**10-C++**

**一、C++基础**

1. **面向对象的三大特征：继承、封装、多态**

2. **static关键字的作用**

* 全局静态变量：全局变量在其它文件是不可见的
* 局部静态变量：作用域为局部作用域，当局部静态变量离开作用域后不会被销毁继续驻留在内存中，直到该函数再次被调用，且值不变
* 静态函数：其它文件可见，但是不可用
* 类的静态数据成员：可以实现多个对象之间的共享，静态数据成员只存在在一个地方，供所有对象共用
* 类的静态成员函数：它不包含类的this指针，不能引用类的非静态成员

3. **static和非static区别**

1. 调用方式‌：

* 静态成员可以直接通过类名调用，不需要创建类的实例。静态成员函数和静态成员变量都属于类，而不是属于类的某个特定实例。因此，它们可以在没有创建类实例的情况下被调用。静态成员函数没有this指针，因此不能直接访问非静态成员变量或调用非静态成员函数。但是，它们可以访问其他静态成员。
* 非静态成员‌（实例成员）必须通过类的实例（对象）来调用。非静态成员函数有this指针，可以访问类的所有成员，包括静态成员。非静态成员函数的生存期与类的实例相关，当实例被创建时，非静态成员函数才会被分配内存。

1. 生命周期‌：

* 静态成员的生命周期与类本身绑定，它们在程序开始时创建，并在程序结束时销毁。静态成员在内存中的位置是固定的，所有实例共享同一个静态成员的拷贝。
* 非静态成员的生命周期与类的实例相关。当类的实例被创建时，非静态成员被分配内存，当实例被销毁时，非静态成员也随之被销毁。

1. 访问权限‌：

* 静态成员可以直接通过类名访问，而不需要创建类的实例。这使得静态成员在多线程环境中特别有用，因为它们不依赖于特定的对象实例，从而避免了线程间的同步问题。
* 非静态成员必须通过类的实例（对象）来访问，它们的值与特定的对象实例相关联。

4. **c++中的四种 cast 转换机制**

* static\_cast：实现基本数据类型之间的转换
* const\_cast 实现从把const成员转换成非const成员
* dynamic\_cast: 将基类指针安全的转换为非基类指针
* reinterpret\_cast 可以转换任何内置数据类

5. **指针和引用的区别，左值引用和右值引用的区别**

* 指针有自己的空间，引用只是别名，显示大小的时候指针是指针类型大小，引用是所指向数据的大小。
* 指针可以被初始化为空，引用必须是一个已有对象的调用
* 指针可以有多级，引用只有一级
* 左值引用是值数据的表达式，比如变量名或者接触引用指针。右值引用表示不能使用地址运算符操作的值。

6. **四个智能指针**

* auto\_ptr：数据只能被同时一个指针持有，发生赋值操作的时候，auto\_ptr会拷贝数据，然后将原来指针置为空，已经没多少人用
* share\_ptr：通过一个引用计数来表示有多少个指针在引用一份数据，当引用计数为0时就会自动删除数据。share\_ptr指针的相互引用就变成了循环引用
* shared\_ptr的核心是引用计数技术。在每个shared\_ptr对象中，都有一个指向所管理对象的指针和一个整型计数器。这个计数器统计有多少个shared\_ptr对象指向该所管理对象。当一个新的shared\_ptr对象指向同一块内存时，该内存的引用计数就会增加1。当一个shared\_ptr对象不再指向该内存时，该内存的引用计数就会减少1。当引用计数为0时，说明没有任何shared\_ptr对象指向该内存，此时该内存将会被自动释放。
* unique\_ptr：只允许一个指针持有数据，想要转移数据给另一个指针需要用移动语义
* weak\_ptr ：是为了解决share\_ptr的循环引用，他是一个弱指针，他指向share\_ptr的时候不会增加Share\_ptr的计数

如果希望只有一个智能指针管理资源或者管理数组就用unique\_ptr，如果希望多个智能指针管理同一个 资源就用shared\_ptr。

**优点：**

* **自动内存管理：**智能指针负责自动释放底层对象，避免内存泄漏。
* **引用计数：**智能指针跟踪对底层对象的引用次数，当不再有引用时自动释放对象。
* **线程安全性：**某些智能指针类型是线程安全的，允许在多线程环境中安全使用。
* **使用简便：**智能指针使用类似内置指针的语法，使用便利。

**潜在缺点：**

* **性能开销：**智能指针比普通指针开销更大，因为需要额外的引用计数机制。
* **潜在错误：**如果智能指针不正确地使用，可能会导致内存错误和异常。
* **所有权复杂性：**在某些情况下，难以确定谁拥有底层对象，这可能导致混乱。

A\* p = new A(10);

shared\_ptr <A> sp1(p), sp2(p);

sp1 和 sp2 并不会共享同一个对 p 的托管计数，而是各自将对 p 的[托管计数](https://zhida.zhihu.com/search?q=%E6%89%98%E7%AE%A1%E8%AE%A1%E6%95%B0&zhida_source=entity&is_preview=1)都记为 1（sp2 无法知道 p 已经被 sp1 托管过）。这样，当 sp1 消亡时要析构 p，sp2 消亡时要再次[析构](https://zhida.zhihu.com/search?q=%E6%9E%90%E6%9E%84&zhida_source=entity&is_preview=1) p，这会导致程序崩溃。

7. **用了智能指针还会内存泄漏吗**

智能指针的使用也存在一些限制和注意事项，不当的使用仍然可能导致内存泄漏。

1. 循环引用‌：当两个或多个对象相互引用时，它们的引用计数都不会变为0，导致内存泄漏。这种情况下，可以使用[weak\_ptr](https://www.baidu.com/s?sa=re_dqa_generate&wd=weak_ptr&rsv_pq=8faaf634013d3c38&oq=%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88%E4%B9%8B%E5%90%8E%E8%BF%98%E4%BC%9A%E5%87%BA%E7%8E%B0%E5%86%85%E5%AD%98%E6%B3%84%E9%9C%B2%E5%90%97&rsv_t=cb7aGIeLruViY0Izv3u02XA7kiY/GbVBGaXZQzU3iq96b/ENUiX6z9jIOWk&tn=baidu&ie=utf-8)来打破循环引用，避免内存泄漏。
2. 裸指针的使用‌：如果使用智能指针管理的对象同时被裸指针引用，当智能指针释放内存时，裸指针仍然指向已经被释放的内存，这可能导致内存泄漏或悬空指针问题。因此，应尽量避免智能指针与裸指针的混合使用。

8. **数组和指针的区别**

* 指针保存的数据的地址，数组直接保存的数据
* 指针分配的空间可以是不连续的，数组分配的空间是连续的
* 指针需要自己分配空间与回收空间，数组是隐式的分配和回收

