C++程序设计基础 第十一章

前言

本文档由 @ItsJiale 创作，作者博客：https://jiale.domcer.com/，作者依据数学与大数据学院 2024 级大数据教学班的授课重点倾向编写而成。所有内容均为手动逐字录入，其中加上了不少自己的理解与思考，耗费近一周时间精心完成。

此文档旨在助力复习 C++ 程序设计基础，为后续学习数据结构筑牢根基。信计专业的同学，也可参考本文档规划复习内容。需注意，若个人学习过程中存在不同侧重点或对重难点的理解有差异，应以教材内容为准。倘若文档内容存在任何不妥之处，恳请各位读者批评指正。

By：ItsJiale

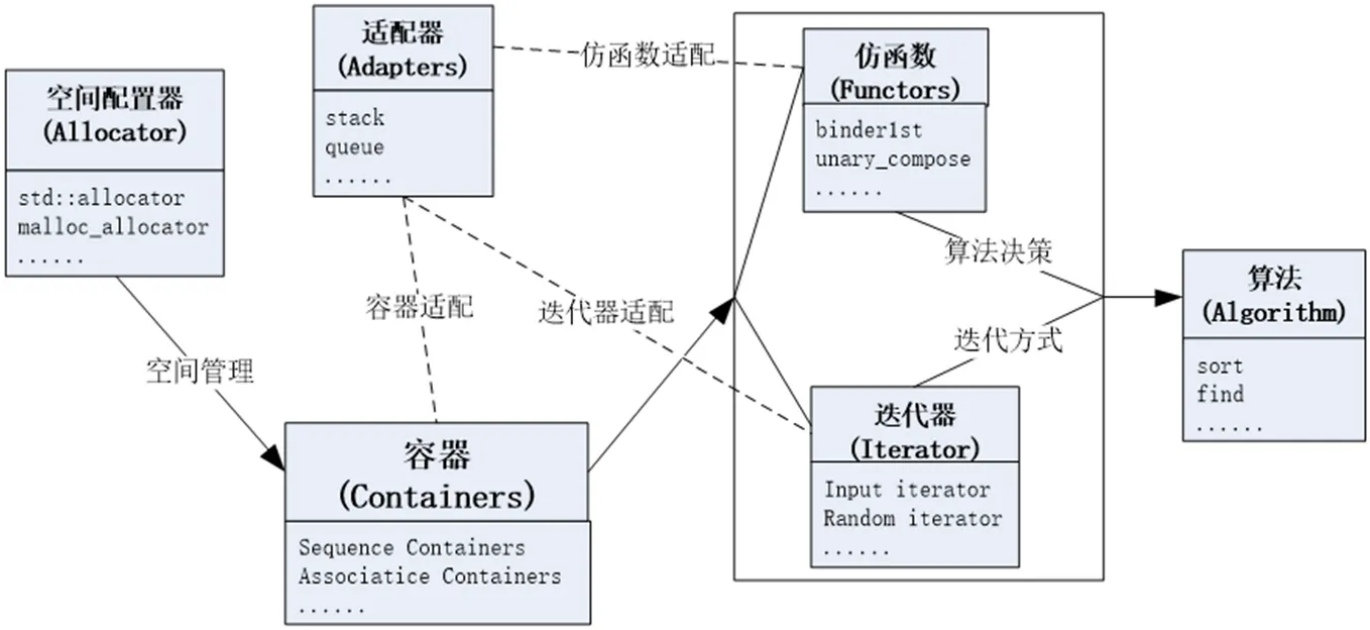
2025.5.5

第十一章 STL

标准模板库（*Standard Template Library*）是所有C++编译器和操作系统平台都支持的一种模板库。STL提供了大量的复用软件组织，能让C++程序设计者快速而高效地进行开发。

如果觉得本章不容易理解或晦涩难懂，那很大程度是你的心无法静下来，认真阅读并加以思考还是很容易明白的。

11.1 STL组成



1. 容器

有一种对象类型，它**可以装入其他对象或指向其他对象的指针**，这种对象类型就叫作容器。这种特殊的对象类型包含了处理其他对象的方法。

序列容器	关联容器	无序容器
vector	set	unordered_set (C++11)
list	multiset	unordered_ multise(C++11)
deque	map	multiset_ map(C++11)
arry(C++11)	multimap	unordered_ multimap(C++11)
forward_list(C++11)		

2. 空间配置器

C++标准库中采用了分配器(`allocator`)实现对象内存的空间分配和归还，空间分配器是特殊的内存模型，实现了对象处理内存的分配和归还操作。例如，使用 `vector` 容器，存储数据的空间由 `allocator` 完成内存的分配和资源回收。 `allocator` 本质上是对 `new` 和 `delete` 运算符的再次封装，对外提供可用的接口，实现了内存资源的自动化管理。

3. 适配器

容器适配器是一个封装了序列容器的类模板，在序列容器的基础上提供了不同的功能。STL中提供了三个容器适配器，`stack`(栈)、`queue`(队列)和`priority_queue`(优先队列)，除了存储普通数据类型，还可以存储`list`、`vector`、`deque`容器，这三种适配器用于特殊数据结构的处理，体现了STL设计的通用性，并极大提高了编程效率。

4. 迭代器

迭代器的作用是**将容器与算法联系起来，起着粘合剂的作用**，几乎STL提供的所有算法都是通过迭代器存取元素实现的。

STL共有输入迭代器、输出迭代器、正向迭代器、双向迭代器和随机访问迭代器五种类型的迭代器，使用迭代器访问容器元素更简单、易用，且代码更加紧凑、简洁。

5. 仿函数

在运算符重载章节讲解了仿函数是通过重载`()`运算符实现的，使类、模板具有函数一样的行为，仿函数在STL中是算法解决问题的策略，常用于生成数据、测试数据和数据操作。例如，在容器的元素排序中提供了`less`和`greater`方法，这些方法可以通过仿函数自己实现相对应的功能。

6. 算法

STL提供了算法的模板函数，这些算法适用于STL中的各类容器。例如，算法`for_each()`为指定序列中的每一个元素调用指定函数，算法`stable_sort()`对所指定的容器元素进行稳定性排序。只需要通过调用就可以完成所需要的功能。

