注意的是，频繁对 vector 调用 push_back()对性能是有影响的，这是因为每插入一个元素，如果空间够用的话还能直接插入，若空间不够用，则需要重新配置空间，移动数据，释放原空间等操作，对程序性能会造成一定的影响

28、STL 中 slist 的实现

list 是双向链表，而 slist（single linked list）是单向链表，它们的主要区别在于：前者的迭代器是双向的 Bidirectional iterator，后者的迭代器属于单向的 Forward iterator。虽然 slist 的很多功能不如 list 灵活，但是其所耗用的空间更小，操作更快。

根据 STL 的习惯，插入操作会将新元素插入到指定位置之前，而非之后，然而 slist 是不能回头的，只能往后走，因此在 slist 的其他位置插入或者移除元素是十分不明智的，但是在 slist 开头却是可取的，slist 特别提供了 insert_after()和 erase_after 供灵活应用。考虑到效率问题，slist 只提供 push_front()操作，元素插入到 slist 后，存储的次序和输入的次序是相反的

slist 的单向迭代器如下图所示：

slist 默认采用 alloc 空间配置器配置节点的空间，其数据结构主要代码如下

```
template <class T, class Allco = alloc>class slist
{
        ...private:
    ...
    static list_node* create_node(const value_type& x){}//配置空间、构造元素
    static void destroy_node(list_node* node){}//析构函数、释放空间 private:
    list_node_base head; //头部 public:
    iterator begin(){}
    iterator end(){}
    size_type size(){}
    bool empty(){}
    void swap(slist& L){}//交换两个 slist，只需要换 head 即可
    reference front(){} //取头部元素
    void push_front(const value& x){}//头部插入元素
    void pop_front(){}//从头部取走元素
    ...
}
```

举个例子：

```
#include <forward_list>#include <algorithm>#include <iostream>using
namespace std;
int main(){
        forward_list<int> fl;
        fl.push_front(1);
        fl.push_front(3);
        fl.push_front(2);
        fl.push_front(6);
        fl.push_front(5);

        forward_list<int>::iterator ite1 = fl.begin();
        forward_list<int>::iterator ite2 = fl.end();
        for(;ite1 != ite2; ++ite1)
        {
                cout << *ite1 <<" "; // 5 6 2 3 1
        }
        cout << endl;

        ite1 = find(fl.begin(), fl.end(), 2); //寻找2 的位置

        if (ite1 != ite2)
                fl.insert_after(ite1, 99);
        for (auto it : fl)
        {
                cout << it << " ";  //5 6 2 99 3 1
        }
        cout << endl;

        ite1 = find(fl.begin(), fl.end(), 6); //寻找6 的位置
        if (ite1 != ite2)
                fl.erase_after(ite1);
        for (auto it : fl)
        {
                cout << it << " ";  //5 6 99 3 1
        }
        cout << endl;
        return 0;
}
```

需要注意的是 C++标准委员会没有采用 slist 的名称，forward_list 在 C++ 11 中出现，它与 slist 的区别是没有 size()方法。

29、STL 中 list 的实现

相比于 vector 的连续线型空间，list 显得复杂许多，但是它的好处在于插入或删除都只作用于一个元素空间，因此 list 对空间的运用是十分精准的，对任何位置元素的插入和删除都是常数时间。list 不能保证节点在存储空间中连续存储，也拥有迭代器，迭代器的“++”、“--”操作对于的是指针的操作，list 提供的迭代器类型是双向迭代器：Bidirectional iterators。

list 节点的结构见如下源码：

```
template <class T>struct __list_node{
    typedef void* void_pointer;
    void_pointer prev;
    void_pointer next;
    T data;
}
```

从源码可看出 list 显然是一个双向链表。list 与 vector 的另一个区别是，在插入和接合操作之后，都不会造成原迭代器失效，而 vector 可能因为空间重新配置导致迭代器失效。

此外 list 也是一个环形链表，因此只要一个指针便能完整表现整个链表。list 中 node 节点指针始终指向尾端的一个空白节点，因此是一种“前闭后开”的区间结构

list 的空间管理默认采用 alloc 作为空间配置器，为了方便的以节点大小为配置单位，还定义一个 list_node_allocator 函数可一次性配置多个节点空间

