# 第九章 顺序容器

> 第三章内容的扩展

---

- 容器就是一些特定类型对象的集合
- **所有容器类都共享公共的接口，不同容器按不同方式对其进行扩展、**
- 标准库提供了三种容器适配器，分别为容器操作定义不同的接口，来与容器适配

## 顺序容器概述

**·标准库中顺序容器类型**

| 类型 | 特性 |
| --- | --- |
| 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或删除元素可能很慢 |
| deque | 双端队列。支持快速随机访问。在头尾插入/删除速度很快 |
| list | 双向链表。只支持双向顺序访问。在list中任何位置进行插入/删除都很快 |
| forward_list | 单向链表。只支持单项顺序访问。链表的插入/删除都很快 |
| array | 固定大小数组。支持快速随机访问。 **不能添加或删除空间** |
| string | 与vector相似的容器，但专门用于保存字符。随机访问快。在尾部插入/删除速度快 |

- 所有顺序容器都提供了 **快速访问元素** 的能力

> 新标准库容器的性能几乎肯定与最精心优化过的同类数据结构一样好（通常会更好）
>
> 现代C++程序应该使用标准库容器，而不是原始的数据结构，如内置数组

**选择容器的基本原则**

- **通常，使用vector是最好的选择，除非你有很好的理由选择其他容器**
- 如果程序有很多小元素，且空间的额外开销很重要，则不要使用list和forwar_list
- 如果程序需要随机访问元素，应使用vector或deque

- 如果程序只有在**读取输入**的时候才需要在容器中间位置插入元素，随后需要随机访问元素，则在输入阶段使用list,

  一旦输入完成，将list的内容拷贝到vector中

## 容器库概述

> 容器类型上的操作分成三类：
>
> - 某些操作是所有容器类型都提供的；
> - 另外一些操作仅针对顺序容器、关联容器或无序容器
> - 还有一些操作只适用一小部分容器
>
> 本节介绍所有容器都是适用的操作和顺序容器的特有操作

- 一般来说，每个容器都定义在一个头文件中，文件名和类型名相同
- 容器均定义为模板类[1]
- 顺序容器几乎可以保存任意类型的元素，当某些容器操作对元素类型有要求

```
1 //noDefault 是是一个没有默认构造函数的类
2 vector<noDefault> v1(10);       // 编译不通过
3 vector<noDefault> v1(10,init);  // init是该类的一个实例化，正确
```

---

| 类型别名 | 含义 |
| --- | --- |
| iterator | 此容器类型的迭代器类型 |
| const_iterator | 可以读取元素，但不能修改元素的迭代器类型 |
| size_type | 无符号整数类型，足够保存此种容器类型最大可能容器的大小 |
| 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 | 容器类保存的元素类型 |
| reference | 元素的左值类型；与value_type& 含义相同 |
| const_reference | 元素const左值类型（即，const value_type &） |

> 类型别名使用方式

```
1 vector<int>::iterator p1;
```

---

| 构造函数 | 含义 |
| --- | --- |
| C c; | 默认构造函数，构造容器.<br>(array默认包含与其大小一样大多的元素并进行默认初始化，其他容器默认为空) |
| C c1(c2);<br>C c1=c2; | 构造c2的拷贝c1<br>（c1和c2必须是**相同容器类型**，保存相同的元素类型；对array类型，两者大小还要相同） |
| c c(b,e); | 构造c，将迭代器b和e指定范围[b,e)内的元素拷贝到c<br>(范围内的元素类型必须与C的元素类型相容，**array不支持**） |
| C c{a, b, c.....};<br>C c={a, b, c}; | 列表初始化c<br>（列表元素必须和c的元素类型相容）<br>（对于array，列表元素必须**等于或小于**array的大小，任何遗漏的元素都进行值初始化[2] |
| **只有顺序容器（除array）的构造函数才能接受大小参数** | |
| C seq(n); | seq包含n个元素，进行值初始化；此构造函数是`explicit`的（string不适用） |
| C seq(n,t); | seq包含n个初始化为值t的值 |

> 注意"相容"而不是"相同"

