一个容器就是一些特定类型对象的集合。顺序容器为程序员提供了控制元素存储和访问顺序的能力，与元素加入容器时的位置相对应。无关联容器则根据关键字的值来存储元素。

# 1.数序容器概述：

|  |  |
| --- | --- |
| vector | 可变大小数组，支持快速随机访问，在尾部之外的位置插入/删除元素会很慢 |
| string | 与vector相似，但用于保存字符 |
| deque | 双端队列，支持快速随机访问，在头尾位置插入/删除速度很快 |
| list | 双向链表，支持双向顺序访问，在任何位置插入/删除速度很快 |
| forward\_list | 单向链表，支持单向顺序访问，在任何位置插入/删除速度很快 |
| array | 固定大小数组，支持快速随机访问，不能添加/删除元素 |

每个容器都定义在一个文件名与类型名相同的头文件中，除了固定的array外，其他容器都支持高效、灵活的内存管理。list、forward\_list容器将元素保存在不连续的内存空间中，和其他四个相比，这两个容器的额外内存开销很大。以上容器都在以下方面进行折中：①向容器中添加或删除元素的快慢，②非顺序访问的快慢。与内置函数相比，array是一种更安全、更容易使用的数组类型。

forward\_list的设计目标是达到与最好的手写的单向链表数据结构相当的性能，没有size操作，因为保存或计算大小就会比手写链表多出额外的开销。对于其他容器而言，size保证是一个快速的常量时间的操作。

容器选择规则：①一般用vector，②如果有很多小元素，其空间的额外开销很重要，则不要使用list、forward\_list，③随机访问元素：使用vector、dequeue、array、string，④要求在中间插入删除元素，用list、forward\_list，⑤要求在头尾插入元素，使用deque，⑥如果程序只有在读取操作时才需要在容器中间插入元素，随后需要随机访问元素，则：首先确定是否真的需要在容器中间位置添加元素。当处理输入数据时，通常可以很容易向vector追加数据，然后再调用标准库的sort函数来重排容器中的元素，从而避免在中间位置添加元素；如果必须在中间插入元素，则在输入阶段使用list，一旦输入完成，拷贝到vector中；⑦如果既需要随机访问元素，又需要在容器中间位置插入元素，这就要取决于在list或forward\_list中访问元素与vector或deque中插入元素的相对性能，看看执行的访问多还是插入多。如果不确定使用哪种容器，可以在程序中只适用vector和list公共的操作：使用迭代器，不使用下标，避免随机访问。