由于 list 的双向特性，其支持在头部（front)和尾部（back)两个方向进行 push 和 pop 操作，当然还支持 erase，splice，sort，merge，reverse，sort 等操作，这里不再详细阐述。

30、STL 中的 deque 的实现

vector 是单向开口（尾部）的连续线性空间，deque 则是一种双向开口的连续线性空间，虽然 vector 也可以在头尾进行元素操作，但是其头部操作的效率十分低下（主要是涉及到整体的移动）

deque 和 vector 的最大差异一个是 deque 运行在常数时间内对头端进行元素操作，二是 deque 没有容量的概念，它是动态地以分段连续空间组合而成，可以随时增加一段新的空间并链接起来

deque 虽然也提供随机访问的迭代器，但是其迭代器并不是普通的指针，其复杂程度比 vector 高很多，因此除非必要，否则一般使用 vector 而非 deque。如果需要对 deque 排序，可以先将 deque 中的元素复制到 vector 中，利用 sort 对 vector 排序，再将结果复制回 deque

deque 由一段一段的定量连续空间组成，一旦需要增加新的空间，只要配置一段定量连续空间拼接在头部或尾部即可，因此 deque 的最大任务是如何维护这个整体的连续性

deque 的数据结构如下：

```
class deque
{
    ...protected:
    typedef pointer* map_pointer;//指向 map 指针的指针
    map_pointer map;//指向 map
    size_type map_size;//map 的大小 public:
    ...
    iterator begin();
    iterator end();
    ...
}
```

deque 内部有一个指针指向 map，map 是一小块连续空间，其中的每个元素称为一个节点，node，每个 node 都是一个指针，指向另一段较大的连续空间，称为缓冲区，这里就是 deque 中实际存放数据的区域，默认大小 512bytes。整体结构如上图所示。

deque 的迭代器数据结构如下：

```
struct __deque_iterator
{
    ...
    T* cur;//迭代器所指缓冲区当前的元素
    T* first;//迭代器所指缓冲区第一个元素
    T* last;//迭代器所指缓冲区最后一个元素
    map_pointer node;//指向 map 中的 node
    ...
}
```

从 deque 的迭代器数据结构可以看出，为了保持与容器联结，迭代器主要包含上述 4 个元素

deque 迭代器的“++”、“--”操作是远比 vector 迭代器繁琐，其主要工作在于缓冲区边界，如何从当前缓冲区跳到另一个缓冲区，当然 deque 内部在插入元素时，如果 map 中 node 数量全部使用完，且 node 指向的缓冲区也没有多余的空间，这时会配置新的 map（2 倍于当前+2 的数量）来容纳更多的 node，也就是可以指向更多的缓冲区。在 deque 删除元素时，也提供了元素的析构和空闲缓冲区空间的释放等机制。

31、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 <class T, class Sequence = deque<T> >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 <class T, class Sequence = deque<T> >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。

32、STL 中的 heap 的实现

heap（堆）并不是 STL 的容器组件，是 priority queue（优先队列）的底层实现机制，因为 binary max heap（大根堆）总是最大值位于堆的根部，优先级最高。

binary heap 本质是一种 complete binary tree（完全二叉树），整棵 binary tree 除了最底层的叶节点之外，都是填满的，但是叶节点从左到右不会出现空隙，如下图所示就是一颗完全二叉树

完全二叉树内没有任何节点漏洞，是非常紧凑的，这样的一个好处是可以使用 array 来存储所有的节点，因为当其中某个节点位于 i 处，其左节点必定位于 $2i$ 处，右节点位于 $2i+1$ 处，父节点位于 $i/2$（向下取整）处。这种以 array 表示 tree 的方式称为隐式表述法。

因此我们可以使用一个 array 和一组 heap 算法来实现 max heap（每个节点的值大于等于其子节点的值）和 min heap（每个节点的值小于等于其子节点的值）。由于 array 不能动态的改变空间大小，用 vector 代替 array 是一个不错的选择。