> 上述只需有个大概印象即可，没人会死究定义为什么这么说。

11.2 序列容器

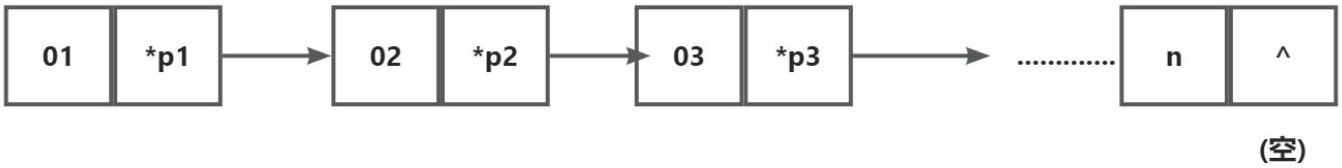
序列容器也叫做顺序容器，序列容器各元素之间有顺序关系，每个元素都有固定位置（与元素本身无关），除非使用**插入**或**删除**操作改变这个元素的位置。序列容器是一种线性结构的有序群集，它最重要的特点就是可以在容器的一端**添加**、**删除**元素。对于双向序列容器，允许在两端添加和删除元素。序列容器有**连续存储**和**链式存储**两种方式。

连续存储

链式存储

顺序表：除了表头表尾，中间任何一个数据元素都有一个**直接前驱**与**直接后继**。

链式储存



指针指向当前元素的直接后继

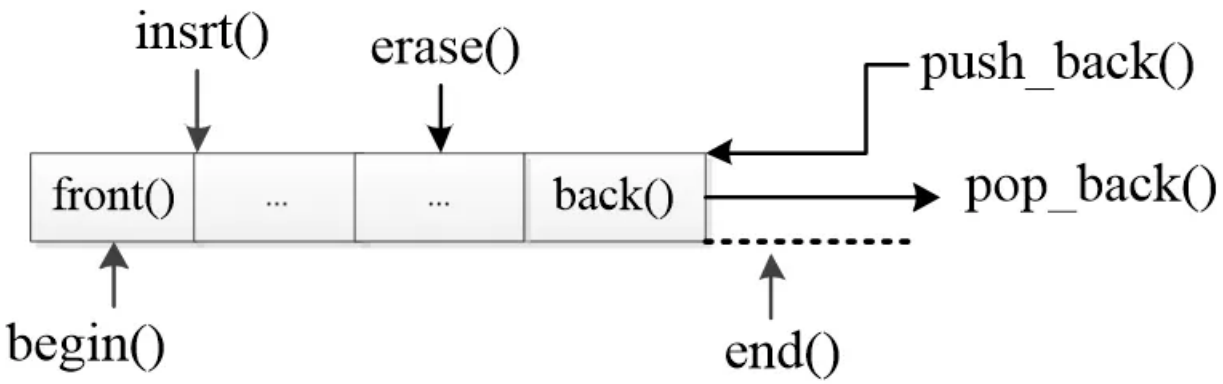
STL提供的基本序列容器包括 `vector` 、 `deque` 、 `list` 、 `array` 和 `forward_list` 五种，在使用时需要包含相应的头文件：

```
#include <vector>                    //随机访问的容器，支持元素添加和自动调整大小
#include <deque>                     //随机访问的双向队列容器类型
#include <list>                      //双向链表容器类型
#include <array>                     //C++11中新增加的固定大小容器类型
```

```
#include <forward_list>          //C++11中新增加的单向链表容器类型
```

11.2.1 vector

`vector` 类似于类模板。 `vector` 容器与动态数组相同，具有在插入或删除元素时自动调整自身大小的能力，容器能够自动处理其存储数据所需的空间。容器中的元素放置在连续的内存空间中，可以使用迭代器对其进行访问和遍历。下面是 `vector` 容器的存储结构图：



其中 `push_back()` 是初栈， `pop_back()` 是终栈， `end()` 是最后一个元素的下一个位置的首地址

`vector` 容器在插入元素时，插入位置之后的元素都要被顺序的向后移动，因此在总体上 `vector` 容器的插入操作效率并不高。插入位置越靠前，执行插入所需的时间就越多。但在 `vector` 容器尾部插入元素的效率就比较高了。

在删除 `vector` 容器中的元素时，被删除元素后面的元素会顺序向前移动，**将被删除元素留下的空位补上**。删除操作的效率和插入类似，被删除元素越靠前，删除操作所需的时间就越多，删除操作不会引起 `vector` 容器容量的改变，因此被删除之前的迭代器、指针、引用都能够继续使用，而删除元素之后的迭代器、指针、引用都会失效。

1. 创建vector容器

`vector` 模板中定义了不同的重载构造函数，因此， `vector` 对象的创建与初始化也有不同的方式。

a. 指定容器大小

`vector<元素类型> 对象名（容器大小）;`

在上述格式中， `<>` 中的类型名表示容器中元素的类型，如 `int` 表示容器中存储的是 `int` 类型的数据。()中指定容器大小。示例代码如下所示：

```
vector<int> v1(10);
```

```
vector<string> v2(5);
```

第一行创建了一个存储 `int` 类型数据的容器 `v1` ，容器大小为 `10`

第二行创建了一个存储 `string` 类型数据的容器 `v2` ，容器大小为 `5`

需要注意的是， `vector` 对象在定义后所有元素都会被初始化，如果是基本数据类型的容器，则都会被初始化为 `0` ；如果是其他类型的容器，则由类的默认构造函数初始化。

b. 指定初始值

在创建 `vector` 容器时，可以同时指定 `vector` 容器大小和初始值，格式如下所示：

```
vector<元素类型> 对象名（容器大小，元素初始值）;
```

按照上述格式创建 `vector` 容器，示例代码如下所示：

```
vector<int> v1(10, 1);
```

```
vector<string> v3(3, "aa");
```

第一行创建存储 `int` 类型数据的容器 `v1` ，容器大小为 `10` ， `10` 个元素的初始值均为 `1`

第二行创建存储 `string` 类型数据的容器 `v2` ，容器大小为 `3` ， `3` 个元素的初始值均为 `"aa"`

c. 列表初始化

C++11新标准还提供了另外一种 `vector` 容器初始化方式，用值的列表初始化，示例代码如下所示：

```
vector<int> v1{1, 2};                                    //v1有两个元素
```

```
vector<string> v2 = { "a", "b", "c" };                   //v2有三个元素
```

d. 初始化状态为空

初始化状态为空，即创建 `vector` 对象时不指定大小也不指定初始值：

```
vector<int> v1;                    //创建存储int类型数据的容器v1
```