# 2.容器库概览：

容器类型上的操作有：①某些操作是所有容器类型都提供的，②有的操作仅对顺序容器、关联容器、无需容器提供，③有的只适用于一小部分容器。

vector<noDefault> v(10);//如果noDefault类没有对应构造函数，则错误

|  |  |
| --- | --- |
| **类别别名：** | |
| iterator | 此容器类型的迭代器类型 |
| const\_ iterator | 可以读取元素，但不能修改的迭代器类型 |
| size\_type | 无符号整数类型，足够保存此种容器类型最大大小 |
| difference\_type | 带符号整数类型，足够保存两个迭代器之间的距离 |
| value\_type | 元素类型 |
| reference | 元素的左值类型，与value\_type &含义相同 |
| const\_reference | 元素的const左值类型（即const value\_type &） |
| **初始化构造函数：** | |
| C c; | 默认构造函数构造容器 |
| C c(b,e); | 构造c，将迭代器b和e指定的范围内的元素拷贝到c（array不支持） |
| C c1(c2); | 构造c2的拷贝c1(array要求两者大小相同) |
| C c1 = c2; |
| C c{a,b,c…}; | 列表初始化(array要求列表中元素数目必须等于或小于array的大小，遗漏的元素将进行值初始化) |
| C c = {a,b,c,…} |
| C c(n); | seq包含n个元素，这些元素都进行值初始化，此构造函数是explicit的（string不支持） |
| C c(n,t); | c包含n个初始值为t的元素（关联容器不支持） |
| **赋值：** | |
| c1 = c2; | 将c1中的元素替换为c2中元素 |
| c1 = {a,b,c,…} | 将c1中的元素替代为列表中的元素（array不支持） |
| **swap：** | |
| a.swap(b); | 交换 |
| swap(a,b); | 交换 |
| **assign操作（仅顺序容器，关联容器和array不支持）：** | |
| c.assign(b,e); | 将c中的元素替换为迭代器b和e所表示的范围中的元素，迭代器b和e不能指向c中的元素 |
| c.assign(il); | 将c中的元素替换为初始化列表il中的元素 |
| c.assign(n,t); | 将c中的元素替换为n个值为t的元素 |
| **添加、删除元素（array不支持）：** | |
| c.insert(args); | 将args中的元素拷贝到c |
| c.emplace(inits); | 使用inits构造c中的一个元素 |
| c.erase(args); | 删除args指定的元素 |
| c.clear(); | 删除所有元素，返回void |
| **大小：** | |
| c.size(); | c中元素的数目（forward\_list不支持） |
| c.max\_size(); | c中可保存的最大元素数目 |
| c.empty(); | 检测c是否为空 |
| **获取迭代器：** | |
| c.begin() c.end() | 返回指向c的首元素和尾元素之后位置的迭代器 |
| c.cbegin() c.cend() | 返回const\_iterator |
| **关系运算符：** | |
| == != | 运算符 |
| < <= > >= | 运算符（无序关联容器不支持） |
| **反向容器的额外成员（forward\_list不支持）：** | |
| reverse\_iterator | 按逆序寻址元素的迭代器 |
| const\_ reverse\_iterator | 不能修改元素 |
| c.rbegin() c.rend() | 返回指向c的尾元素和首元素之前位置的迭代器 |
| c.crbegin() c.crend() | 不能修改 |

## 2.1迭代器：

迭代器有公共的接口：如果一个迭代器提供某个操作，那么所有提供相同操作的迭代器对这个操作的实现方式是相同的。

|  |  |
| --- | --- |
| 所有迭代器支持的运算符（forward\_list迭代器不支持） | |
| \*iter | 返回迭代器iter所指元素的引用 |
| iter->mem | 解引用iter并获取该元素的名为mem的成员，等价于(\*iter).mem |
| ++iter | 令iter指示容器中的下一个元素 |
| --iter | 令iter指示容器中的上一个元素 |
| iter1 == iter2 | 判断两个迭代器是否相等（不相等），如果两个迭代器指示的是同一个元素或者它们是同一个容器的尾后迭代器，则相等；反之，不相等 |
| iter1 != iter2 |

|  |  |
| --- | --- |
| 部分迭代器支持的运算符（list、forward\_list不支持） | |
| iter + n | 迭代器加上一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向前移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置 |
| iter - n | 迭代器减去一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向后移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置 |
| iter1 += n | 迭代器加法的复合赋值语句，将iter1加n的结果赋给iter1 |
| iter1 -= n | 迭代器减法的复合赋值语句，将iter1减n的结果赋给iter1 |
| iter1 - iter2 | 两个迭代器相减的结果是它们之间的距离，也就是说，将运算符右侧的迭代器向前移动差值个元素后将得到左侧的迭代器。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置 |
| >、 >=、 <、<= | 迭代器的关系运算符，如果某迭代器指向的容器位置在另一个迭代器所指位置之前，则说前者小于后者。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置（无序关联容器不支持）。 |

begin和end构成迭代器范围的要求：①它们指向同一个容器中的元素，或者是容器最后一个元素之后的位置，②可以通过反复递增begin来到达end（即end不在begin之前）。