那 heap 算法有哪些？常见有的插入、弹出、排序和构造算法，下面一一进行描述。

push_heap 插入算法

由于完全二叉树的性质，新插入的元素一定是位于树的最底层作为叶子节点，并填补由左至右的第一个空格。

事实上，在刚执行插入操作时，新元素位于底层 vector 的 end()处，之后是一个称为 percolate up（上溯）的过程，举个例子如下图：

新元素 50 在插入堆中后，先放在 vector 的 end()存着，之后执行上溯过程，调整其根结点的位置，以便满足 max heap 的性质，如果了解大根堆的话，这个原理跟大根堆的调整过程是一样的。

pop_heap 算法

heap 的 pop 操作实际弹出的是根节点吗，但在 heap 内部执行 pop_heap 时，只是将其移动到 vector 的最后位置，然后再为这个被挤走的元素找到一个合适的安放位置，使整颗树满足完全二叉树的条件。

这个被挤掉的元素首先会与根结点的两个子节点比较，并与较大的子节点更换位置，如此一直往下，直到这个被挤掉的元素大于左右两个子节点，或者下放到叶节点为止，这个过程称为 percolate down（下溯）。

举个例子：

根节点 68 被 pop 之后，移到了 vector 的最底部，将 24 挤出，24 被迫从根节点开始与其子节点进行比较，直到找到合适的位置安身，需要注意的是 pop 之后元素并没有被移走，如果要将其移走，可以使用 pop_back()。

sort 算法

一言以蔽之，因为 pop_heap 可以将当前 heap 中的最大值置于底层容器 vector 的末尾，heap 范围减 1，那么不断的执行 pop_heap 直到树为空，即可得到一个递增序列。

make_heap 算法

将一段数据转化为 heap，一个一个数据插入，调用上面说的两种 percolate 算法即可。

代码实测：

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main(){
        vector<int> v = { 0,1,2,3,4,5,6 };
        make_heap(v.begin(), v.end()); //以 vector 为底层容器
        for (auto i : v)
        {
                cout << i << " "; // 6 4 5 3 1 0 2
        }
        cout << endl;
        v.push_back(7);
        push_heap(v.begin(), v.end());
        for (auto i : v)
        {
                cout << i << " "; // 7 6 5 4 1 0 2 3
        }
        cout << endl;
        pop_heap(v.begin(), v.end());
        cout << v.back() << endl; // 7
        v.pop_back();
        for (auto i : v)
        {
                cout << i << " "; // 6 4 5 3 1 0 2
        }
        cout << endl;
        sort_heap(v.begin(), v.end());
        for (auto i : v)
        {
                cout << i << " "; // 0 1 2 3 4 5 6
        }
        return 0;
}
```

33、STL 中的 priority_queue 的实现

priority_queue，优先队列，是一个拥有权值观念的 queue，它跟 queue 一样是顶部入口，底部出口，在插入元素时，元素并非按照插入次序排列，它会自动根据权值（通常是元素的实值）排列，权值最高，排在最前面，如下图所示。

默认情况下，priority_queue 使用一个 max-heap 完成，底层容器使用的是一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现 。priority_queue 的这种实现机制导致其不被归为容器，而是一种容器配接器。关键的源码如下：

```
template <class T, class Squence = vector<T>, class Compare = less<typename
Sequence::value_tyoe> >class priority_queue{
        ...protected:
    Sequence c; // 底层容器
    Compare comp; // 元素大小比较标准 public:
    bool empty() const {return c.empty();}
    size_type size() const {return c.size();}
    const_reference top() const {return c.front()}
    void push(const value_type& x)
    {
        c.push_heap(x);
        push_heap(c.begin(), c.end(),comp);
    }
    void pop()
    {
        pop_heap(c.begin(), c.end(),comp);
        c.pop_back();
    }
};
```

priority_queue 的所有元素，进出都有一定的规则，只有 queue 顶端的元素（权值最高者），才有机会被外界取用，它没有遍历功能，也不提供迭代器

举个例子：