9. **野指针是什么**

指向一个已经删除的对象的空间或者未分配的空间

10. **析构函数为什么必须是虚函数，为什么C++不是默认的虚构函数**

防止派生类没有被调用析构函数，因为在删除基类指针，如果编译器执行的是静态绑定，则不会调用派生类的析构函数。定义为虚函数会增加类的大小

11. **静态函数和虚函数的区别**

静态函数在编译阶段就已经确定了运行时机，虚函数在运行的时候进行动态绑定

12. **重载和覆盖**

* 重载指的是两个同名函数，他们的参数列表不同
* 覆盖指的是父类和派生类有同名函数

13. **说一说虚函数和多态的理解**

* 多态分为静态多态和动态多态
* 静态多态指的是函数的重载，函数重载是指函数名相同，但是参数列表不同，在编译阶段就已经确定了运行时机。
* 动态多态性是指的基类和派生类的虚函数，当基类中一个函数被声明为虚函数的时候，其派生类对其重写，当用基类指针指向派生类对象调用这个函数时候会调用派生类的函数，这是因为对象中有一个虚表指针，指向对应派生类的虚函数表，虚表指针在父类进行初始化，在派生类中进行赋值。虚函数表是在编译的时候确定的

14. **多态使用场景**

1. 当你有一个基类的指针或引用指向派生类对象时，你想调用派生类的方法，而不是基类的方法，这时就需要用到多态。
2. 实现接口继承和行为配置，可以在不修改现有代码的情况下，通过创建派生类来改变对象的行为。

15. **C++栈的最大空间**

可以使用ulimit查看其大小

16. **extern的作用**

主要是为了保持和C语言的兼容性，同时可以引用其它文件中的函数和变量

17. **new/delete和malloc/free的区别、malloc是如何实现的**

* new操作符是从自由存储区上为对象分配内存空间，malloc函数是从堆上动态分配内存。
* new分配内存成功的时候返回的是对象的指针，而malloc返回的是void\*指针
* malloc/free需要手动计算分配的大小，new/delete是自动计算
* new内存分配出错的时候会报异常，malloc会返回空指针
* malloc/free只是动态的分配内存和释放内存，new/delete不仅仅动态分配内存空间，还会调用构造函数和析构函数进行初始化和清理

18. **RTTI**

运行时类型检查，主要是体现在dynamic\_cast，对基类的指针和引用检查这些指针或者引用实际的派生类类型

19. **参数压栈顺序**

参数压栈是从右到左

20. **拷贝构造函数能不能用值传递**

不能，会造成无限调用，导致栈溢出

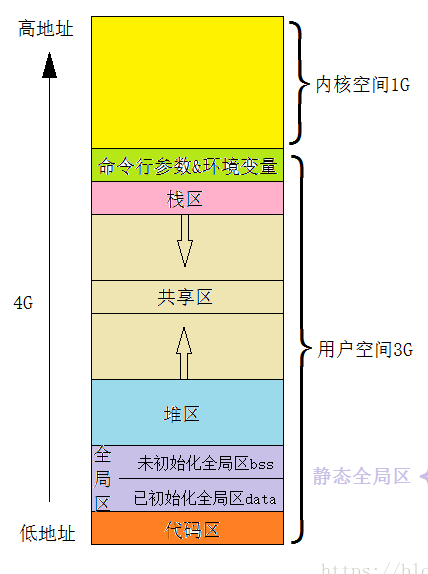
21. **c++类可以定义引用数据成员吗**

可以，在初始化列表中进行初始化

22. **程序编译的全过程**

* 首先是预编译，对伪指令和特殊符号进行处理
* 然后是编译，对程序进行语法和词法分析，然后转换为汇编代码
* 优化程序
* 汇编：将汇编程序转换为目标机器指令
* 链接程序：将相关的目标文件连接在一起最后形成可执行文件

23. **C++中内存分配**



24. **什么时候会发生段错误**

访问了不能访问到的内存

25. **const对象可以调用成员函数吗**

可以调用const数据成员以及const函数，不能调用非const

26. **常量指针和指针常量区别**

常量指针是指向常量的指针，指针常量是指针是一个常量

27. **模板类可以有虚函数吗，模板成员函数可以是虚函数吗**

模板类可以有虚函数，模板成员函数不可以是虚函数，因为模板成员函数在初始化的时候就需要确定

28. **构造函数为什么不能是虚函数**

构造函数发生在虚表指针初始化之前

29. **什么是纯虚函数**

有纯虚函数的类被称为接口，纯虚函数没有定义，只有声明，需要在子类被重载，以多态的方式调用，接口不能被实例化。

30. **c++继承的种类以及原理**

* 利用已有的数据类型来定义新的数据类型，解决代码复用的问题
* 继承种类有：单继承，父类空间在高地址派生类在底地址
* 多继承：根据派生类中声明继承的顺序来排列内存位置
* 菱形继承：会产生二义性
* 虚拟菱形继承：此时基类内存会在最低处，同时对象会有一个虚基类表指针指向虚基类

31. **内存对齐的作用以及规则**

* 为了方便平台的移植，不是所有硬件平台都可以访问任意地址上的数据
* 为了性能，未对齐的数据可能需要两次内存访问
* 按照结构体中基础元素的size最大的成员对齐

32. **typedef和#define区别，#define和Const区别**

* Typedef只是别名，而#define是定义常量在预处理的时候替换
* #define是定义常量在预处理的时候替换，不会做检查
* Const是在编译期间会进行检查

33. **大小端如何确定**

电脑一般是小端，网络传输是大端。通过定义一个联合体，来判别，一般是16为为一组进行反转

34. **函数调用过程**

首先将参数从左到右压入栈中，压入返回地址，再将函数入栈，再将函数局部变量退出

35. **RAII机制是什么**

资源获取就是初始化，为了资源管理，防止内存泄漏，在对象生命周期内控制资源访问，使其始终有效，在对象析构的时候释放资源

36. **什么时候必须使用到初始化成员列表**

* 初始化一个const成员
* 初始化一个引用成员
* 调用基类的构造函数，且函数有一组参数
* 调用一个数据成员的构造函数，且该函数有一组参数

37. **野指针和悬空指针的区别**

* 野指针指的是没有初始化的指针
* 悬空指针指的是释放后没有置空

38. **define和const的区别**

define是在预处理阶段起作用，而const是在编译期间起作用

define只做替换，const可以做安全检查

define在内存中会有多分相同的备份，const只有一份备份

39. **final和override关键字**

override加在函数后面，表示这个函数一定是从父类的虚函数继承过来，编译器会做检查父类是否存在一样的虚函数。

final表示这个函数不能被继承或者重写，编译器会做检查。

40. **构造函数有几类**

默认构造函数

有参数构造函数

拷贝构造函数

转换构造函数

41. **内联和宏区别，类中的内联**

内联会做编译检查，宏不会

内联是函数，宏不是

内联是编译时候展开，宏是预编译的时候展开

析构函数和构造函数默认不会展开。类中的函数默认是内联的，虚函数使用多态性的时候不会展开内联，不使用会展开。

42. **什么时候会用拷贝构造函数**

* 用类的一个实例化对象去初始化另一个对象
* 函数的参数是类的对象（非引用）
* 函数的返回值是函数内部局部对象的类的独享时候，返回的时候会用拷贝构造函数

43. **new的类型**

* 普通的new会返回异常
* new(nothrow) xx 不会返回异常，返回NULL
* placement new 复用已经申请的空间，不能使用delete直接删除空间，需要显示调用析构函数再用delete