除了上述方式， `vector` 容器还可以用另外一个 `vector` 容器完成初始化：

```
vector<int> v1(10, 1);
```

```
vector<int> v2(v1);                //用容器v1初始化容器v2
```

```
vector<int> v3 = v2;               //用容器v2给容器v3赋值
```

用一个 `vector` 容器初始化另一个容器或相互赋值时，两个 `vector` 容器的**元素类型必须相同**。

2. 获取容器容量和实际元素个数

`vector` 容器的容量与容器中元素实际个数并**不一定相同**，容器容量指**容器最多可以存储**的元素个数，是容器预分配的内存空间。**而容器实际存储元素个数可能小于容器容量**。

`vector` 提供了两个函数 `capacity()` 和 `size()` ，分别用于获取容器容量和容器实际元素个数：

```
v.capacity();
```

```
v.size();
```

3. 访问容器中的元素

由于 `vector` 重载了 `[]` 运算符，因此可以使用索引方式访问 `vector` 容器中的元素。此外， `vector` 还提供了一个成员函数 `at()` ，用于随机访问容器中的元素， `at()` 函数调用形式如下所示：

```
v.at(int indx);
```

参数 `indx` 表示索引。 `at()` 函数返回值为索引指向的数据。

4. 赋值函数

`vector` 容器中的元素可以在创建容器时指定，也可以通过[]运算符完成赋值。除此之外， `vector` 还提供了一个成员函数 `assign()` ，用于给空的 `vector` 容器赋值。 `assign()` 函数有两种重载形式，分别如下所示：

```
v.assign(n, elem);              //将n个elem元素赋值给容器
```

```
v.assign(begin, end);           //将[begin, end)区间中的元素赋值给容器  区间取值为 [  ,   )
```

5. 获取头部和尾部元素

`vector` 提供了 `front()` 函数与 `back()` 函数，分别用于获取容器的头尾元素。 `front()` 函数和 `back()` 函数调用形式如下所示：

```
v.front();                      //获取容器头部元素（第一个元素）
```

```
v.back();                       //获取容器尾部元素（最后一个元素）
```

6. 从尾部插入和删除元素

`vector` 提供了一对函数 `push_back()` 与 `pop_back()`，分别用于从尾部添加元素和删除尾部元素。`push_back()` 函数和 `pop_back()` 函数调用形式如下所示：

`v.push_back(type elem& t);`

`v.pop_back();`

C++

```cpp
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> v;           //创建一个空的vector容器v
    for (int i = 0; i < 10; i++)
        v.push_back(i + 1);     //从尾部向容器v中插入10个元素
    for (int i = 0; i < 10; i++)
        cout << v[i] << "  ";   //输出容器v中的元素
    cout << endl;
    v.pop_back();            //移除尾部元素
    for (int i = 0; i < 9; i++)          //此时元素个数为9
        cout << v[i] << "  ";   //输出容器v中的元素
    cout << endl;
    return 0;
}

```

7. 容器的迭代器

`vector` 容器提供了迭代器，通过迭代器可以访问、修改容器中的元素。`vector` 容器提供了 `iterator`、`const_iterator`、`reverse_iterator` 和 `const_reverse_iterator` 四种迭代器。

- iterator：正向遍历容器元素。
- const_iterator：正向遍历容器元素，但通过const_iterator只能访问容器元素，不能修改元素的值。
- reverse_iterator：反向遍历容器元素。
- const_reverse_iterator：反向遍历容器元素，但通过const_reverse_iterator只能访问容器元素，不能修改元素的值。

在使用迭代器之前，必须先要定义迭代器对象，`vector`容器迭代器对象的定义格式如下所示：

`vector<元素类型> 迭代器 迭代器对象名称;`

> *理解形式上：迭代器相当于指针；迭代器对象名称相当于指针名*

在实际编程中，通常将创建的迭代器对象简称为迭代器。迭代器可以执行`++`、`--`、与整数相加减等算术运算，以达到遍历容器元素的目的。

`vector`提供了获取上述四种迭代器的成员函数，如下表。

函数	含义
begin()	返回容器的**起始位置**迭代器iterator
end()	返回迭代器的结束位置迭代器iterator
rbegin()	返回容器结束位置作为起始位置的反向迭代器reverse_iterator
rend()	返回反向迭代的最后一个元素之后的位置的反向迭代器reverse_iterator
cbegin()	返回指向容器中起始位置的常量迭代器const_iterator，不能修改迭代器指向的内容
cend()	返回迭代器的结束位置的常量迭代器const_iterator，不能修改迭代器指向的内容
crbegin()	返回容器结束位置作为起始位置的迭代器const_reverse_iterator
crend()	返回第一个元素之前位置的常量迭代器const_iterator

迭代器遍历容器到达尾部时，指向最后一个元素的后面，而不是指向最后一个元素，即使用`end()`函数、`rend()`函数、`cend()`函数、`crend()`函数获取的迭代器，指向最后一个元素后面的位置，而不是指向最后一个元素。

还是那句话——防自学教材

我们用例子来看：

C++

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    // 创建一个包含整数的向量
    vector<int> numbers = {1, 2, 3, 4, 5};

    // 使用 begin() 和 end() 正向遍历
    cout << "正向遍历: ";
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    // 使用 rbegin() 和 rend() 反向遍历
    cout << "反向遍历: ";
    for (auto rit = numbers.rbegin(); rit != numbers.rend(); ++rit) {
        cout << *rit << " ";
    }
    cout << endl;

    // 使用 cbegin() 和 cend() 正向常量遍历
    cout << "正向常量遍历: ";
    for (auto cit = numbers.cbegin(); cit != numbers.cend(); ++cit) {
        cout << *cit << " ";
    }
    cout << endl;