## 2.2容器类型成员：

反向迭代器各种操作的含义发生了颠倒，比如对一个反向迭代器执行++操作，会得到上一个元素。

vector<int>::difference\_type count; //使用vector<int>类的difference\_type成员

## 2.3begin和end成员：

begin和end包含带r的版本和带c（const）的版本（c在r前）。当auto和begin或end结合使用时，获得的迭代器类型依赖于容器类型。

auto it = a.begin();//当a是const时，it是const\_iterator，依赖于a的类型

auto it = a.cbegin();//it是const\_iterator

## 2.4容器定义和初始化：

除array之外，其他容器的默认构造函数都会创建一个指定类型的空容器，且都可以接受指定容器大小和元素初始值的参数。

2.4.1将一个容器初始化为另一个容器的拷贝：

初始化时，为了创建一个容器为另一个容器的拷贝，两个容器的类型及其元素类型必须匹配。不过，当传递迭代器参数来拷贝一个范围时，就不要求容器类型相同，而且新容器和原容器中的元素类型也可以不同，只要能进行转换即可。

vector<const char \*> articles = {“a”,”an”,”the”};

vector<string> words(articles);//错误，容器类型要匹配

forward\_list<string> words(srticles.begin(),articles.end());//正确

2.4.2列表初始化：

2.4.3与顺序容器大小相关的构造函数：

如果元素类型是内置类型或者是具有默认构造函数的类类型，可以只为构造函数提供一个容器大小的参数；如果元素类型没有构造函数，则除了大小参数外，还必须指定一个显式的元素初始值。

list<string> vec(10,”hi”);

forward\_list<int> vec(10);

当定义一个array时，除了指定元素类型，还要指定容器大小。

array<int,42>::size\_type t;

array不支持普通的容器构造函数，这些函数都会确定容器的大小，要么隐式地、要么显式地。而允许用户向一个array构造函数传递大小参数，最好情况下也是多余的，而且容易出错。一个默认构造函数array是非空的：它包含了与其大小一样多的元素，这些元素都被默认初始化，就像一个内置数组中的元素那样。

array<xclass,42> ia;//注意：要求xclass类必须有默认构造函数

虽然不能对内置数组类型进行拷贝或对象赋值，但array并无此限制。

## 2.5赋值和swap：

赋值运算符将其左边容器中的全部元素替换为右边容器中元素的拷贝。

c1 = {a,b,c};//是整体替换，如果原来c1中元素超过3个，则替换成3个

array<int,10> a1 = {0,1,2,3,4,5,6,7,8,9};

array<int,10> a2 = {0};

a1 = a2;

a2 = {0};//错误，不能将一个花括号列表赋予array数组

赋值相关运算会导致指向左边容器内部的迭代器、引用和指针失效。而swap操作将容器内容交换不会导致指向容器的迭代器、引用、指针失效（容器类型为array和string的情况除外）。

赋值运算符要求左边和右边的运算对象具有相同的类型。assign允许从一个不同但相容的类型赋值，或者从容器的一个子序列赋值。由于就元素被替换，因此传递给assign的迭代器不能指向调用assign的容器。

swap操作交换两个相同类型容器的内容。除array、string外，交换两个容器内容的操作保证会很快——元素本身并未交换，swap只是交换了两个容器的内部数据，因此可以保证在常数时间内完成。元素不会被移动意味着：指向容器的迭代器、引用和指针在操作后不会失效，仍指向操作之前所指向的那些元素，但是在swap后这些元素已经属于不同的容器了。

## 2.6容器大小操作：

## 2.7关系运算符：

关系运算符左右两边的运算对象必须是相同类型的容器，且必须保存相同类型的元素。判断大小和前面提到的判断string类型的判断方式一样。

只有当容器中元素类型也定义了相应的比较运算符时，我们才可以使用关系运算符来比较两个容器。

vector<Sale\_data> A,B;