44. **NULL和nullptr区别**

NULL在C++表示0，nullptr表示一种类型，表示空指针

45. **static什么时候开始初始化**

初始化只有一次，C语言中在代码执行之前，因此无法使用变量对静态局部变量初始化。C++是在首次使用的时候才会初始化。

46. **值传递、指针传递、引用传递的区别**

值传递要复制值，指针传递需要拷贝指针地址，引用也有地址拷贝但是起了一个别名

47. **this指针是什么时候创建的，放在哪**

在成员函数的开始执行前构造，在成员的执行结束后清除。可能放在栈、寄存器或者全局变量都有可能，通过寄存器传递this指针，同时将对象指针放在另一个寄存器。

48. **什么是虚拟继承**

就是在继承的时候加上virtual字段，在菱形继承的时候保证各个父类实例只有一个，通过在使用虚拟继承的地方加上指针，指针表示偏移量，偏移到同一个地址上

49. **如何获得结构成员相对于字节开头的偏移量**

使用offsetof

50. **c++11的新特性**

nullptr取代NULL

引入了auto和decltype实现类型推导

基于范围的for循环 for(auto &i :res){}

类和结构体中的初始化列表

lambda表达式

std::forward\_list单链表

右值引用和move语义

51. **迭代器中++it和it++区别**

++it返回一个引用，it++返回一个对象

++it不会产生临时对象，it++必须产生临时对象

52. **菱形继承**

菱形继承会导致数据冗余以及二义性问题

子类中有虚表指针，虚基表中存放的是一个偏移量，一个当前位置相对于公共父类成员的偏移量。

53. **析构函数和构造函数里面能抛异常嘛？**

在C++中，构造函数和析构函数在设计和使用上存在一些差异，这些差异直接影响了它们处理异常的能力。从语法上讲，构造函数确实可以抛出异常，这是因为构造函数在对象创建时执行，如果构造函数中遇到错误情况，通过抛出异常可以通知调用者出现问题，从而采取相应的处理措施。然而，构造函数抛出异常后，由于概念上该对象被视为未成功构造，因此该对象的析构函数不会被自动调用。这可能导致资源泄露问题，尤其是在构造函数中分配了内存资源的情况下。因此，虽然构造函数可以抛出异常，但在实践中应谨慎使用，确保在抛出异常前释放所有已分配的资源，以避免内存泄漏12。

相比之下，析构函数在C++标准中明确指出不应该抛出异常。这是因为析构函数的主要职责是清理和释放对象生命周期内申请的资源。如果在析构函数中抛出异常，可能会导致栈展开时程序崩溃或其他不明确的行为。此外，如果析构函数未能完成其清理工作，就意味着资源未被正确释放，这可能导致资源泄露或其他未定义行为。因此，最佳实践是避免在析构函数中抛出异常，而是通过其他方式处理可能出现的错误情况23。

综上所述，虽然C++的构造函数在技术上可以抛出异常，但应谨慎处理以避免资源泄露问题；而析构函数则应避免抛出异常，以确保资源的正确释放和程序的稳定性。

54. **c++抛了异常出现内存泄漏怎么办**

在C++中，如果抛出异常导致内存泄漏，通常是因为在异常被抛出之前，某些资源（如动态分配的内存）没有被正确释放。解决这个问题的关键是确保在异常被抛出时，所有应该被释放的资源都被正确释放。

解决方法：

1. 使用智能指针（如std::unique\_ptr或std::shared\_ptr）来自动管理资源。
2. 使用RAII（资源获取即初始化）技术，将资源管理和对象生命周期绑定。
3. 在catch块中添加throw;以重新抛出异常，以便堆栈展开时，能够释放已获取的资源。
4. 使用try-catch-finally模式，在finally块中释放资源。

**二、STL**

55. **STL中hashTable如何实现**

使用的开链法解决的hash冲突

56. **STL中的两级空间配置器如何实现**

频繁的开辟内存会在堆上造成很多的外部碎片。内存分配器在找不到内存情况下需要合并空闲块，这样会浪费大量时间，降低效率。为了精密分工内存分配，STL 这两个阶段操作区分开来。内存配置操作由 成员函数 allocate() 负责，内存释放由 deallocate() 负责；对象构造由 construct() 负责，对象析构则由 destroy() 负责。

一级配置器：最重要的是allocate、deallocate、reallocate。一级配置器是以malloc、free、realloc执行实际的内存配置

1. 直接allocate分配内存，就是用malloc来分配，成功返回，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，没有就返回异常
3. 如果自定义了处理函数就进行处理，完事就再继续分配

二级配置器：

1. 他维护了16条链表，最小8字节，以8字节递增，最大128。然后用户传入一个字节参数，比如13，他会自动分配到16字节大小的链表，然后查看链表中是否为空，不为空直接从链表中拔出节点来分配空间
2. 如果为空，查看内存池是不是为空，如果不为空，检查内存池是否满足所需20个链表节点大小，满足的话拿出一个给用户，另外19个挂到对应链表上。如果不够，看是否满足一个，满足的话就把一个给用户，然后尽可能多的把节点挂到链表上。如果一个节点都不满足，则把内存池所有剩余的空间挂到对应的链表上去。
3. 内存池为空，申请内存，申请大小为所需内存\*2\*20+一段额外空间，共40，一半拿来用，另一半放到内存池
4. 申请没有成功，申请链表后面比较大的空间，如果都没有，调用一级适配器

57. **vector和list区别**

vector：有连续的内存空间，用于快速查找，插入和删除会造成内存块的拷贝，如果空间不够会申请一整块内存，然后将内容拷贝过去。

list：双链表实现，内存空间不连续，可以实现高效的插入和删除。但是遍历比较慢。

58. **STL 中vector为什么是两倍扩容？它如何释放空间？**

size()函数返回的是已用空间大小，capacity()返回是总空间。如果capacity已经用完，则vector会扩容。VS是1.5倍，GCC是两倍。避免空间的浪费，因为扩充了不一定用，因此最好是在（1，2）之间。两倍扩容那个是因为这样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。vector内存空间增加只增不减，删除元素，空间不会发生改变。

59. **STL容器内部删除一个元素影响**

erase删除顺序容器的时候，使得被删除之后的所有迭代器失效，所以不能使用erase(it++)方式，但是erase返回值是下一个有效的迭代器

erase删除关联容器的时候，只是使得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

60. **map和set如何实现的，为什么红黑树可以实现这两种，为什么用红黑树？**

map和set底层都是使用红黑树实现，因此删除和插入都在O(logn)内实现，用红黑树可以高效下实现map和set自动排序的，所以使用红黑树

61. **如何在共享内存上使用STL标准**

把容器放在共享内存中确定的地址上，然后再通过该共享内存的地址来共享容器

62. **map插入方式**

（1）map.insert(pair<int,string>(1,"xx"))

（2）map.insert(map<int,string>::value\_type(1,"xx"))

（3）map.insert(make\_pair(1,"xx"))

（4）map[1]="xx"