- 用迭代器初始化，可以不要求容器类型相同

```
list<string> author = {"syy","oyjw"};

list<string> author2(author);
vector<string> author3(author);      // 错误，容器类型不匹配
vector<string> author4(author.begin(),author.end());
```

---

| 赋值与Swap | 含义 |
| --- | --- |
| c1=c2 | （c1, c2必须有完全相同的类型） |
| c1={a,b,.....} | 将c1中的元素替换为列表元素(**array不适用**) |
| a.swap(b) | 交换a,b的元素<br>（a, b必须具有完全相同的类型） |
| swap(a,b) | |
| `assign`**操作不适用于关联容器和array** | |
| seq.assign(b,e) | 将seq中的元素替换成[b,e)所表示范围中的元素<br>（迭代器不能指向seq中的元素） |
| seq.assign(il) | 将seq中的元素替换为初始化列表的il的元素 |
| seq.assign(n,t) | 将seq中的元素替换为n个值为t的元素 |

- 赋值运算符将左边容器中的 **全部元素** 替换成右边容器元素的拷贝
- 赋值运算会导致指向左边容器内部的迭代器、引用和指针失效

  > 失效是指 指向的内存区域变化

  ```
  vector<int> a = {1, 2};
  vector<int> b = {3, 4};
  auto p1 = a.begin()+1;
  a=b;
  cout<<*p1;  // 4     没有失效,仍然指向a

  swap(a,b);
  cout<<*p1;  //2, 没有失效 因为原先指向a的内存现在被交换到b，所有继续指向该内存，也就保证值不变

  ```

  swap操作不会失效

  > - 赋值会重新分配内存空间；交换只是值的变化
  > - `string` 除外，swap也会导致失效

- `assign` 不要求完全相同，只要元素基本类型相容即可

  > 其实比较像初始化的操作，只是可以单独拿出来用了

- `swap`：元素本身并未交换，只是交换了两个容器的内部数据结构 **？？**

  > 不会对任何元素进行拷贝、删除或插入操作，保证在常数时间完成

- **建议：**统一使用非成员函数版本的swap

  > 非成员版本在泛型编程中非常重要

---

| **大小** | **含义** |
| --- | --- |
| c.size() | （**不支持forward_list** ） |
| c.max_size() | 该类型容器可保存的最大元素数目 |
| c.empty() | |

```
vector<int> p;
cout<<p.max_size();          // 输出4611686018427387903
```

---

| **添加/删除元素（不支持array）** | **含义** |
| --- | --- |
| c.insert(args) | 将args中的元素拷贝进c |
| c.emplace(inits) | 使用inits构造一个c中的指定元素 |
| c.erase(args) | 删除args指定的元素 |
| c.clear() | |

> 在不同容器中，这些操作括号内参数都不同，具体使用在分析
>
> args代表的是参数列表，不一定就是一个参数

---

| **关系运算符** | **含义** |
| --- | --- |
| ==，!= | **所有容器都支持相等(不相等)运算符** |
| <, <=, >, >= | 关系运算符 **（无序关联容器不支持）** |

- 容器完全相同（容器类型和基本元素类型）
- 比较的实质是进行容器内元素的逐对比较

  > - 类似string的比较

> - **只有当容器内元素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**

---

| 获取迭代器 | 含义 |
| --- | --- |
| c.begin(), c.end() | |
| c.cbegin(), c.cend() | 返回const_iterator |

> cbegin()、cend()是重载过的函数（**深入**）。
>
> 实际上有两个名为begin的成员。一个是`const成员`[3]，一个是非常量成员

---

| 反向迭代器的额外成员（不支持forward_list） | 含义 |
| --- | --- |
| reverse_iterator | 按**逆序寻址**元素的迭代器类型 |
| const_reverse_iterator | 不能修改元素的逆序迭代器 |
| c.rbegin(), c.rend() | 返回指向c的尾元素和首元素之前位置迭代器 |
| c.crbegin(), c.rcend() | |

> 反向迭代器是一种反向遍历容器的迭代器；
>
> 与正向迭代器相比，各种操作的含义都发生类颠倒
>
> - 对一个反向迭代器进行`++`，会得到上一个元素
>
> （第10章扩展）

- 当不需要写访问时，应使用cbegin()、cend()