```
#include <queue>
#include <iostream>
using namespace std;
int main(){
        int ia[9] = {0,4,1,2,3,6,5,8,7 };
        priority_queue<int> pq(ia, ia + 9);
        cout << pq.size() <<endl;  // 9
        for(int i = 0; i < pq.size(); i++)
        {
                cout << pq.top() << " "; // 8 8 8 8 8 8 8 8 8
        }
        cout << endl;
        while (!pq.empty())
        {
                cout << pq.top() << ' ';// 8 7 6 5 4 3 2 1 0
                pq.pop();
        }
        return 0;
}
```

34、STL 中 set 的实现？

STL 中的容器可分为序列式容器（sequence）和关联式容器（associative），set 属于关联式容器。

set 的特性是，所有元素都会根据元素的值自动被排序（默认升序），set 元素的键值就是实值，实值就是键值，set 不允许有两个相同的键值

set 不允许迭代器修改元素的值，其迭代器是一种 constance iterators

标准的 STL set 以 RB-tree（红黑树）作为底层机制，几乎所有的 set 操作行为都是转调用 RB-tree 的操作行为，这里补充一下红黑树的特性：

- 每个节点不是红色就是黑色
- 根结点为黑色
- 如果节点为红色，其子节点必为黑
- 任一节点至（NULL）树尾端的任何路径，所含的黑节点数量必相同

关于红黑树的具体操作过程，比较复杂读者可以翻阅《算法导论》详细了解。

举个例子：

```
#include <set>#include <iostream>using namespace std;

int main(){
        int i;
        int ia[5] = { 1,2,3,4,5 };
        set<int> s(ia, ia + 5);
        cout << s.size() << endl; // 5
        cout << s.count(3) << endl; // 1
        cout << s.count(10) << endl; // 0

        s.insert(3); //再插入一个3
        cout << s.size() << endl; // 5
        cout << s.count(3) << endl; // 1

        s.erase(1);
        cout << s.size() << endl; // 4

        set<int>::iterator b = s.begin();
        set<int>::iterator e = s.end();
        for (; b != e; ++b)
                cout << *b << " "; // 2 3 4 5
        cout << endl;

        b = find(s.begin(), s.end(), 5);
        if (b != s.end())
                cout << "5 found" << endl; // 5 found

        b = s.find(2);
        if (b != s.end())
                cout << "2 found" << endl; // 2 found

        b = s.find(1);
        if (b == s.end())
                cout << "1 not found" << endl; // 1 not found
        return 0;
}
```

关联式容器尽量使用其自身提供的 find()函数查找指定的元素，效率更高，因为 STL 提供的 find()函数是一种顺序搜索算法。

35、STL 中 map 的实现

map 的特性是所有元素会根据键值进行自动排序。map 中所有的元素都是 pair，拥有键值(key)和实值(value)两个部分，并且不允许元素有相同的 key

一旦 map 的 key 确定了，那么是无法修改的，但是可以修改这个 key 对应的 value，因此 map 的迭代器既不是 constant iterator，也不是 mutable iterator

标准 STL map 的底层机制是 RB-tree（红黑树），另一种以 hash table 为底层机制实现的称为 hash_map。map 的架构如下图所示

map 的在构造时缺省采用递增排序 key，也使用 alloc 配置器配置空间大小，需要注意的是在插入元素时，调用的是红黑树中的 insert_unique()方法，而非 insert_euqal()（multimap 使用）

举个例子：

```
#include <map>
#include <iostream>
#include <string>
using namespace std;
int main(){
        map<string, int> maps;
    //插入若干元素
        maps["jack"] = 1;
        maps["jane"] = 2;
        maps["july"] = 3;
        //以 pair 形式插入
        pair<string, int> p("david", 4);
        maps.insert(p);
        //迭代输出元素
        map<string, int>::iterator iter = maps.begin();
        for (; iter != maps.end(); ++iter)
        {
                cout << iter->first << " ";
                cout << iter->second << "--";
//david 4--jack 1--jane 2--july 3--
        }
        cout << endl;
        //使用 subscipt 操作取实值
        int num = maps["july"];
        cout << num << endl; // 3
```