63. **无序图和有序图的区别**

无序图不会排序，有序退会根据键值排序，是按照二叉树存储的。同时无序图底层使用的hash\_table，而有序图使用的红黑树。

64. **vector越界访问下标和map越界访问下标的区别**

* 通过下标访问vector中的元素不会做边界检查，即便下标越界，使用at会做边界检查
* map下标[]是查找key，并且返回相应的值，如果不存在这个key，则会将key和value插入这个map，find会使用关键码执行查找，找到就返回该位置的迭代器，如果不存在则返回尾部迭代器。

65. **说一下STL每种容器对应的迭代器**

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

**三、STL详细数据结构**

1. **序列式容器**

**1.1 数组array**

**（1）原理以及特性**

* array实际上是对C/C++语言中原生数组进行的封装，底层为数组
* 内存分配在栈，大小固定，不能再重新分配，随机访问元素

**（2）头文件**

|  |
| --- |
| Plain Text #include <array> |

**（3）初始化**

|  |
| --- |
| Plain Text array<double, 10> values; //创建了具有10个元素的array，但是并没有默认值  array<double, 10> values {}; //创建了10个元素的array，初始化为0.0  array<double, 10> values {1, 2, 4.0, 2.2}; //创建了10个元素的array，并进行初始化，其余默认为0.0 |

**（4）容量**

* size()：返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。
* max\_size()：返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。

**（5）修改**

* fill(val)：将val这个值赋给容器中的每个元素。
* array1.swap(array2)：交换array1和array2容器中所有的元素，前提是长度和类型相同。

**（6）迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**（7）访问**

* at(n)：返回容器中n位置处元素的引用，该函数自动检查 n 是否在有效的范围内，如果不是则抛出 out\_of\_range 异常。
* front()：返回容器中第一个元素的直接引用，该函数不适用于空的 array 容器。
* back()：返回容器中最后一个元素的直接应用，该函数同样不适用于空的 array 容器。
* data()：返回一个指向容器首个元素的指针。利用该指针，可实现复制容器中所有元素等类似功能。

**（8）遍历**

|  |
| --- |
| Plain Text array<int>::iterator it; for (it = arr.begin(); it != arr.end(); it++)  cout << \*it << endl; //或者 for (size\_t i = 0; i < arr.size(); i++) {  cout << arr.at(i) << endl; } |

**1.2 向量vector**

**（1）原理以及特性**

* vector底层为数组，访问元素方便，因为内存是连续的，插入和删除效率低
* vector由一段连续的线性内存空间实现，扩容会完全弃用现有的内存空间，重新申请更大的内存空间，将旧内存空间中的数据，按原有顺序移动到新的内存空间中，最后将旧的内存空间释放。

**（2）头文件**

|  |
| --- |
| Plain Text #include <vector> |

**（3）初始化**

|  |
| --- |
| Plain Text vector<int> vec; //声明一个int型向量 vector<int> vec(5); //声明一个初始大小为5的int向量 vector<int> vec(10, 1); //声明一个初始大小为10且值都是1的向量 vector<int> vec(tmp); //声明并用tmp向量初始化vec向量 vector<int> tmp(vec.begin(), vec.begin() + 3); //用向量vec的第0个到第2个值初始化tmp int arr[5] = {1, 2, 3, 4, 5}; vector<int> vec(arr, arr + 5); //将arr数组的元素用于初始化vec向量 //说明：当然不包括arr[4]元素，末尾指针都是指结束元素的下一个元素， //这个主要是为了和vec.end()指针统一。 vector<int> vec(&arr[1], &arr[4]); //将arr[1]~arr[4]范围内的元素作为vec的初始值 |

**（4）容量**

* 向量大小： vec.size();
* 向量最大容量： vec.max\_size();
* 更改向量大小： vec.resize();
* 向量真实大小： vec.capacity();
* 向量判空： vec.empty();
* 减少向量大小到满足元素所占存储空间的大小： vec.shrink\_to\_fit();

**（5）修改**

* 多个元素赋值： vec.assign(); //类似于初始化时用数组进行赋值
* 末尾添加元素： vec.push\_back();
* 末尾删除元素： vec.pop\_back();
* 任意位置插入元素： vec.insert();
* 任意位置删除元素： vec.erase();
* 交换两个向量的元素： vec.swap();
* 清空向量元素： vec.clear();
* 任意位置插入元素： de.emplace();
* 末尾添加元素： de.emplace\_back();

**（6）迭代器**

* 开始指针：vec.begin();
* 末尾指针：vec.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针： vec.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： vec.cend();

**（7）访问**

* 下标访问： vec[1]; //并不会检查是否越界
* at方法访问： vec.at(1); //以上两者的区别就是at会检查是否越界，是则抛出out of range异常
* 访问第一个元素： vec.front();
* 访问最后一个元素： vec.back();
* 返回一个指针： int\* p = vec.data(); //可行的原因在于vector在内存中就是一个连续存储的数组，所以可以返回一个指针指向这个数组。这是是C++11的特性。

**（8）遍历**

|  |
| --- |
| Plain Text vector<int>::iterator it; for (it = vec.begin(); it != vec.end(); it++)  cout << \*it << endl; //或者 for (size\_t i = 0; i < vec.size(); i++) {  cout << vec.at(i) << endl; } |

**1.3 双端队列deque**

**（1）原理以及特性**

deque为双端队列，底层为动态数组，提供随机存取，可以在队尾快速的插入或者删除，但是如何是从中间插入元素的时候，速度比较慢，因为要移动元素

**（2）头文件**

|  |
| --- |
| Plain Text #include <deque> |

**（3）初始化**

|  |
| --- |
| Plain Text deque<int> d; //创建一个没有任何元素的空 deque 容器 deque<int> d(10); //创建一个具有 n 个元素的 deque 容器 deque<int> d(10, 5); //创建10个元素的 deque 容器，每个元素值为5  //已有 deque 容器的情况下，可以通过拷贝该容器创建一个新的 deque 容器 deque<int> d1(5);  deque<int> d2(d1);  //拷贝普通数组，创建deque容器 int a[] = { 1,2,3,4,5 }; deque<int>d(a, a + 5); //适用于所有类型的容器 array<int, 5>arr{ 11,12,13,14,15 }; deque<int>d(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15} |

**（4）容量**

* 向量大小： de.size();
* 向量最大容量： de.max\_size();
* 更改向量大小： de.resize();
* 向量判空： de.empty();
* 减少向量大小到满足元素所占存储空间的大小： de.shrink\_to\_fit();

**（5）修改**

* 末尾添加元素： de.push\_back();
* 头部添加元素： de.push\_front();
* 尾部删除元素： de.pop\_back();
* 头部删除元素： de.pop\_front();
* 任意位置插入元素： de.insert();
* 任意位置插入元素： de.emplace()
* 末尾添加元素： de.emplace\_back();
* 头部添加元素： de.emplace\_front();

|  |
| --- |
| Plain Text deque<int> d{ 1,2 }; //第一种格式用法 d.insert(d.begin() + 1, 3);//{1,3,2} //第二种格式用法 d.insert(d.end(), 2, 5);//{1,3,2,5,5} //第三种格式用法 std::array<int, 3>test{ 7,8,9 }; d.insert(d.end(), test.begin(), test.end());//{1,3,2,5,5,7,8,9} //第四种格式用法 d.insert(d.end(), { 10,11 });//{1,3,2,5,5,7,8,9,10,11} |