    // 使用 crbegin() 和 crend() 反向常量遍历
    cout << "反向常量遍历: ";
    for (auto crit = numbers.crbegin(); crit != numbers.crend(); ++crit) {
        cout << *crit << " ";
    }
    cout << endl;

    return 0;
}
```

执行结果 Plain Text

```
正向遍历: 1 2 3 4 5
反向遍历: 5 4 3 2 1
正向常量遍历: 1 2 3 4 5
反向常量遍历: 5 4 3 2 1
```

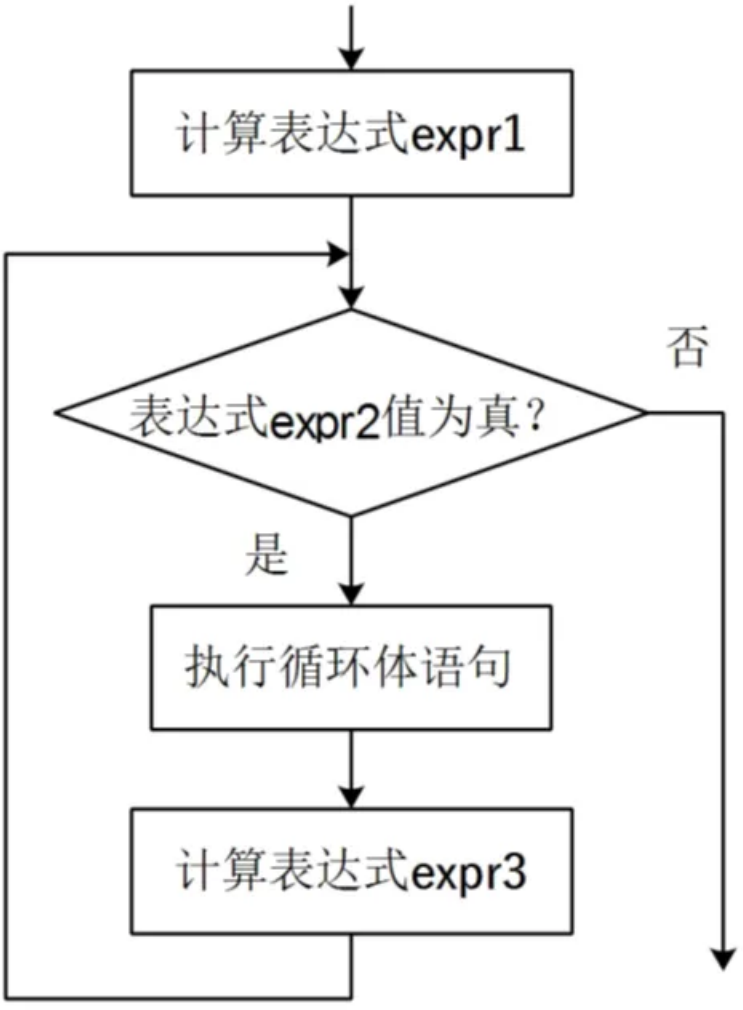
对于 `for` 循环语句的重新思考:

在开发等日常中，我们常见的 `for` 循环一般采用遍历，即大多数都是 `for(int i =0;i<某个值;i++){}`

可是这里的 `for` 循环却不是这种样式。 `for` 循环的本质是什么?

`for` 语句的语法格式为:

```
for(expr1;expr2;expr3){
    statement;
}
```



首先计算表达式 `expr1` ，**再求**表达式 `expr2` 的值，如果 `expr2` 的值**为真**，则执行循环体语句，否则跳出 `for` 语句的执行。在循环体语句执行完毕后，**紧接着**计算表达式 `expr3` ，**再次**计算表达式 `expr2` ，决定是继续执行还是结束 `for` 语句的执行。

在 `for` 语句中，表达式 `expr1` 只被执行一次，一般情况下用于初始化对象，表达式 `expr2` 常用来控制循环的条件，而表达式 `expr3` 常用来**修改表达式** `expr1` 中初始化的对象的值。

注意：

`vector` 容器是一个顺序容器，在内存中是一块**连续的内存**，当删除一个元素后，内存中的数据会移动，从而保证数据的连续，当删除数据后，其他数据的地址也发生变化，迭代器获取容器位置信息的数据不正确，导致访问出错。

8. 在任意位置插入和删除元素

`vector` 提供了一对向容器任意位置插入和删除元素的函数 `insert()` 与 `erase()` 。其中， `insert()` 函数用于向容器中插入元素，它有三种重载形式：

```
v.insert(pos, elem);            //在pos位置上插入元素elem
v.insert(pos, n, elem):         //在pos位置上插入n个elem元素
v.insert(pos, begin, end);      //在pos位置插入[begin, end)区间的数据
```

`erase()` 函数用于删除容器中的元素，它有两种重载形式，分别如下所示：

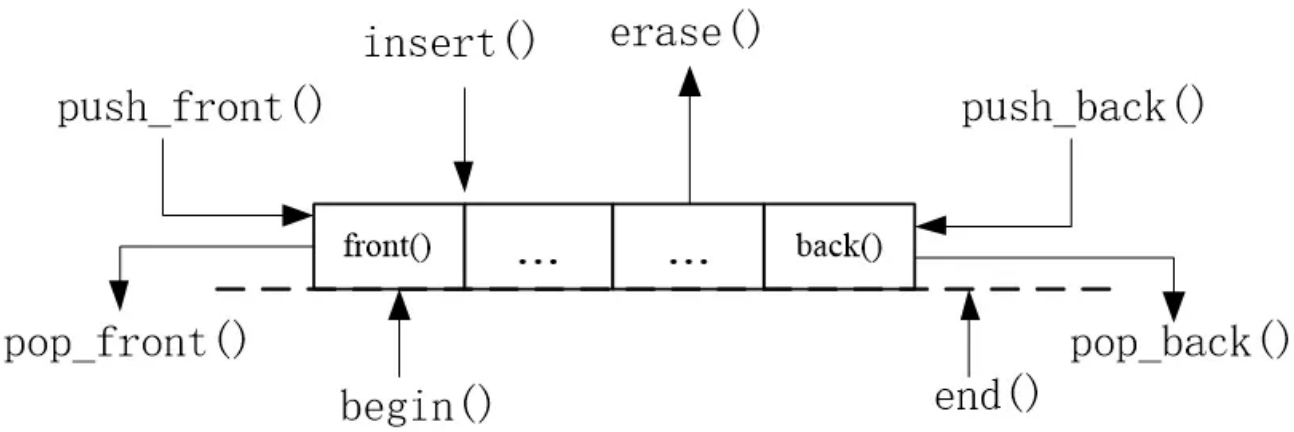
```
v.erase(pos);                   //移除pos位置上的元素
v.erase(begin, end);            //移除[begin, end)区间的数据
```

`insert()` 函数与 `erase()` 函数中的位置只能由 `begin()` 、 `end()` 等函数返回的**迭代器指示**，**不能用**纯粹的数字作为参数。下面用例子展示：