## 迭代器

- 所有迭代器具有公共的接口：如果一个迭代器提供某一个操作，那么所有提供相同操作的迭代器对这个操作的实现方式都是相同的。

| 迭代器支持的操作 | 含义 |
| --- | --- |
| *iter | 返回iter所指元素的**引用** |
| iter->mem | |
| ++iter | |
| --iter | forward_list的迭代器不支持该操作 |
| iter1 == iter2 | |
| iter != iter2 | |

---

| 迭代器（`string、vector、array、deque`）支持的运算 | 含义 |
| --- | --- |
| iter + n<br>iter - n | 仍然返回一个迭代器，向前/后移动n个元素（以元素为基本距离） |
| iter += n<br>iter -= n | |
| iter1 - iter2 | 返回两个迭代器之间的距离（中间元素的个数） |
| >、>=、<、<= | 比较指向位置的前后 |

> 以上迭代器运算的容器是连续存储的

---

- 迭代器范围：标准库的基础
  - **左闭右合区间**
    - 如果`begin=end`，则范围为空
    - 如果`begin!=end`，范围内至少一个元素

```
while(begin != end){
    ....;
    ++begin;
}
```

## array

与内置数组一样，标准库array的大小也是类型的一部分。

```
array(string,10) p;
```

> 大小必须是常量类型。
>
> ```
> int a=1;
> array(int,a) p;
> ```

- 由于大小是array的类型的一部分，array不支持普通的容器构造函数

  > 这些构造函数都会确定容器的大小，要么隐式的，要么显式的

- 与内置数组不允许进行拷贝或对象赋值操作不同，array并无限制

  ```
  int diags[10] = {1, 2, 3};
  int diags2[10] = diags;     // 错误

  array<int,10> p1 = {1, 2, 3};
  array<int,10> p2 = p1;
  array<int,5> p3 = p1;       // 错误
  ```

  - 要求类型和大小相同

- array的 `swap` 会真正交换它们的元素

  > 所需时间和元素的数目成正比

  ```
  array<int , 2> c1={1,2};
  array<int, 2> c2={3,4};
  auto  p2 = c1.begin();
  swap(c1,c2);
  cout<<*p2;      // 输出3
  ```

  > 注意和上面vector例子的不同
  >
  > 因为，内存没变，所有p2仍然指向c1的内存，输出3

# 顺序容器操作（重点）

## 添加元素（非array）

这些操作都会改变容器大小，*array*不支持

| 操作 | 含义 |
| --- | --- |
| c.push_back(t)<br>c.emplace_back(args) | 尾部插入 |
| c.push_front(t)<br>c.emplace_front(args) | 头部插入 |
| c.insert(p,t)<br>c.emplace(p,args) | 在迭代器p指向元素**之前**插入。返回指向新添加元素位置的迭代器 |
| c.insert(p,n,t) | 在迭代器p指向元素之前插入n个值为t的元素。返回指向新添加的第一个元素位置的迭代器 |
| c.insert(p,b,e) | 将迭代器b和e指定范围内的元素插入迭代器p指向的元素之前。返回新添加的第一个元素位置的迭代器<br>（b,e不能指向c中的元素） |
| c.insert(p,il) | il是一个花括号包围的元素值列表。将这些值插入到迭代器p之前的位置。返回新添加的第一个元素位置的迭代器 |

- `forward_list` 有自己**专有版本**的 `insert` 和 `emplace`；

  `forward_list` 不支持 `push_back` 和 `emplace_back`；
- `vector` 和 `string` 不支持 `push_front` 和 `emplace_front`

  > 可用insert实现头插
- **向vector、string或deque插入元素会使所有指向容器的迭代器、引用和指针失效**

  > vector在push_back的时候，当容量不足时会触发扩容，导致整个vector**重新申请内存**，并且**将原有的数据复制到新的内存中，并将原有内存释放**。
  >
  > 所以，
  >
  > - 如果容量充足，不会失效
  > - 如果容量不足，会失效
  > - 如果在中间插入，插入前的不会失效，插入后的因为之前的值的存储内存发生变化，所以也会失效
- 通过使用 `insert` 可以在容器中一个特定位置反复插入元素
- `C++11`：emplace使用参数在容器管理的内存空间中直接**构造元素**(先使用构造函数创建对象)

```
c.emplace("111",23,15.99);       // 调用类的构造函数创建元素
c.push_back("111",23,15,9);      // 错误
c.push_bakc(Sale_data("111",23,15.99));
```