* 任意位置删除元素： de.erase();
* 交换两个向量的元素： de.swap();
* 清空向量元素： de.clear();

**（6）迭代器**

* 开始指针：de.begin();
* 末尾指针：de.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针： de.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： de.cend();

**（7）访问**

* 下标访问： de[1]; //并不会检查是否越界
* at方法访问： de.at(1); //以上两者的区别就是at会检查是否越界，是则抛出out of range异常
* 访问第一个元素： de.front();
* 访问最后一个元素： de.back();

**（8）遍历**

|  |
| --- |
| Plain Text deque<int>::iterator it; for (it = vec.begin(); it != vec.end(); it++)  cout << \*it << endl; //或者 for (size\_t i = 0; i < vec.size(); i++) {  cout << vec.at(i) << endl; } |

**1.4 双链表list**

**（1）原理以及特性**

list由双向链表实现，不支持随机存储，但是插入和删除速度都很快，查找比较慢

**（2）头文件**

|  |
| --- |
| Plain Text #include <list> |

**（3）初始化**

|  |
| --- |
| Plain Text list<int> values; //创建一个没有任何元素的空list容器 list<int> values(10); //创建一个包含n个元素的list容器 list<int> values(10 , 5); //创建一个包含n个元素的list容器并制定初始值  //通过拷贝已有的list容器创建新的容器 list<int> values1(10 , 5); list<int> values2(values1);  //通过拷贝已有的list容器中的部分元素创建新的容器 //拷贝普通数组，创建list容器 int a[] = { 1,2,3,4,5 }; std::list<int> values(a, a+5); //拷贝其它类型的容器，创建 list 容器 std::array<int, 5>arr{ 11,12,13,14,15 }; std::list<int>values(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15} |

**（4）容量**

* 向量大小：list.size();
* 向量最大容量： list.max\_size();
* 更改向量大小： list.resize();
* 向量判空： list.empty();

**（5）修改**

* 末尾添加元素： list.push\_back();
* 头部添加元素： list.push\_front();
* 尾部删除元素： list.pop\_back();
* 头部删除元素： list.pop\_front();
* 任意位置插入元素： list.emplace();
* 任意位置插入元素： list.insert();

|  |
| --- |
| Plain Text list<int> d{ 1,2 }; //第一种格式用法 d.insert(d.begin() + 1, 3);//{1,3,2} //第二种格式用法 d.insert(d.end(), 2, 5);//{1,3,2,5,5} //第三种格式用法 std::array<int, 3>test{ 7,8,9 }; d.insert(d.end(), test.begin(), test.end());//{1,3,2,5,5,7,8,9} //第四种格式用法 d.insert(d.end(), { 10,11 });//{1,3,2,5,5,7,8,9,10,11} |

* 头部插入元素：list.emplace\_front();
* 尾部插入元素：list.emplace\_back();
* 任意位置删除元素： list.erase();
* 交换两个向量的元素： list.swap();
* 清空向量元素： list.clear();
* 删除容器中所有等于val的值：list.remove(val)
* splice()：将其他 list 容器存储的多个元素添加到当前 list 容器的指定位置处

|  |
| --- |
| Plain Text //创建并初始化 2 个 list 容器 list<int> mylist1{ 1,2,3,4 }, mylist2{10,20,30}; list<int>::iterator it = ++mylist1.begin(); //指向 mylist1 容器中的元素 2  //调用第一种语法格式 mylist1.splice(it, mylist2); // mylist1: 1 10 20 30 2 3 4  // mylist2:  // it 迭代器仍然指向元素 2，只不过容器变为了 mylist1 //调用第二种语法格式，将 it 指向的元素 2 移动到 mylist2.begin() 位置处 mylist2.splice(mylist2.begin(), mylist1, it); // mylist1: 1 10 20 30 3 4  // mylist2: 2  // it 仍然指向元素 2  //调用第三种语法格式，将 [mylist1.begin(),mylist1.end())范围内的元素移动到 mylist.begin() 位置处  mylist2.splice(mylist2.begin(), mylist1, mylist1.begin(),mylist1.end());//mylist1:  //mylist2:1 10 20 30 3 4 2 |

**（6）迭代器**

* 开始指针：list.begin();
* 末尾指针：list.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针： list.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： list.cend();

**（7）访问**

* 访问第一个元素： de.front();
* 访问最后一个元素： de.back();

**（8）遍历**

|  |
| --- |
| Plain Text const std::list<int> mylist{1,2,3,4,5}; auto it = mylist.begin(); cout << \*it << " "; ++it; while (it!=mylist.end()) {  cout << \*it << " ";  ++it;  } |

**1.5 单链表 forward\_list**

**（1）原理以及特性**

list由单向链表实现，不支持随机存储，但是插入和删除速度都很快，查找比较慢，是一类比链表容器快、更节省内存的容器。

**（2）头文件**

|  |
| --- |
| Plain Text #include <forward\_list> |

**（3）初始化**

|  |
| --- |
| Plain Text list<int> values; //创建一个没有任何元素的空list容器 list<int> values(10); //创建一个包含n个元素的list容器 list<int> values(10 , 5); //创建一个包含n个元素的list容器并制定初始值  //通过拷贝已有的list容器创建新的容器 list<int> values1(10 , 5); list<int> values2(values1);  //通过拷贝已有的list容器中的部分元素创建新的容器 //拷贝普通数组，创建list容器 int a[] = { 1,2,3,4,5 }; std::list<int> values(a, a+5); //拷贝其它类型的容器，创建 list 容器 std::array<int, 5>arr{ 11,12,13,14,15 }; std::list<int>values(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15} |

**（4）容量**

* 向量最大容量： list.max\_size();
* 更改向量大小： list.resize();
* 向量判空： list.empty();

**（5）修改**

* 头部添加元素： list.push\_front();
* 头部删除元素： list.pop\_front();
* 任意位置插入元素，并返回位置指针： list.insert\_after();

|  |
| --- |
| Plain Text list<int> d{ 1,2 }; //第一种格式用法 d.insert\_after(d.begin() + 1, 3);//{1,3,2} //第二种格式用法 d.insert\_after(d.end(), 2, 5);//{1,3,2,5,5} //第三种格式用法 std::array<int, 3>test{ 7,8,9 }; d.insert\_after(d.end(), test.begin(), test.end());//{1,3,2,5,5,7,8,9} //第四种格式用法 d.insert\_after(d.end(), { 10,11 });//{1,3,2,5,5,7,8,9,10,11} |

* 头部插入元素：list.emplace\_front();
* 任意位置插入元素，并返回位置指针： list.emplace\_after();
* 任意位置删除元素： list.erase();
* 交换两个向量的元素： list.swap();
* 清空向量元素： list.clear();
* 删除容器中所有等于val的值：list.remove(val)

**（6）迭代器**

* 开始指针：list.begin();
* 末尾指针：list.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针： list.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： list.cend();

**（7）访问**

访问第一个元素：list.front();