```C++
#inlcude<iostream>
#inlcude<vector>
using namespace std;
int main()
{
    vector<char> v;                 //创建空的vector容器v
    v.insert(v.begin(), 'a');       //在头部位置插入元素a
    v.insert(v.begin(), 'b');       //在头部位置插入元素b
    v.insert(v.begin(), 'c');       //在头部位置插入元素c
    v.insert(v.begin() + 1, 5, 't');    //在v.begin()+1位置插入5个元素t
    for (int i = 0; i < 8; i++)     //输出容器v中的元素
        cout << v[i] << " ";
     v.erase(v.begin() + 1, v.begin() + 6);     //删除5个元素
    for (int i = 0; i <3; i++)          //输出容器v中的元素
        cout << v[i] << " ";
    return 0;
}
```

补充:

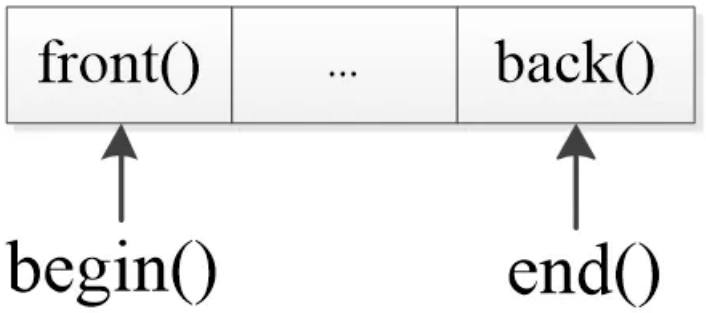
`deque` 容器与 `vector` 容器非常相似，采用的是动态内存管理的方式存储元素的，提供了随机访问的方法，有着和 `vector` 容器几乎相同的操作方法。 `deque` 容器的存储结构如下图。



`deque` 容器的实现是一个双向队列，这是与 `vector` 容器最大的区别。 `deque` 容器不支持 `vector` 容器中的 `reserve()` 、 `capacity()` 和 `data()` 函数，其余函数均支持。 `deque` 容器新增的容器操作函数有 `pop_front()` 、 `push_front()` ，用于从队首和队尾弹出元素， `emplace_back()` 从队尾添加元素。

11.2.2 array

C++11标准增加了 `array` 容器， `array` 是一个编译阶段**确定大小**的序列容器，是一个严格按照线性排序的特定数量元素的容器，大小与定义的数组是等效的。与其他容器不同的是， `array` **大小固定**，且没有分配容器空间、删除等操作。 `array` 容器结构如下图。



下面是 `array` 和 `vector` 的比较

比较维度	**`array`**	**`vector`**
大小特性	编译时确定大小，运行时不可改变	运行时可动态调整大小
内存分配	元素存储在栈上，由编译器自动管理	元素存储在堆上，运行时动态分配和释放
性能表现	访问元素效率高，不支持插入和删除操作	尾部插入删除平均 (O(1))，中间或开头插入删除 (O(n))
接口和场景	接口类似 C 风格数组，适用于固定大小、高性能场景	接口丰富，适用于元素数量不确定、需动态调整的场景

1. 创建array容器

`array` 容器由类模板定义，创建 `array` 容器的时候需要指定元素类型和元素个数，这是与 `vector` 定义不同的地方，示例代码如下：

```
array<int,3>a1;                  //定义array容器a1,未初始化
```

`array<int,3>a1={1,2,3};` `//定义array容器a2,使用列表初始化方式初始化`

2. 修改容器元素

`array` 提供了 `fill()` 函数和 `swap()` 函数用于修改元素。 `fill()` 函数使用指定的数据填充容器； `swap()` 函数用于交换两个容器的元素。

`fill(val);` `//使用val填充容器`

`a1.swap(a2);` `//交换容器a1和容器a2的元素`

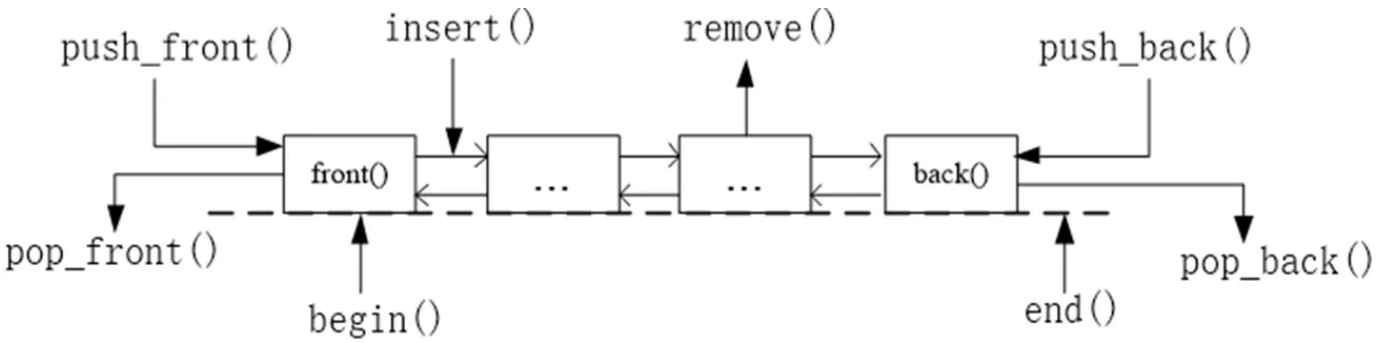
下面是简单的例子：

C++

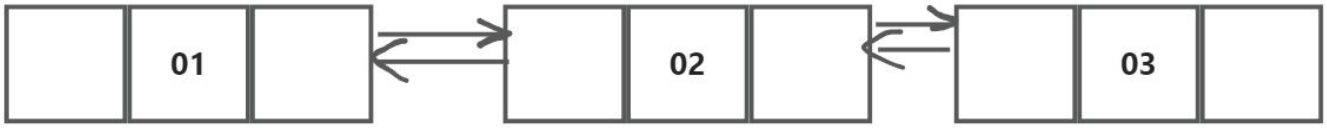
```cpp
#include<iostream>
#include<array>
using namespace std;
int main()
{
    array<int,3>c={1,2,3};
    array<int,3>c1={2,3,4};
    array<int,3>::iterator pos;   //定义iterator迭代器pos
    c.swap(c1);                   //交换容器c和容器c1的元素
    for(pos=c.begin();pos!=c.end();++pos){    //使用迭代器pos遍历容器c中的元素
        cout<<*pos<<" ";
    }
    return 0;
}

```

11.2.3 list

`list` 容器是一个双向链表，因为同为序列式容器，所以它的接口大部分都与 `vector` 和 `deque` 相同。 `list` 容器容器存储结构如下图。



双向链表



`list` 容器是以**双向链表**形式实现的， `list` 容器中的元素**通过指针将前面的元素和后边的元素链接到一起**。与 `vector` 容器和 `array` 容器相比， `list` 容器通过**迭代器**获取元素和插入元素，使用 `list` 容器可以使用大量的算法，提高编程效率。 `list` 容器的不足之处是不能直接通过位置（下标）访问元素，**只能**从迭代器获取的位置获取元素。

1. 创建list容器

`list` 类模板中也实现了多个重载构造函数，因此 `list` 容器的创建也有多种方式。创建 `list` 容器的几种常见方式如下所示：