> 传递给emplace函数的参数必须与元素类型的构造函数相匹配

## 访问元素

| 操作 | 含义 |
| --- | --- |
| c.back() | 返回c中尾元素的**引用** |
| c.front() | 返回c中首元素的引用 |
| c[n] | 返回c中下标为n的元素的引用 |
| c.at(n) | 返回下标为n的元素的引用。如果下标越界，会抛出`out_of_range`异常 |

- 如果容器中没有元素，访问的操作是未定义的(会报错)

```
vector<int> a;
cint>>a[0];      // 错误，c[0]是未定义的
```

- 包括`array`在内的每个顺序容器都有一个`front`函数；

  除`forward_list`之外的所有顺序容器都有一个`back`函数；
- `at`和下标操作只适用于string、vector、deque和array

> 都是使用连续的地址存放元素

## 删除元素

删除元素操作不适用于array

| 操作 | 含义 |
| --- | --- |
| c.pop_back() | 删除尾元素。返回void |
| c.pop_front() | 删除首元素。返回void |
| c.erase(p) | 删除迭代器p所指定的元素，返回一个指向被删除元素之后元素的迭代器 |
| c.erase(b,e) | 删除迭代器b和e指定范围的元素[b,e)。返回e |
| c.clear() | 删除所有元素。返回void |

- 删除 `deque` 中除首尾之外的任何元素都会使所有迭代器、引用和指针失效；
  **理解一下失效的意思**

  指向 `vector` 或 `string` 中删除点之后位置的迭代器、引用和指针会失效
- 删除前，必须保证该位置是存在的
- `vector` 和 `string` 不支持 `pop_front` ；

  `forward_list` 不支持 `pop_back` ，有其特殊版本的 `erase()`

## 特殊的forward_list操作

因为删除链表，要求修改指针的指向，需要访问前驱。但是单向链表获得前驱非常麻烦，所以添加或删除操作实际上是**通过改变给定元素之后的元素来完成的**

| 操作 | 含义 |
| --- | --- |
| lst.before_begin()<br>lst.cbefor_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.insert_after(p,t) | 在迭代器p**之后**插入元素t。返回指向最后一个插入元素的迭代器。 |
| lst.insert_after(p,n,t) | 在迭代器p**之后**插入n个元素t。返回指向最后一个插入元素的迭代器。 |
| lst.insert_after(p,b,e) | 在迭代器p**之后**插入范围在[b,e)的所有元素。返回指向最后一个插入元素的迭代器。 |
| lst.insert_after(p,il) | 在迭代器p**之后**插入列表所有元素。返回指向最后一个插入元素的迭代器。 |
| emplace_after(p,args) | 使用args在p指定的位置之后创建一个元素。返回指向新元素的迭代器 |
| lst.erase_after(p) | 删除p指向的为止之后的元素。返回指向被删除元素之后的迭代器 |
| lst.erase_after(b,e) | 删除[b,e)之间的元素。返回指向被删除元素之后的迭代器 |

- p不能是尾后迭代器

  `lst.before_begin()` 返回首前迭代器
- 当在 `forward_list` 中添加或删除元素时，必须关注两个迭代器——一个指向当前正在处理的元素，另一个指向其前驱

## 改变容器大小

array不支持 `resize()`

| 操作 | 含义 |
| --- | --- |
| c.resize(n) | 调整c的大小为n个元素 |
| c.resize(n,t) | 调整c的大小为n个元素，并且都初始化为t |

- 如果n<size()，则容器多出的元素会被删除；
- 如果缩小容器大小，则指向被删除元素的迭代器、引用和指针都会失效

  对 `vector、string、deque` 进行resize **可能** 导致迭代器、指针和引用失效

# 顺序容器的一些特性

## 迭代器失效问题

- 向容器添加或删除元素，可能会使指向容器元素的指针、引用或迭代器失效
- 一个失效的指针、引用或迭代器将不再表示任何元素 `？？`

**添加元素**

- 如果容器是 vector 或 string，且存储空间被重新分配，则指向容器的迭代器、指针和引用都会失效。如果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、指针和引用将会失效。
- 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。
- 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）、指针和引用仍有效。