2. **关联式容器**

**2.1 pair**

**原理**

其专门用来将 2 个普通元素 first 和 second（可以是 C++ 基本数据类型、结构体、类自定的类型）创建成一个新元素<first, second>

**头文件**

|  |
| --- |
| Plain Text #include <utility> |

**初始化**

|  |
| --- |
| Plain Text //默认构造函数，即创建空的 pair 对象 pair();  pair <string, double> pair1;  // 直接使用 2 个元素初始化成 pair 对象 pair (const first\_type& a, const second\_type& b); pair <string, string> pair2("aaa","bbb");   // 拷贝（复制）构造函数，即借助另一个 pair 对象，创建新的 pair 对象 template<class U, class V> pair (const pair<U,V>& pr); pair <string, string> pair3(pair2);  // 移动构造函数 template<class U, class V> pair (pair<U,V>&& pr); pair <string, string> pair4(make\_pair("aaa", "bbb"));  // 使用右值引用参数，创建 pair 对象 template<class U, class V> pair (U&& a, V&& b); pair <string, string> pair5(string("aaa"), string("bbb")) |

**使用方法**

|  |
| --- |
| Plain Text pair <string, string> p("aaa","bbb");  cout<<p.first; //aaa cout<<p.second; //bbb |

**2.2 map**

**原理**

* 作为关联式容器的一种，map 容器存储的都是 pair 对象，也就是用 pair 类模板创建的键值对。其中，各个键值对的键和值可以是任意数据类型，包括 C++ 基本数据类型（int、double 等）、使用结构体或类自定义的类型
* 是由红黑树(RBTree)实现

**头文件**

|  |
| --- |
| Plain Text #include <map> |

**初始化**

|  |
| --- |
| Plain Text map<string, int>myMap; //创建一个空的map容器  //创建map容器的同时进行初始化 map<string, int>myMap{ {"a",10},{"b",20} };  map<string, int>myMap{std::make\_pair("a",10),std::make\_pair("b",20)};  map<string, int>newMap(myMap); //利用已有的map容器创建新的map容器  利用已有map容器中的部分键值对创建新的map容器 map<std::string, int>myMap{ {"a",10},{"b",20} }; map<std::string, int>newMap(++myMap.begin(), myMap.end()); |

**容量**

* size()：返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。
* max\_size()：返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。

**修改**

* clear(); 清空容器
* erase()； 删除指定key的键值对
* insert(); 插入键值对

|  |
| --- |
| Plain Text std::pair<string, string> STL = { "STL教程","http://c.biancheng.net/stl/" }; //指定要插入的位置 std::map<string, string>::iterator it = mymap.begin(); //向 it 位置以普通引用的方式插入 STL auto iter1 = mymap.insert(it, STL); |

* emplace(); 插入键值对

|  |
| --- |
| Plain Text std::map<string, string>mymap;  //插入键值对 pair<map<string, string>::iterator, bool> ret = mymap.emplace("STL教程", "http://c.biancheng.net/stl/"); |

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**访问**

|  |
| --- |
| Plain Text //创建并初始化 map 容器 std::map<std::string, std::string>myMap{ {"a","1"},  {"b","2"},  {"c","3"} };   cout << myMap["a"] << endl; //方式一 如果不存在则会添加一个新的键值对 cout << myMap.at("b") << endl; //方式二 如果不存在则抛出异常 myMap.count("a"); //查找map是否存在，存在几个 |

**遍历**

|  |
| --- |
| Plain Text std::map<std::string, std::string>myMap{ {"a","1"},  {"b","2"},  {"c","3"} }; for (auto iter = myMap.begin(); iter != myMap.end(); ++iter) {  if (!iter->first.compare("c")) {   } } |

**2.3 multimap**

**原理**

multimap容器与map容器比较相似，multimap 容器也用于存储 pair<const K， T> 类型的键值对（其中 K 表示键的类型，T 表示值的类型），其中各个键值对的键的值不能做修改；并且，该容器也会自行根据键的大小对存储的所有键值对做排序操作。和 map 容器的区别在于，multimap 容器中可以同时存储多（≥2）个键相同的键值对

**头文件**

|  |
| --- |
| Plain Text #include <map> |

**初始化**

|  |
| --- |
| Plain Text multimap<string, int>myMap; //创建一个空的map容器  //创建map容器的同时进行初始化 multimap<string, int>myMap{ {"a",10},{"b",20} };  multimap<string, int>myMap{std::make\_pair("a",10),std::make\_pair("b",20)};  multimap<string, int>newMap(myMap); //利用已有的map容器创建新的map容器  利用已有map容器中的部分键值对创建新的map容器 multimap<std::string, int>myMap{ {"a",10},{"b",20} }; multimap<std::string, int>newMap(++myMap.begin(), myMap.end()); |

**容量**

* size()：返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。
* max\_size()：返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。

**修改**

* clear(); 清空容器
* cout(key); 查找键为key的键值对个数
* erase()； 删除指定key的键值对
* insert(); 插入键值对

|  |
| --- |
| Plain Text pair<string, string> STL = { "STL教程","http://c.biancheng.net/stl/" }; //指定要插入的位置 multimap<string, string::iterator it = mymap.begin(); //向 it 位置以普通引用的方式插入 STL auto iter1 = mymap.insert(it, STL); |

* emplace(); 插入键值对

|  |
| --- |
| Plain Text multimap<string, string>mymap;  //插入键值对 pair<multimap<string, string::iterator, bool> ret = mymap.emplace("STL教程", "http://c.biancheng.net/stl/"); |

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**访问**

|  |
| --- |
| Plain Text //创建并初始化 map 容器 multimap<std::string, std::string>myMap{ {"a","1"},  {"b","2"},  {"c","3"} };   cout << myMap["a"] << endl; //方式一 如果不存在则会添加一个新的键值对 cout << myMap.at("b") << endl; //方式二 如果不存在则抛出异常 |

**遍历**

|  |
| --- |
| Plain Text std::multimap<std::string, std::string>myMap{ {"a","1"},  {"b","2"},  {"c","3"} }; for (auto iter = myMap.begin(); iter != myMap.end(); ++iter) {  if (!iter->first.compare("c")) {   } } |

**2.4 set**

**原理**

* 使用红黑树实现
* 使用 set 容器存储的各个键值对，要求键 key 和值 value 必须相等。

**头文件**

|  |
| --- |
| Plain Text #include <set> |

**初始化**

|  |
| --- |
| Plain Text set<std::string> myset; //创建一个空的map容器  //创建map容器的同时进行初始化 std::set<std::string> myset{"http://c.biancheng.net/java/",  "http://c.biancheng.net/stl/",  "http://c.biancheng.net/python/"};  set<std::string> copyset(myset); //利用已有的map容器创建新的map容器，等同于set<std::string> copyset = myset  //利用已有map容器中的部分键值对创建新的map容器 set<std::string> myset{ "http://c.biancheng.net/java/",  "http://c.biancheng.net/stl/",  "http://c.biancheng.net/python/" }; set<std::string> copyset(++myset.begin(), myset.end()); |

**容量**

* size()：返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。
* max\_size()：返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。