```
        //查找某key
        iter = maps.find("jane");
        if(iter != maps.end())
                cout << iter->second << endl; // 2
    //修改实值
        iter->second = 100;
        int num2 = maps["jane"]; // 100
        cout << num2 << endl;

        return 0;
}
```

需要注意的是 subscript（下标）操作既可以作为左值运用（修改内容）也可以作为右值运用（获取实值）。例如：

```
maps["abc"] = 1; //左值运用int num = masp["abd"]; //右值运用
```

无论如何，subscript 操作符都会先根据键值找出实值，源码如下：

```
...T& operator[](const key_type& k){
    return (*((insert(value_type(k, T()))).first)).second;
}...
```

代码运行过程是：首先根据键值和实值做出一个元素，这个元素的实值未知，因此产生一个与实值型别相同的临时对象替代：

```
value_type(k, T());
```

再将这个对象插入到 map 中，并返回一个 pair：

```
pair<iterator,bool> insert(value_type(k, T()));
```

pair 第一个元素是迭代器，指向当前插入的新元素，如果插入成功返回 true，此时对应左值运用，根据键值插入实值。插入失败（重复插入）返回 false，此时返回的是已经存在的元素，则可以取到它的实值

```
(insert(value_type(k, T()))).first; //迭代器
*((insert(value_type(k, T()))).first); //解引用
(*((insert(value_type(k, T()))).first)).second; //取出实值
```

由于这个实值是以引用方式传递，因此作为左值或者右值都可以

36、set 和 map 的区别，multimap 和 multiset 的区别

set 只提供一种数据类型的接口，但是会将这一个元素分配到 key 和 value 上，而且它的 compare_function 用的是 identity()函数。这个函数是输入什么输出什么，这样就实现了 set 机制，set 的 key 和 value 其实是一样的了。其实他保存的是两份元素，而不是只保存一份元素

map 则提供两种数据类型的接口，分别放在 key 和 value 的位置上，他的比较 function 采用的是红黑树的 comparefunction（），保存的确实是两份元素。

他们两个的 insert 都是采用红黑树的 insert_unique() 独一无二的插入 。

multimap 和 map 的唯一区别就是：multimap 调用的是红黑树的 insert_equal(),可以重复插入而 map 调用的则是独一无二的插入 insert_unique()，multiset 和 set 也一样，底层实现都是一样的，只是在插入的时候调用的方法不一样。

红黑树概念

面试时候现场写红黑树代码的概率几乎为 0，但是红黑树一些基本概念还是需要掌握的。

1、它是二叉排序树（继承二叉排序树特显）：

- 若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值。
- 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值。
    - 左、右子树也分别为二叉排序树。

2、它满足如下几点要求：

- 树中所有节点非红即黑。
- 根节点必为黑节点。
- 红节点的子节点必为黑（黑节点子节点可为黑）。
- 从根到 NULL 的任何路径上黑结点数相同。

3、查找时间一定可以控制在 O(logn)。

37、STL 中 unordered_map 和 map 的区别和应用场景

map 支持键值的自动排序，底层机制是红黑树，红黑树的查询和维护时间复杂度均为 $O(\log n)$，但是空间占用比较大，因为每个节点要保持父节点、孩子节点及颜色的信息

unordered_map 是 C++ 11 新添加的容器，底层机制是哈希表，通过 hash 函数计算元素位置，其查询时间复杂度为 O(1)，维护时间与 bucket 桶所维护的 list 长度有关，但是建立 hash 表耗时较大

从两者的底层机制和特点可以看出：map 适用于有序数据的应用场景，unordered_map 适用于高效查询的应用场景

38、hashtable 中解决冲突有哪些方法？

记住前三个：

线性探测

使用 hash 函数计算出的位置如果已经有元素占用了，则向后依次寻找，找到表尾则回到表头，直到找到一个空位

开链

每个表格维护一个 list，如果 hash 函数计算出的格子相同，则按顺序存在这个 list 中

再散列

发生冲突时使用另一种 hash 函数再计算一个地址，直到不冲突

二次探测

使用 hash 函数计算出的位置如果已经有元素占用了，按照 1^2、2^2、3^2...的步长依次寻找，如果步长是随机数序列，则称之为伪随机探测

公共溢出区

一旦 hash 函数计算的结果相同，就放入公共溢出区