**删除元素**

- 对于 list 和 forward_list，指向容器其他位置的迭代器（包括尾后迭代器和首前迭代器）、引用和指针仍有效。
- 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；如果是删除首元素，这些也不会受影响。
- 对于 vector 和 string，指向被删元素之前元素的迭代器、引用和指针仍有效。

  注意：当我们删除元素时，尾后迭代器总是会失效。

- 当添加/删除 `vector` 或 `string` 的元素后，或在 `deque` 中首元素之外任何位置添加/删除元素后，原来 `end` 返回的迭代器总是会失效

  > - 添加或删除元素的循环程序必须反复调用 `end()`，而不能在循环之前保存end返回的迭代器
  > - 失效后，原来保存的end迭代器将不在指向容器中任何元素，或是尾元素之后的位置

## vector/string对象是如何增长的

**动态增长：** vector在push_back的时候，当容量不足时会触发扩容，导致整个vector **重新申请内存**，并且**将原有的数据复制到新的内存中，并将原有内存释放。**

> 为了保证连续性

**扩容策略：** 通常会分配比新的空间需求更大的内存空间。容器预留这些空间作为备用，可用来保存更多的元素，避免每次添加新的元素都重新分配容器内存空间了

> 标准库的策略是：需要重新分配内存空间时将当前的容量翻倍

| 操作 | 含义 |
| --- | --- |
| c.shrink_ to_fit() | 请求容器容量减少为size()的相同大小 |
| c.capacity() | 获取容器容量 |
| c.reserver(n) | 给容器分配n个容量 |

- `shrink_to_fit()` 只适用于vector、string和deque;

  调用该函数只是一个请求，标准库并不一定保证退还内存 `？？`
- `capatity()、reserve()` 只适用于vector和string;
- 调用 `reserve()` 永远不会减少容器所占的内存大小;

**容量和大小**

- 容量是指容器所占的内存空间大小;
- 大小是容器目前所保存的元素的数目;
  - `resize()` 只改变容器内元素的数目，而不是容器容量

```cpp
vector<int> a{1,2,3,4};
cout<<a.capacity()<<endl;       // 4
a.push_back(5);
cout<<a.capacity()<<endl;       // 8
a.shrink_to_fit();
cout<<a.capacity()<<endl;       // 5
a.reserve(4);
cout<<a.capacity()<<endl;       // 5
```

# 额外的string操作（重点）

大部分提供是string类和C风格字符数组之间的相互转换，要么是增加了允许我们用下标代替迭代器的版本

## 其他构造方法

- 第3章介绍的方法
- 和其他顺序容器一致的方法
- | 操作 | 含义 |
  | --- | --- |
  | string s(cp, n) | s是cp指向的**字符数组**中前n个字符的拷贝<br>此数组至少应该包含n个字符，否则超过的部分是未定义的 |
  | string s(s2, pos2) | s是s2从**下标**pos2开始的字符的拷贝<br>pos2<s2.size() |
  | string s(s2, pos2, len2) | s是所从下标pos2开始的len2个字符的拷贝。<br>不管len2是多少，构造函数至多拷贝s2.size()-pos2个字符 |

  - ```cpp
    char a[]="hello world";
    string s(a,100);        // a中没有100个字符
    cout<<s;        // hello world    •?          ??      ?@
        =       恸@
    ```

    > 因为这个是从右向左的，所有a+99 内存区域是未定义的
    ```

- ```
  string a="hello world";
  string s(a,100);        // terminate called after throwing an instance of 'std::out_of_range'
  string s(a,1,100);
  cout<<s;        // ello world
  ```

  > 从左向右复制，要么读到限制的数量停止，要么读到`\0`自动停止，否则之后的操作就是未定义的

**substr()**：返回一个string的子串