**修改**

* clear(); 清空容器
* erase()； 删除数据

|  |
| --- |
| Plain Text //创建并初始化 set 容器 std::set<int>myset{1,2,3,4,5};   //1) 调用第一种格式的 erase() 方法 int num = myset.erase(2); //删除元素 2，myset={1,3,4,5}  //2) 调用第二种格式的 erase() 方法 set<int>::iterator iter = myset.erase(myset.begin()); //删除元素 1，myset={3,4,5}  //3) 调用第三种格式的 erase() 方法 set<int>::iterator iter2 = myset.erase(myset.begin(), --myset.end());//删除元素 3,4，myset={5} |

* insert(); 插入数据

|  |
| --- |
| Plain Text //创建并初始化set容器 std::set<std::string> myset; //准备接受 insert() 的返回值 pair<set<string>::iterator, bool> retpair; //采用普通引用传值方式 string str = "http://c.biancheng.net/stl/"; retpair = myset.insert(str); |

* emplace(); 插入数据

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**访问**

只能借助迭代器，没有重载[]，也没有at

**2.5 multiset**

**原理**

* 使用红黑树实现
* 使用 set 容器存储的各个键值对，要求键 key 和值 value 必须相等。
* multiset允许数据重复

**头文件**

|  |
| --- |
| Plain Text #include <set> |

**初始化**

|  |
| --- |
| Plain Text multiset<std::string> myset; //创建一个空的map容器  //创建map容器的同时进行初始化 multiset<std::string> myset{"http://c.biancheng.net/java/",  "http://c.biancheng.net/stl/",  "http://c.biancheng.net/python/"};  multiset<std::string> copyset(myset); //利用已有的map容器创建新的map容器，等同于set<std::string> copyset = myset  //利用已有map容器中的部分键值对创建新的map容器 multiset<std::string> myset{ "http://c.biancheng.net/java/",  "http://c.biancheng.net/stl/",  "http://c.biancheng.net/python/" }; multiset<std::string> copyset(++myset.begin(), myset.end()); |

**容量**

* size()：返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。
* max\_size()：返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。

**修改**

* clear(); 清空容器
* count(); 查找键值为Key的个数
* erase()； 删除数据

|  |
| --- |
| Plain Text //创建并初始化 set 容器 multiset<int>myset{1,2,3,4,5};   //1) 调用第一种格式的 erase() 方法 int num = myset.erase(2); //删除元素 2，myset={1,3,4,5}  //2) 调用第二种格式的 erase() 方法 multiset<int>::iterator iter = myset.erase(myset.begin()); //删除元素 1，myset={3,4,5}  //3) 调用第三种格式的 erase() 方法 multiset<int>::iterator iter2 = myset.erase(myset.begin(), --myset.end());//删除元素 3,4，myset={5} |

* insert(); 插入数据

|  |
| --- |
| Plain Text //创建并初始化set容器 multiset<std::string> myset; //准备接受 insert() 的返回值 pair<multiset<string>::iterator, bool> retpair; //采用普通引用传值方式 string str = "http://c.biancheng.net/stl/"; retpair = myset.insert(str); |

* emplace(); 插入数据

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**访问**

* 只能借助迭代器，没有重载[]，也没有at

3. **无序关联式容器**

**3.1 unordered\_map unordered\_multimap**

**原理**

* 底层采用哈希表实现无序容器时，会将所有数据存储到一整块连续的内存空间中，并且当数据存储位置发生冲突时，解决方法选用的是“链地址法”
* unordered\_multimap允许多个重复的键值对，而且count(key)可以查找重复键值对的个数

**头文件**

|  |
| --- |
| Plain Text #include <unordered\_map> |

**初始化**

|  |
| --- |
| Plain Text unordered\_map<std::string, int>umap; //创建一个空的unordered\_map容器  //创建unordered\_map容器的同时进行初始化 unordered\_map<std::string, int>umap{ {"C语言教程",10},{"STL教程",20} };  unordered\_map<std::string, int>umap2(umap); //利用已有的unordered\_map容器创建新的map容器  //利用已有unordered\_map容器中的部分键值对创建新的unordered\_map容器 unordered\_map<std::string, int>umap{ {"C语言教程",10},{"STL教程",20} }; unordered\_map<std::string, int>umap2(++umap.begin(), umap.end()); |

**容量**

|  |
| --- |
| Plain Text size()：返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。 max\_size()：返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。 empty()：判断容器是否为空，相当于size()==0，但是其效率更高。 |

**修改**

* clear(); 清空容器
* erase()； 删除指定key的键值对

|  |
| --- |
| Plain Text //创建 umap 容器 unordered\_map<string, string> umap{  {"STL教程", "http://c.biancheng.net/stl/"},  {"Python教程", "http://c.biancheng.net/python/"},  {"Java教程", "http://c.biancheng.net/java/"} }; //first 指向第一个键值对 unordered\_map<string, string>::iterator first = umap.begin(); //last 指向最后一个键值对 unordered\_map<string, string>::iterator last = --umap.end(); //删除[fist,last)范围内的键值对 auto ret = umap.erase(first, last); |

* find(); 查找

|  |
| --- |
| Plain Text //创建 umap 容器 unordered\_map<string, string> umap{  {"a","1"},  {"b","2"},  {"c","3"}}; //查找成功 unordered\_map<string, string>::iterator iter = umap.find("a");  //查找失败 unordered\_map<string, string>::iterator iter2 = umap.find("d"); if (iter2 == umap.end()) {  cout << "当前容器中没有以\"d\"为键的键值对"; } |

* insert(); 插入键值对

|  |
| --- |
| Plain Text std::pair<string, string> STL = { "STL教程","http://c.biancheng.net/stl/" }; //指定要插入的位置 unordered\_map<string, string>::iterator it = mymap.begin(); //向 it 位置以普通引用的方式插入 STL auto iter1 = mymap.insert(it, STL); |

* emplace(); 插入键值对

|  |
| --- |
| Plain Text std::unordered\_map<string, string>mymap;  //插入键值对 pair<unordered\_map<string, string>::iterator, bool> ret = mymap.emplace("STL教程", "http://c.biancheng.net/stl/"); |

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**访问**

|  |
| --- |
| Plain Text //创建并初始化 map 容器 unordered\_map<std::string, std::string>myMap{ {"a","1"},  {"b","2"},  {"c","3"} };   cout << myMap["a"] << endl; //方式一 如果不存在则会添加一个新的键值对 cout << myMap.at("b") << endl; //方式二 如果不存在则抛出异常 |

**遍历**

|  |
| --- |
| Plain Text unordered\_map<std::string, std::string>myMap{ {"a","1"},  {"b","2"},  {"c","3"} }; for (auto iter = myMap.begin(); iter != myMap.end(); ++iter) {  if (!iter->first.compare("c")) {   } } |

**3.2 unordered\_set unordered\_multiset**

**原理**

* 底层采用哈希表实现无序容器时，会将所有数据存储到一整块连续的内存空间中，并且当数据存储位置发生冲突时，解决方法选用的是“链地址法”
* unordered\_multiset允许多个重复的键值对，而且count(key)可以查找重复键值对的个数