```
list<T> lt;                    //创建一个空的list容器lt
list<T> lt(n);                 //创建一个list容器lt，大小为n
list<T> lt(n, elem);           //创建一个list容器lt，包含n个elem元素
list<T> lt(begin, end);        //创建一个list容器lt，用[begin, end)区间的值为元素赋值
list<T> lt(lt1);               //创建一个list容器lt，用容器lt1初始化
```

2. 赋值

`list` 也提供了 `assign()` 函数为容器元素赋值， `assign()` 函数有两种重载形式，分别如下所示：

```
lt.assign(n, elem);                    //将n个elem元素的值赋值给lt
lt.assign(begin, end);                 //用[begin, end)区间的值给lt中的元素赋值
```

`在list容器中，assign()函数的用法和vector中的assign()函数用法一样。`

3. 元素访问

因为 `list` 是由链表实现的，**内存区域并连续**，所以**无法**用 `[]` 运算符访问元素，也没有可随机访问元素的 `at()` 方法，但 `list` 提供了 `front()` 函数和 `back()` 函数用于获取头部元素和尾部元素。此外， `list` 也支持迭代器访问元素，提供了 `iterator` 、 `const_iterator` 、 `reverse_iterator` 和 `const_reverse_iterator` 四种迭代器，也提供了获取这四种迭代器的成员函数。 `list` 迭代器的用法与 `vector` 迭代器相同。

4. 插入元素和删除元素

`list` 提供了多个函数向容器中添加元素，这些函数调用如下所示：

```
lt.push_back();                        //在尾部插入元素
lt.push_front();                       //在头部插入元素
lt.insert(pos, elem);                  //在pos位置插入元素elem
lt.insert(pos, n, elem);               //在pos位置插入n个元素elem
lt.insert(pos, begin, end);            //在pos位置插入[begin, end)区间的值作为元素
```

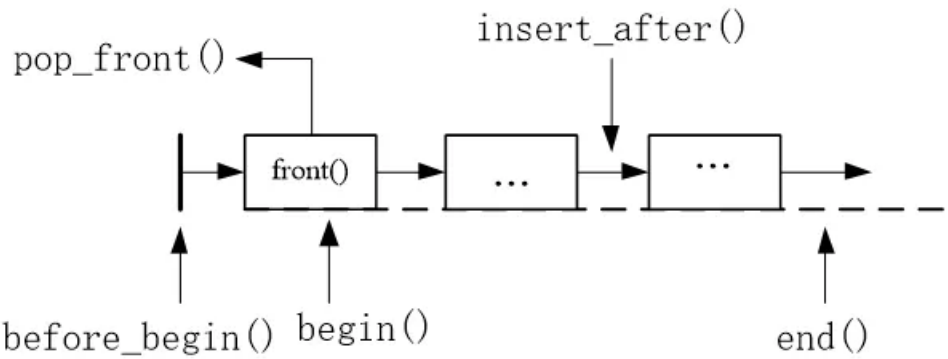
`list` 也提供了多个函数用于删除元素，可以从头尾两端删除元素，也可以删除中间任意一个元素。 `list` 各个删除函数调用形式如下所示：

```
lt.pop_back();                         //从尾部删除元素
lt.pop_front();                        //从头部删除元素
lt.erase(pos);                         //从中间删除元素
lt.erase(begin, end);                  //删除[begin, end)区间的元素
```

```
lt.remove(elem);                                    //从容器中删除所有与elem匹配的元素
```

11.2.4 forward_list

C++11标准新增了 `forward_list` 容器，该容器由**单链表**实现。在 `forward_list` 容器中，**除了最后一个元素**，每个元素与下一个元素**通过指针链接**。由于是单链表实现的，因此 `forward_list` 容器只能向后迭代 ***（直接后继）***。 `forward_list` 容器存储结构如下图。



由于同为链式存储的容器，因此 `forward_list` 的接口与 `list` 大部分相同。但是又因为 `forward_list` 是单链式存储，所以 `forward_list` 还提供了一些自己特有的函数。

1. 插入和删除元素

`forward_list` 容器不支持 `insert()` 函数和 `erase()` 函数，但它提供了 `insert_after()` 函数和 `erase_after()` 函数用于插入和删除元素。 `insert_after()` 函数和 `erase_after()` 函数的调用形式如下所示：