if(A < B)//看Sale\_data类型有没有<运算符

# 3.顺序容器操作：

## 1.添加元素：

除array外，所有的标准库容器都提供灵活的内存管理：

①这些操作会改变容器的大小，array不支持这些操作，

②forward\_list有自己专有版本的insert和emplace

③forward\_list不支持push\_back和emplace\_back

④vector和string不支持push\_front和emplace\_front

|  |  |
| --- | --- |
| c.push\_back(t); | 在c尾部创建一个值为t或由args创建的元素，size自动增加，返回void（array、forward\_list不支持） |
| c.emplace\_back(args); |
| c.push\_front(t); | 在c首部创建一个值为t或由args创建的元素，size自动增加，返回void（vector、array、string不支持） |
| c.emplace\_front(args); |
| c.insert(p,t); | 在迭代器p指向的元素之前创建一个值为t或者由args创建的元素，size自动增加，返回指向新添加的元素的迭代器（无限制） |
| c.emplace(p,args); |
| c.insert(p,n,t); | 在迭代器p指向的元素之前创建n个值为t的元素，size自动增加，返回指向新添加的第一个元素的迭代器，若n为0，返回p |
| c.insert(p,b,e); | 将迭代器b和e指定的范围内的元素插入到迭代器p指向的元素之前。size自动增加，b和e不能指向c中的元素，返回指向新添加的第一个元素的迭代器，若范围为空，则返回p |
| c.insert(p,il); | il是一个花括号包围的元素值列表，将这些给定值插入到迭代器p指向的元素之前，size自动增加，返回新添加的第一个元素的迭代器，若列表为空，则返回p |

通过使用insert的返回值，可以在容器中一个特定位置反复插入元素（巧妙）：

list<string> lst;

auto iter = lst.begin();

while(cin>>word)

liter = lst.insert(iter,word);//等价于调用push\_front

对于emplace、emplace\_front、emplace\_back，这些操作是构造而不是拷贝元素。当调用push或insert成员函数时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器中；当调用emplace成员函数时，则是将参数传递给元素类型的构造函数，emplace成员使用这些参数在容器管理的内存空间中直接构造元素。

c.emplace\_back(“978-0590353403”,25,15.99);//正确，使用三个参数的构造函数

c.push\_back(“978-0590353403”,25,15.99);//错误

c.push\_back(Sales\_data(“978-0590353403”,25,15.99));//正确，创建一个临时的Sales\_data对象传递给push\_back

//调用emplace\_back时会在容器管理的内存空间中直接创建对象；调用push\_back会创建一个局部临时对象，并将其压入容器。

emplace函数的参数根据元素类型而变化，参数必须与元素类型的构造函数相匹配：

c.emplace\_back();//使用默认构造函数

c.emplace(iter,”999-99999999”);//使用Sales\_data(string)

c.emplace\_front(“978-0590353403”,25,15,99);//使用对应的构造函数

## 3.2访问元素：

|  |  |
| --- | --- |
| c.back(); | 返回c中尾元素的引用，若c为空，函数行为未定义（forward\_list不支持） |
| c.front(); | 返回c中首元素的引用，若c为空，函数行为未定义 |
| c[n] | 返回c中下表为n的元素的引用，n是一个无符号证书，若n >= c.size()，则函数行为未定义（list、forward\_list不支持） |
| c.at(n); | 返回下标为n的元素的引用，如果下标越界，则抛出一out\_of\_range异常（list、forward\_list不支持） |

对于一个空容器调用front和back，就像使用一个越界的下标一样，是一种严重的程序设计错误。调用以上几个元素返回的都是引用。如果容器是一个const对象，则返回值是const引用；如果容器不是const的，则返回值是普通引用。只有at()函数能检测输入的下标是否越界。

## 3.3删除元素：

|  |  |
| --- | --- |
| c.erase(p) | 删除迭代器 p 所指向的元素。返回一个迭代器，它指向被删除元素后面的元素。如果 p 指向容器内的最后一个元素，则返回的迭代器指向容器的超出末端的下一位置。如果 p 本身就是指向超出末端的下一位置的迭代器，则该函数未定义（array不支持） |
| c.erase(b,e) | 删除迭代器 b 和 e 所标记的范围内所有的元素。返回一个迭代器，它指向被删除元素段后面的元素。如果 e 本身就是指向超出末端的下一位置的迭代器，则返回的迭代器也指向容器的超出末端的下一位置（array不支持） |
| c.clear() | 删除容器 c 内的所有元素。返回 void（array不支持） |
| c.pop\_back() | 删除容器 c 的最后一个元素。返回 void。如果 c 为空容器，则该函数未定义（array、forward\_list不支持） |
| c.pop\_front() | 删除容器 c 的第一个元素。返回 void。如果 c 为空容器，则该函数未定义。只适用于 list 或 deque 容器（array、vector、string不支持） |

forward\_list有特殊版本的erase。删除deque中除首位置之外的任何元素都会使所有迭代器、引用、指针失效。指向vector或string中删除点之后位置的迭代器、引用、指针都会实现。