```
string s("hello world");
string s2 = s.substr(0, 5); // [s[0],s[5])
string s3 = s.substr(6);    // [s[6],s.end())
string s4 = s.substr(14)    // 抛出out_of_range异常
```

## 修改string的其他方法

| 操作 | 含义 |
| --- | --- |
| s.insert(pos, args) | 在pos前插入args指定的字符<br>pos可以是下标或迭代器 |
| s.erase(pos, len) | 删除从位置pos开始的len个字符<br>如果len被省略，则删除pos到末尾的所有字符 |
| s.assign(args) | 将s中的**所有字符**替换为args指定的字符 |
| s.append(args) | 将args追加到s的末尾 |
| s.replace(range,args) | 删除s中范围range内的元素，替换为args指定的字符<br>range可以是下标和一个长度；或者是一对迭代器 |

- ```
  // s.insert(...)
  s.insert(s.size(), 5, '!');
  s.insert(s.size(), "abc");
  char *cp="abcd";
  s.insert(s.size(), cp+2);
  s.insert(0, s2, 0, s2.size());      // 在s[0]之前插入s2中s2[0]开始的s2.size()个字符
  ```

- ```
  // s.erase(...)
  s.erase(s.size()-5, 5);     // 删除末尾5个元素
  ```
  ```

- ```
  s.assign(cp,7);      // s中的内容替换为cp指向的地址开始的7个字符；要求赋值的字符数必须小于或等于cp指向的数组中的字符数
  s.assign(s2);
  // 还有多种参数形式，需要可查
  ```

- ```
  s.replace(11,3,"5th");  // 从位置11开始，删除3个，然后插入"5th"
  s.replace(s2.begin(), s2.begin()+2, "5th");
  ```

- **args支持的形式：**
  - str——字符串str

    > str不能与s相同

  - str, pos, len——str中从pos开始最多len个字符
  - cp, len——从cp指向的数组的前len个字符
  - cp——cp指向的数组以空字符结尾的数组
  - n, c——n个字符c
  - b, e——迭代器[b, e)
  - 初始化列表——花括号包围的，以逗号分隔的字符列表

  > - **`append、assign` 可以使用所有的形式**
  > - **`replace、insert` 允许的形式依赖于range和pos是如何指定的**

| replace (pos, len, *args*) | replace (b, e, *args*) | insert (pos, *args*) | insert (iter, *args*) | *args* 可以是 |
|---|---|---|---|---|
| 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 |

## string搜索操作

string类提供6个不同的搜索函数，每个函数有4个重载版本。

| 操作 | 含义 |
| --- | --- |
| c.find(args) | 查找s中args第一次出现的**位置** |
| s.rfind(args) | 查找s中args最后一次出现的位置 |
| s.find_first_of(args) | 在s中查找args中**任何一个字符**第一次出现买的位置 |
| s.find_last_of(args) | |
| 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) | |

- 每个搜索都返回一个`string::size_type`值，表示匹配发生位置的下标；

  如果搜索失败，则返回一个名为`string::nops`的static成员；

  > 标准库将`string::size_type`定义为**无符号整数**，`nops`初始化为-1，**此初始值意味着nops等于任何string最大的可能大小**、
  >
  > ```
  > string a = " abc";
  > unsigned int b = a.find('d');
  > int c = a.find('d');
  > cout<<b<<endl;      // 4294967295 (-1对应的无符号整数值)
  > cout<<c;         // -1
  > ```

- **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
  - s2, pos——从s中位置pos开始查找字符串s2
  - 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

    > 以上pos可不写，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，必须要写

  > 一种常见的程序设计模式是用pos参数**在字符串中循环地搜索子字符串出现的所有位置**

## compare函数

string关系运算符仅支持两个string类型进行比较；`compare()`允许两个string或一个string和字符数组进行比较

| s.compare()参数形式 | 含义 |
| --- | --- |
| s2 | 比较s和s2 |
| pos1, n1, s2 | 将s中从pos1开始的n1个字符与s2进行比较 |
| pos1, n1, s2, pos2, n2 | 将s中从pos1开始的n1个字符与s2中从pos2开始n2个字符进行比较 |
| cp | 比较s与cp指向的以空字符结尾的字符数组 |
| pos1, n1, cp | 将s中从pos1开始的n1个字符与p指向的以空字符结尾的字符数组 |
| pos1, n1, cp, n2 | 将s中从pos1开始的n1个字符与p指向的以空字符结尾的字符数组的前n2个字符进行比较 |