**头文件**

|  |
| --- |
| Plain Text #include <unordered\_set> |

**初始化**

|  |
| --- |
| Plain Text unordered\_set<std::string, int>umap; //创建一个空的unordered\_map容器  //创建unordered\_map容器的同时进行初始化 unordered\_set<std::string, int>umap{ "a"，"b" };  unordered\_set<std::string, int>umap2(umap); //利用已有的unordered\_map容器创建新的map容器  //利用已有unordered\_map容器中的部分键值对创建新的unordered\_map容器 unordered\_set<std::string, int>umap{ "a","b"}; unordered\_set<std::string, int>umap2(++umap.begin(), umap.end()); |

**容量**

* size()：返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。
* max\_size()：返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。

**修改**

* clear(); 清空容器
* erase()； 删除指定key的键值对

|  |
| --- |
| Plain Text //创建并初始化 set 容器 std::set<int>myset{1,2,3,4,5}; cout << "myset size = " << myset.size() << endl;//5  //1) 调用第一种格式的 erase() 方法 int num = myset.erase(2); //删除元素 2，myset={1,3,4,5}  //2) 调用第二种格式的 erase() 方法 set<int::iterator iter = myset.erase(myset.begin()); //删除元素 1，myset={3,4,5}  //3) 调用第三种格式的 erase() 方法 set<int::iterator iter2 = myset.erase(myset.begin(), --myset.end());//删除元素 3,4，myset={5} |

* insert(); 插入键值对

|  |
| --- |
| Plain Text //创建并初始化set容器 std::set<std::string> myset; //准备接受 insert() 的返回值 pair<set<string::iterator, bool> retpair; //采用普通引用传值方式 string str = "http://c.biancheng.net/stl/"; retpair = myset.insert(str); |

* emplace(); 插入键值对

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**访问**

只能通过迭代器访问

4. **其他容器**

**4.1 string**

**（1）原理**

就是普通字符串的封装

**（2）头文件**

|  |
| --- |
| Plain Text #include <string> |

**（3）初始化**

|  |
| --- |
| Plain Text string strs ;//生成空字符串 string s(str);//生成字符串str的复制品 string s(str, stridx) ; //将字符串str中始于stridx的部分作为构造函数的初值 string s(str, strbegin, strlen); //将字符串str中始于strbegin、长度为strlen的部分作为字符串初值 string s(cstr);//以C\_string类型cstr作为字符串s的初值 string s(cstr,char\_len);//以C\_string类型cstr的前char\_len个字符串作为字符串s的初值 string s(num, c);//生成一个字符串，包含num个c字符 string s(strs, beg, end) ;//以区间[beg, end]内的字符作为字符串s的初值 |

**（4）关键函数**

|  |
| --- |
| Plain Text size()：返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。 max\_size()：返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。 empty()：判断容器是否为空，相当于size()==0，但是其效率更高。 data(); 将内容以const char\*形式返回 copy(); 将内容复制为一个string形式 substr(); 返回子字符串 c\_str(); 将内容以const char\*形式返回 find(); 查找某子字符串 push(); 推入到字符串尾部 |

（5）迭代器

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**4.2 stack**

**原理**

由于 stack 适配器以模板类 stack<T,Container=deque>的形式位于头文件中

**头文件**

|  |
| --- |
| Plain Text #include <stack> |

**初始化**

|  |
| --- |
| Plain Text std::stack<int> values;//创建一个不包含任何元素的 stack 适配器，并采用默认的 deque 基础容器  \\stack<T,Container=deque> 模板类提供了 2 个参数，通过指定第二个模板类型参数，我们可以使 \\用出 deque 容器外的其它序列式容器，只要该容器支持  \\empty()、size()、back()、push\_back()、pop\_back() 这 5 个成员函数即可 std::stack<std::string, std::list<int>> values; std::list<int> values {1, 2, 3}; std::stack<int,std::list<int>> my\_stack (values);  std::list<int> values{ 1, 2, 3 }; std::stack<int, std::list<int>> my\_stack1(values); std::stack<int, std::list<int>> my\_stack=my\_stack1; |

**关键函数**

* size()：返回容器中当前元素的数量
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。
* top(); 返回栈中第一个元素的引用，如果栈为空，则程序会报错
* push(); 推入一个元素 支持右值引用
* pop(); 弹出一个元素，没有返回值

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个 元素的下一个位置
* 指向常量的 开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**4.3 queue**

**原理**

由于 queue适配器以模板类 queue<T,Container=deque>的形式位于头文件中

**头文件**

|  |
| --- |
| Plain Text #include <queue> |

**初始化**

|  |
| --- |
| Plain Text std::queue<int> values;//创建一个不包含任何元素的 stack 适配器，并采用默认的 deque 基础容器  \\queue<T,Container=deque> 模板类提供了 2 个参数，通过指定第二个模板类型参数，我们可以使 \\用出 deque 容器外的其它序列式容器，只要该容器支持  \\front()、back()、push\_back()、pop\_front()、empty() 和 size() 这 5 个成员函数即可 std::queue<std::string, std::list<int>> values; std::list<int> values {1, 2, 3}; std::queue<int,std::list<int>> queue(values);  std::list<int> values{ 1, 2, 3 }; std::queue<int, std::list<int>> my\_queue1(values); std::queue<int, std::list<int>> my\_queue=my\_queue1; |

**关键函数**

* size()：返回容器中当前元素的数量
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。
* front(); 返回队列中第一个元素的引用，如果队列为空，则程序会报错
* back(); 返回队列中最后一个元素的引用，如果队列为空，则程序会报错
* push(); 推入队尾一个元素 支持右值引用
* pop(); 弹出队头一个元素，没有返回值

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbeg in(); //意思就是不能 通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();

**4.4 priority\_queue**

**原理**

由于 priority\_queue适配器以模板类 priority\_queue<T,Container=deque>的形式位于头文件中

**头文件**

|  |
| --- |
| Plain Text #include <queue> |

**初始化**

|  |
| --- |
| Plain Text std::priority\_queue<int> values;//创建一个不包含任何元素的 stack 适配器，并采用默认的 deque 基础容器  \\priority\_queue<T,Container=deque> 模板类提供了 2 个参数，通过指定第二个模板类型参数，我们可以使 \\用出 deque 容器外的其它序列式容器，只要该容器支持  \\front()、back()、push\_back()、pop\_front()、empty() 和 size() 这 5 个成员函数即可 std::priority\_queue<std::string, std::list<int>> values; std::list<int> values {1, 2, 3}; std::priority\_queue<int,std::list<int>> queue(values);  std::list<int> values{ 1, 2, 3 }; std::priority\_queue<int, std::list<int>> my\_queue1(values); std::priority\_queue<int, std::list<int>> my\_queue=my\_queue1; |

**关键函数**

* size()：返回容器中当前元素的数量
* empty()：判断容器是否为空，相当于size()==0，但是其效率更高。
* front(); 返回队列中第一个元素的引用，如果队列为空，则程序会报错
* back(); 返回队列中最后一个元素的引用，如果队列为空，则程序会报错
* push(); 推入队尾一个元素 支持右值引用
* pop(); 弹出队头一个元素，没有返回值

**迭代器**

* 开始指针：arr.begin();
* 末尾指针：arr.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针：arr.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： arr.cend();