删除元素的成员函数并不检查其函数，在删除元素之前，程序员必须确保它是存在的，先判断容器是否为空。

## 3.4特殊的forward\_list操作：

为了添加或产出一个元素，我们需要访问其前驱，以便改变前驱的链接。在一个单向链表中，没有简单的方法来获得一个元素的前驱，所以在一个forward\_list中删除添加元素的操作是通过改变给定元素之后的元素来完成的。forward\_list未定义insert、emplace、erase。当在forward\_list中添加或删除元素时，我们必须关注两个迭代器——要处理的元素及其前驱。

|  |  |
| --- | --- |
| lst.before\_begin(); | 返回指向链表首元素之前不存在的元素的迭代器，此迭代器不能解引用 |
| lst.cbefore\_begin(); |
| lst.insert\_after(p,t); | 在迭代器p之后的位置插入元素，t是一个对象，n是数量，b和e是表示范围的一对迭代器（b和e不能指向lst内），il是一个花括号列表。返回一个指向最后一个插入元素的迭代器。如果范围为空，则返回p。若p为尾后迭代器，则函数行为未定义。 |
| lst.insert\_after(p,n,t); |
| lst.insert\_after(p,b,e); |
| lst.insert\_after(p,il); |
| emplace\_after(p,args); | 使用args在p指定位置之后创建一个元素。返回一个指向这个新元素的迭代器。若p为尾后迭代器，则函数行为未定义 |
| lst.erase\_after(p); | 删除p指向的位置之后的元素，或删除从b（不包含）之后直到e之间的元素。返回一个指向被删元素之后元素的迭代器，若不存在这样的元素，则返回尾后。如果p指向lst的尾元素或者是一个尾后迭代器，则函数行为未定义。 |
| lst.erase\_after(b,e); |

forward\_list<int> a = {0,1,2,3,4,5,6,7,8,9};

auto curr = a.begin();

auto prev = a.before\_begin();

while(curr != a.end())

{

if((\*curr) %2)

curr = a.erase\_after(prev);

else

{

prev = curr;

++curr;

}

}

## 3.5改变容器大小：

|  |  |
| --- | --- |
| c.resize(n) | 调整容器 c 的长度大小，使其能容纳 n 个元素，如果 n < c.size()，则删除多出来的元素；否则，向尾部添加采用值初始化的新元素。若必须添加新元素，对新元素进行值初始化（array不支持）。 |
| c.resize(n,t) | 调整容器 c 的长度大小，使其能容纳 n 个元素。所有新添加的元素值都为 t（array不支持） |

如果resize缩小容器，则指向被删除元素的迭代器、引用、指针都会失效；对vector、string、deque进行resize可能导致迭代器、指针、引用失效。

## 3.6容器操作可能使迭代器失效：

向容器中添加元素可能会导致迭代器失效：

①对于vector和string，若存储空间被重新分配，则指向容器的迭代器、指针和引用会失效；若存储空间未重新分配，指向插入位置之前的元素的迭代器、指针和引用仍有效。但指向插入位置之后元素的迭代器、指针、引用将会失效

②对于deque，插入到除首尾位置之外的任何位置都会导致迭代器、指针和引用失效。如果在首尾位置添加元素，迭代器会失效，但指向存在的元素的引用和指针不会失效

③对于list和forward\_list，指向容器的迭代器（包括尾后迭代器和首前迭代器），指针和引用扔有效

当我们删除元素时，尾后迭代器总会失效。向容器中删除元素可能会导致迭代器失效：

①对于list和forward\_list，指向容器其他位置的迭代器（包括尾后和首前）、引用指针仍有效

②对于deque，如果在首尾之外的任何位置删除元素，那么指向被删除元素外其他元素的迭代器、引用或指针也会失效。如果是删除deque的尾元素，则尾后迭代器也会失效，但其他迭代器、引用和指针不受影响：如果删除首元素，这些也不会受影响