- 大于返回正数，小于返回负数，等于返回0

## 数值转换(常用)

| 操作 | 含义 |
| --- | --- |
| to_string（val) | 返回数值val的string表示<br>val可以是任何数值类型 |
| stoi(s, p, b)<br>stol(s, p, b)<br>stoul(s, p, b)<br>stoull(s, p, b) | 返回s的起始子串的数值，返回类型是int、long、unsing long、long long、unsigned long long.<br>b是转换所用的基数，默认是10<br>p是`size_t`指针，用来保存s中第一个非数值字符的下标，默认是0，即函数不保存下标（`??`） |
| stof(s, p)<br>stod(s, p)<br>stold(s, p) | 返回s的起始子串的数值，返回类型是float、double、long double; |

> 直接调用就行

- 要转换为数值的string中第一个非空白字符必须是数值中可能出现的字符

> +-.0123456789
>
> 可以包含e或E来表示指数部分

- 如果string不能转换为一个数值，抛出`invalid_argument`；

  如果转换得到的数值无法用任何类型表示，抛出`our_of_range`

## 容器适配器

> **理解容器适配器**
>
> 容器适配器的实现是通过封装**某个容器**，使其满足某些特定场景的需要；
>
> 其本质还是容器，只是该容器内部实现利用了大量基础容器模版类中已经写好的成员函数

STL提供三种容器适配器：`stack`、`queue`、`priority_queue`

- 所有容器适配器都支持的操作和类型

| 操作/类型 | 含义 |
| --- | --- |
| 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 | |
| A a(c) | 将荣器c中的元素拷贝给a(a, c元素类型相容) |
| 关系运算符 | 每个适配器都支持所有的关系运算符<br>这些运算符返回底层容器比较的结果 |
| a.empty() | |
| a.size() | |
| swap(a,b)<br>a.swap(b) | 交换a, b的内容，a, b必须有相同的类型，包括底层容器类型也必须相同 |

### 定义适配器

```
stack<int,deque<int>> stk;      // 第一个声明适配性的元素类型，第二个声明适配器底层实现的容器类型(可选)
```

| 容器适配器 | 基础容器筛选条件 | 默认使用的基础容器 |
| --- | --- | --- |
| stack | 基础容器需包含以下成员函数：<br>empty()size()back()push_back()pop_back()满足条件的基础容器有 vector、deque、list。 | deque |
| queue | 基础容器需包含以下成员函数：<br>empty()size()front()back()push_back()**pop_front()**满足条件的基础容器有 deque、list。 | deque |
| priority_queue | 基础容器需包含以下成员函数：<br>empty()size()front()push_back()pop_back()并且**要求随机访问能力**<br>满足条件的基础容器有vector、deque。 | vector |

> `back()` 返回当前容器中起始元素的引用。
>
> `front()` 返回当前容器中起始元素的引用。
>
> `pop_front()` vector没有该成员函数

**如何选择底层容器类型？？** （《*effective stl*》有详细解释）

## 栈适配器

`<stack>`

| 操作 | 含义 |
| --- | --- |
| s.pop() | 删除栈顶元素，但不返回该元素值 |
| s.push(item)<br>s.emplace(args) | 元素进栈 |
| s.top() | 获取栈顶元素 |

## 队列适配器

`<queue>`

| 操作 | 含义 |
| --- | --- |
| q.pop() | 移除队列首元素或优先级最高元素 |
| q.front() | 获取队列首元素<br>**只适用用于queue** |
| q.back() | 获取队列末尾元素<br>**只适用于queue** |
| q.top() | 获取最高优先级元素<br>**只适用于priority_queue** |
| q.push(item)<br>q.emplace(args) | 在queue末尾或priority_queue中恰当位置创建一个元素 |

- 优先级队列，默认使用 `<运算符` 来确定相对优先级

  > 第11章 介绍如何重载这个默认设置

---

1. 类模版的实例化（第3章） ↩

2. 内置类型初始化为0，类类型有默认构造函数初始化。(第3章) ↩

3. const成员函数(第7章) ↩