```
insert_after(pos,val);                              //将元素val插入到pos位置值

insert_after(pos,begin,end);                        //在pos位置插入[begin,end)区间内的元素

erase_after(pos);                                   //删除pos位置的元素

erase_after(begin,end);                             //删除[begin,end)区间内的元素
```

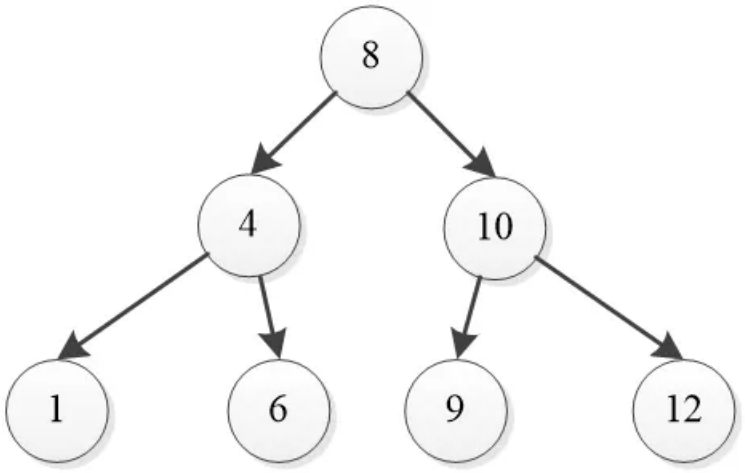
2. 获取迭代器

`forward_list` 新增了两个函数 `before_begin()` 和 `cbefore_begin()`，其中，`befor_begin()` 函数用于获取指向容器第一个元素**之前位置**的迭代器。`cbefore_begin()` 用于获取指向容器第一个元素**之前位置**的 `const_iterator` 迭代器。

11.3 关联容器

关联型容器所有元素都是经过排序的，关联型容器都是有序的。它的每一个元素都有一个键（key），容器中的元素是按照键的**取值升序排列**的。

关联型容器内部实现为一个**二叉树**，在二叉树中，每个元素都有一个父节点和两个子节点，左子树的所有元素都比自己小，右子树的所有元素都比自己大。



关联型容器内部结构都以这种二叉树结构实现，这也使得它可以高效的查找容器中的每一个元素，但却不能实现任意位置的操作。

STL提供了四种关联型容器：`set`、`multiset`、`map`、`multimap`，其中 `set` 与 `multiset` 包含在头文件 `set` 中，`map` 与 `multimap` 包含在头文件 `map` 中。

1. set与multiset

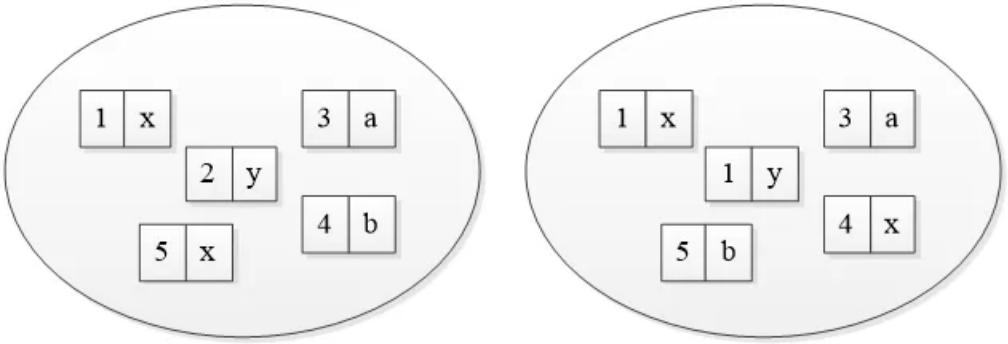
`set` 与 `multiset` 都是集合，用于存储一组相同数据类型的元素。**两者的区别**是 `set` 用来存储一组**无重复**的元素，而 `multiset` **允许有重复**的元素。

集合支持插入、删除、查找等操作，但集合中的元素值不可以直接修改，因为这些元素都是自动排序的，如果想修改某一个元素的值，**必须先删除**原有的元素，再插入新的元素。

2. map与multimap

`map` 与 `multimap` 称为映射，映射与集合的主要区别在于，集合的元素是键本身，而映射的元素是**由键和附加数据构成的二元组**，它很像“字典”，通过**给定的键**，可以**快速找出与键对应的值*（类似于Java中的HashMap）***。因此，映射的二叉树节点中存储了两个数据，一个是用来**定位的数据**，称为**键**，另一个是与键对应的数据，称为**值**。通常也说映射中存储的是一对**键值对**，映射的一种通常用法就是**根据键查找对应的值。**

映射又分为单重映射 `（map）` 与多重映射 `（multimap）` ，两者的主要区别是 `map` 存储的是无重复键的元素对，而 `multimap` 允许相同的键重复出现，即一个键可以对应多个值。



11.3.1 set和multiset

`set` 与 `multiset` 的区别在于是否允许有重复元素，其他用法都很相似，因此将这两种容器放在一起讲解。

1. 创建set与multiset容器

`set` 与 `multiset` 都重载了多个构造函数，因此创建 `set` 和 `multiset` 容器的方式有多种。 `set` 容器的创建方式主要有以下几种：

```
set<T> s;                          //创建一个空的set容器，默认升序排列
set<T, op> s;                      //创建一个空的set容器，按op规则排序
set<T> s(begin, end);     //创建一个容器，用[begin, end)区间为其初始化
set<T,op> s(begin, end);  //创建一个容器，用[begin, end)区间为其初始化，并按op规则排序
set<T> s(s1);                       //创建一个空的set容器，用另一个容器s1初始化
```

`T` 代表的是 `set` 容器所存储元素的类型，`op` 指的是用于元素排序的比较规则（*不是原批！*），它是一个可选的模板参数。`set` 会依据这个规则来确定元素的顺序，并且保证元素的唯一性。`op` 必须是一个二元谓词，也就是一个可调用对象（例如函数、函数对象或者 `Lambda` 表达式），该对象接受两个 `T` 类型的参数，然后返回一个布尔值。

如果不指定 `op`，`set` 会采用 `less<T>` 作为默认的比较规则，这意味着元素会按照升序排列。

下面分别用这五种方式创建 `set` 容器：

创建 `set` 容器：

```
set<char> s1;
```

```
set<int,greater<int>()> s2;
```

```
set<float> s3(begin, end);
```

```
set<string, greater<string>()> s4(begin, end);
```

```
set<int> s5(s2);
```

注意：`less` 是**升序**，`greater` 是**降序**

创建 `multiset` 容器：

```
multiset<char> ms1;
```

```
multiset <int, greater<int>> ms2;
```

```
multiset <float> ms3(begin, end);
```

```
multiset <string, greater<string>> ms4(begin, end);
```

```
multiset <int> ms5(s2);
```

2. 容器的大小

与其他容器一样，`set` 与 `multiset` 也提供了获取容器实际元素个数的函数 `size()`，判断容器是否为空的函数 `empty()`，以及返回容器容量的成员函数 `max_size()`。

```
s.size();                    //返回容器中实际元素个数
```

```
s.max_size();                //返回容器容量
```

```
s.empty();                          //判断容器是否为空
```

3. 容器元素的查找和统计

`set` 与 `multiset` 还提供了查找函数 `find()` 和统计函数 `count()` 。