③对于vector和string，指向被删除元素之前元素的迭代器、引用和指针仍然有效。

注意:当我们删除元素时，尾后迭代器总会失效。当使用迭代器时，最小化要求迭代器必须保持有效的程序片段是一个好的方法。因此必须保证每次改变容器的操作之后都正确地重新定位迭代器。通常不要保存end返回的迭代器，实时调用即可。

用来删除偶数元素，复制每个奇数元素：

list<int> vi = {0,1,2,3,4,5,6,7,8,9};

auto iter = vi.begin();

while (iter != vi.end()) {

if (\*iter % 2) {

iter = vi.insert(iter, \*iter);

advance(iter, 2);

} else

iter = vi.erase(iter);

}

# 4.vector对象是如何增长的：

虽然vector在每次重新分配内存空间都要移动所有元素，但使用此策略后，其扩张操作通常比list和deque还要快。

只有在执行insert操作时size与capacity相等，或者调用resize或reserve时给定的大小超过当前capacity，vector才可能重新分配内存空间。

|  |  |
| --- | --- |
| c.shrink\_to\_fit(); | 请将capacity()减少为与size()相同的大小（list、forward\_list、array不支持） |
| c.capacity(); | 不重新分配内存空间的话，c可以保存多少元素（只适用vector和string） |
| c.reserve(n); | 分配至少能容纳n个元素的空间，并不改变容器中元素的数量，仅影响vector预先分配多大的内存空间。如果需求大小小于等于当前容量，reserve什么都不做；如果需求大小大于当前容量，reserve至少分配与需求一样大的空间（可能更大）（只适用vector和string） |

# 5.额外的string操作：

## 5.1构造string的其他方法：

|  |  |
| --- | --- |
| string s(cp,n); | s是cp指向的数组中前n个字符的拷贝，此数组至少应该包含n个字符 |
| string s(s2,pos2); | s是string s2从下标pos2开始的字符拷贝。若pos2>s2.size()，构造函数的行为未定义 |
| string s(s2,pos2,len2); | s是string s2从下标pos2开始len2个字符的拷贝。若pos2>s2.size()，抛出out\_of\_range异常，构造函数的行为未定义。不管len2的值是多少，构造函数至多拷贝s2.size() – pos2个字符 |

s2是string或const char\*参数。如果从一个const char\*创建string时，指针指向的数组必须以空字符串结尾，拷贝操作遇到空字符时停止，如果我们还传递给构造函数一个计数值，数组就不必以空字符结尾。如果我们未传递计数值且数组也未以空字符结尾，或者给定计数值大于数组大小，则构造函数的行为是未定义的。

## 5.2子字符串操作：

|  |  |
| --- | --- |
| s.substr(pos,n); | 返回一个string，包含s中从pos开始的n个字符的拷贝。pos的默认值为0。n的默认值为s.size() – pos，即拷贝从pos开始的所有字符。 |

如果开始位置超过了string的大小，则substr函数抛出一个out\_of\_range异常。

## 5.3改变string的其他方法：

除了公有的insert版本外，string还定义了额外的insert和erase版本（下标版本）：

s.insert(s.size(),5,’!’);//在s末尾插入5个感叹号

s.erase(s.size() – 5,5);//从s删除最后五个字符

string还提供了接受C风格字符数组的insert和assign版本：

const char \*cp = “Stately,asdasdas”,cp2 = “some string”;

s.assign(cp,7);// s = “Stately”，拷贝从cp开始的7个字符要求赋值的字符数必须小于等于cp的字符数

s.insert(s.size(),cp + 7);//c = “Stately,asdasdas”，把字符插入到s.size()（不存在的）处之前的位置。

s.insert(0,s2);//在s位置0之前插入s2的拷贝

s.insert(0,s2,0,s2.size());//在s位置0之前插入s2中s2[0]开始的s2.size()个字符

string还定义了两个额外的成员函数：append和replace，这两个函数可以改变string的内容。

|  |  |
| --- | --- |
| s.insert(pos,args); | 在pos之前插入args指定的字符，pos可以是一个下标或一个迭代器。接受下标的版本返回一个指向s的引用；接受迭代器的版本返回指向第一个插入字符的迭代器。 |
| s.erase(pos,len); | 删除从位置pos开始的len个字符，如果len被省略，则删除从pos开始直至s结尾的所有字符。返回一个s的引用。 |
| s.append( ***args***) | 将 args 串接在 s 后面。返回 s 的引用 |
| s.replace(range, ***args***) | 删除 s 中从范围range内的字符，用 args 指定的字符替换之，可以替换为长度不一样的字符串。range或者是一个下表和一个长度，或者是一对指向s的迭代器。返回一个s的引用 |

args可以是下列形式之一：

①str：字符串str

②str,pos,len：str中从pos开始最多len个字符

③cp,len：从cp指向的字符数组的（最多）前len个字符

④cp：cp指向的以空字符结尾的字符数组

⑤n,c：n个字符c

⑥b,e：迭代器b和e指定的范围内的字符

⑦初始化列表：

append和assign可以使用以上7个形式，replace和insert可以使用的形式依赖于range和pos是如何指定的。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| replace | replace | insert | insert | args可以是 |
| (pos,len,args) | (b,e,args) | (pos,args) | (iter,args) |  |
| 是 | 是 | 是 | 否 | str |
| 是 | 否 | 是 | 否 | str,pos,len |
| 是 | 是 | 是 | 否 | cp,len |
| 是 | 是 | 否 | 否 | cp |
| 是 | 是 | 是 | 是 | n,c |
| 否 | 是 | 否 | 是 | b2,e2 |
| 否 | 是 | 否 | 是 | 初始化列表 |

assign和append函数无需指定要替换string中哪个部分：asign总是替换string中所有内容，append总是将新字符追加到string末尾。 +

## 5.3string搜索操作：

string numbera(“0123456789”),name(“r2d2”);

auto pos = name.find\_first\_of(numbers);//定位name中的第一个数字

string dept(“03714p3”);

auto pos = dept.find\_first\_not\_of(numbers);//搜索dept中第一个非数字字符

|  |  |
| --- | --- |
| s.find(args); | 查找s中args第一次出现的位置 |
| s.rfind(args); | 查找s中args最后出现的位置 |
| s.find\_first\_of(args); | 在s中查找args中任何一个字符第一次出现的位置 |
| s.find\_last\_of(args); | 在s中查找args中任何一个字符最后出现的位置 |
| s.find\_first\_not\_of(args); | 在s中查找第一个不在args中的字符 |
| s.find\_last\_not\_of(args); | 在s中查找最后不在args中的字符 |

args必须是以下形式之一：

c,pos：从s中位置pos开始查找字符c，pos默认为0

s2,pos：从s中位置pos开始查找字符串s2，pos默认为0

cp,pos：从s中位置pos开始查找指针cp指向的以空字符结尾的C风格字符串，pos默认为0

cp,pos,n：从s中位置pos开始查找指针cp指向的数组的前n个字符，pos和n无默认值。

rfind、find\_last\_not\_of、find\_last\_of是逆向搜索。

用可选参数在字符串中循环地搜索子字符串出现的所有位置：

string::size\_type pos = 0;

while((pos = name.find\_first\_of(numbers,pos)) != string::npos)

{

cout<<”found number at index:”<<pos<<”element is”<<name[pos]<<endl;

++pos;

}

## 5.4compare函数：

|  |  |
| --- | --- |
| s.compare(s2); | 比较s和s2 |
| s.compare(pos1,n,s2); | 将s中从pos1开始的n个字符与s2比较 |
| s.compare(pos1,n1,s2,pos,n2); | 将s中从pos1开始的n1个字符与s2中从pos2开始的n个字符比较 |
| s.compare(cp); | 比较s与cp指向的以空字符结尾的字符数组 |
| s.compare(pos1,n,cp); | 将s中从pos1开始的n个字符与cp指向的 以空字符结尾的字符数组 |
| s.compare(pos1,n1,cp,n2); | 将s中从pos1开始的n个字符与cp指向的地址开始的n2个字符比较 |

## 5.5数值转换：

string参数中的一个非空白符必须是符号或数字。它可以以0x或0X开头来表示十进制数。对那些将字符串转换为浮点值的函数，string参数也可以以小数点开头，并可以包含e或E来表示指数部分。对于那些将字符串转换为整型值的函数，根据基数不同，string参数可以包含字母字符，对应大于数字9的数。

如果string不能转换为一个数值，这些函数抛出一个invalid\_argument异常。如果转换得到的数值无法用任何人类型来表示，则抛出一个out\_of\_range异常。

|  |  |
| --- | --- |
| to\_string(val); | 一组重载函数，返回值val的string表示。val可以是任何算数类型，与往常一样，小整型会被提升。 |
| stoi(s,p,b); | 返回s的起始子串（表示整数内容）的数值，返回值类型分别是int、long、unsigned long、unsigned long long 、long long。b表示转换所用的基数，默认值为10。p是size\_t指针，用来保存s中第一个非数值字符的下标，p默认为0，即：函数不保存下标 |
| stol(s,p,b); |
| stoul(s,p,b); |
| stoll(s,p,b); |
| stoull (s,p,b); |
| stof(s,p); | 返回s的起始子串（表示浮点数内容）的数值，返回值类型分别是float、double、long double。参数p的作用同上。 |
| stod(s,p); |
| stold(s,p); |

# 6.容器适配器：

除了顺序容器外，标准库还定义了三个顺序容器适配器：stack（基于deque实现，还可以list、deque、vector）、queue（基于deque实现，还可以list）和priority\_queue（基于vector实现，还可以deque）。容器、迭代器、函数都有适配器。一个适配器能使某种事物的行为看起来像另外一种事物一样。一个容器适配器接受一种已有的容器类型，使其行为看起来像一种不同的类型。我们只可以使用适配器操作，不能使用底层容器进行操作。queue和priority\_queue定义在queue头文件中。

|  |  |
| --- | --- |
| 所有容器适配器都支持的操作和类型 | |
| size\_type | 一种类型，足以保存当前类型的最大对象的大小 |
| value\_type | 元素类型 |
| container\_type | 实现适配器的底层容器类型 |
| A a; | 创建一个名为a的空容器 |
| A a(c); | 创建一个名为a的适配器，带有c的一个拷贝 |
| 关系运算符==、!=、<、<=、>、>= | 这些运算符返回底层容器的比较结果 |
| a.empty(); | 是否为空 |
| a.size(); | 元素数目 |
| swap(a,b); | 交换，a和b必须有相同的类型，包含底层容器类型也必须相同 |
| a.swap(b); |

deque<int> deq;

stack<int> stk(deq);//用deq初始化stk

在创建一个适配器时将一个命名的顺序容器作为第二个类型参数，来重载默认容器类型：

stack<string,vector<string>> str\_stk;//在vector上实现的空栈。

①适配器不能构造在array之上：所有适配器都要求容器具有添加和删除元素能力。

②适配器不能构造在forward\_list之上：所有适配器都要求容器具有添加、删除以及访问尾部元素的能力。

|  |  |
| --- | --- |
| 未列出的栈操作 | |
| s.pop(); | 删除栈顶元素，但不返回该元素值 |
| s.push(item) | 创建一个新元素压入栈顶，该元素通过拷贝或移动item而来，或者由args构造 |
| s.emplace(args); |
| s.top(); | 返回栈顶元素 |

|  |  |
| --- | --- |
| 未列出的queue和priority\_queue操作 | |
| q.pop(); | 返回queue的首元素或priority\_queue的最高优先级的元素，但不删除此元素 |
| q.front(); | 返回首元素，但不删除此元素 |
| q.back(); | 返回尾元素，但不删除此元素，只适用于queue |
| q.top(); | 返回最高优先级元素，但不删除该元素，只是用于priority\_queue |
| q.push(item); | 在queue末尾或priority\_queue中恰当的位置创建一个元素，值为item，或者由args构造 |
| q.emplace(args); |

priority\_queue为队列中的元素建立优先级，新加入的元素会排在所有优先级比它低的已有元素之前，默认使用<运算符来确定相对优先级。