```
s.find(elem);
```

```
s.count(elem);
```

`find()` 函数的功能是在容器中查找 `elem` 元素是否存在，如果元素存在，**返回指向查找元素的迭代器**；如果元素不存在，**返回末尾迭代器**。 `count()` 函数用于统计 `elem` 元素的个数。对于 `set` ， `count()` 函数的返回值只能是0或1；对于 `multiset` ， `count()` 函数返回值可能大于1。

4. 容器的迭代器

`set` 与 `multiset` 支持迭代器操作，提供了 `iterator` 、 `const_iterator` 、 `reverse_iterator` 、 `const_reverse_iterator` 四种迭代器，并且提供了获取这四种迭代器的成员函数。 `set` 与 `multiset` 的迭代用法与 `vector` 迭代器相同。

5. 插入和删除元素

`set` 与 `multiset` 提供了 `insert()` 函数与 `erase()` 函数，用于向容器中插入和删除元素。 `insert()` 函数主要有三种重载形式，分别如下所示：

```
s.insert(elem);                     //在容器中插入元素elem
```

```
s.insert(pos, elem);                //在pos位置插入元素elem
```

```
s.insert(begin, end);               //在容器中插入[begin, end)区间的元素
```

第一种形式的 `insert()` 函数将元素 `elem` 插入容器中。对于 `set` ，第一种形式 `insert()` 函数调用的返回值是一个 `pair<iterator, bool>` 对象。 `pair<iterator, bool>` 对象第一个参数 `iterator` 是迭代器，指示元素插入的位置；第二个参数 `bool` 类型的值代表元素是否插入成功。这是因为 `set` 容器中不允许存在重复的元素，如果要插入一个容器中已存在的元素，则插入操作会失败，而 `pair` 中的 `bool` 值就是标识插入是否成功。 `multiset` 不存在这样的情况，因此 `multiset` 返回的是一个 `iterator` 。

第二种形式的 `insert()` 函数将元素 `elem` 插入指定位置 `pos` ，返回指向 `pos` 位置的迭代器；

第三种形式的 `insert()` 函数将 `[begin,end)` 区间的元素插入容器，函数没有返回值。

`set` 与 `multiset` 提供的 `erase()` 函数主要有三种重载形式，分别如下所示：

```
s.erase(pos);                   //删除pos位置上的元素
s.erase(begin, end);            //删除[begin, end)区间上的元素
s.erase(elem);                  //删除元素elem
```

补充：pair类模版

`pair` 就像是一个“二人组小包裹”，它能把两个数据打包在一起，形成一个二元组，也就是一对元素。这两个数据可以是同一种类型，比如都是整数；**也能是不同类型**的，像一个整数搭配一个字符串。
要是你有两个元素想把它们**关联起来**、**组合成一个整体**，就可以用 `pair` 对象来实现。举个例子，你要存储某本书的价格和销量，就可以把价格和销量这两个元素组合在一个 `pair` 里。
在写代码的时候，如果一个函数需要**返回两个数据**，用 `pair` 就很合适。函数可以返回一个 `pair` 对象，把这两个数据一次性带出来，就像用一个包裹把两个东西一起寄出去一样。

1. 创建pair对象

创建 `pair` 对象可以调用 `pair` 的构造函数，还可以调用STL提供的 `make_pair()` 函数， `make_pair()` 是一个函数模板，原型如下所示：

```
template<class T1, class T2>
pair<V1,V2> make_pair( T1&& t, T2&& u );
```

在上述函数模板原型中，参数 `t` 与 `u` 是构成 `pair` 对象的两个数据。 `make_pair()` 函数内部调用的仍然是 `pair` 构造函数。

```
pair<int,string> p1(1, "abc");
make_pair(1, "abc");
```

2. pair对象的使用

`pair` 类模板提供了两个成员变量 `first` 与 `second` ， `first` 表示 `pair` 对象中的第一个元素， `second` 表示第二个元素。例如下面的代码：

`pair<int,string> p1(1, "abc");`

`cout << p1.first << endl;`

`cout << p1.second << endl;`

上述代码创建了一个 `pair` 对象 `p1`， `p1.first` 获取的是第一个元素： `int` 类型的 `1`； `p1.second` 获取的是第二个元素： `string` 类型的 `“abc”` 。

完整代码如下：

```C++
#include <iostream>
#include <utility>
#include <string>

using namespace std;

int main() {
    // 直接构造 pair 对象
    pair<int, string> p1(1, "abc");
    cout << "p1: " << p1.first << ", " << p1.second << endl;

    // 使用 make_pair 函数创建 pair 对象
    auto p2 = make_pair(2, "def");
    cout << "p2: " << p2.first << ", " << p2.second << endl;

    return 0;
}
```

11.3.2 map和multimap

`map` 与 `multimap` 中存储的是元素对 `（key-value）`， `map` 和 `multimap` 容器通常也可理解为关联数组，可以**使用键**作为下标获取对应的值。关联的本质在于元素的值与某个特定的键相关联，**而不是通过位置索引**获取元素。

1. 创建map和multimap容器

`map` 与 `multimap` 容器创建方式一样，有以下五种。

`map<T1, T2> m;` `//创建一个空的map容器`

`map<T1,T2, op> m;` `//创建一个空的map容器，以op为准则排序`

`map<T1, T2> m(begin, end);` //创建一个map容器，用[begin, end)区间值赋值

`map<T1, T2> m(begin,end,op);` //创建一个map容器，用[begin, end)区间值按op准则赋值

`map<T1, T2> m(m1);` //创建一个map容器，用另一个map容器m1初始化

下面分别用五种方式创建 `map` 容器：

创建 `map` 容器

`map<int, int> m1;`

`map<char, float, greater<int>> m2;`

`map<string, int> m3(begin, end);`

`map<string, int, greater<string>> m4(begin, end);`

`map<int, int> m5(m1);`

2. 容器元素的访问

`map` 重载了 `[]` 运算符，可以通过 `m[key]` 的方式获取 `key` 对应的值。此外， `map` 也提供了 `at()` 函数用于访问指定键对应的值。

`m.at(key);`

`m[key];`

`m[key] = value;`

`m.at(key) = value;`

`map` 容器可以通过上述两种方式随机访问容器中元素，但 `multimap` 中允许存在重复的键值，因此**无法使用**上述两种方式随机访问容器中元素。

3. 容器的迭代器

`map` 与 `multimap` 支持迭代器操作，提供了 `iterator` 、 `const_iterator` 、 `reverse_iterator` 、 `const_reverse_iterator` 四种迭代器，并且提供了获取这四种迭代器的成员函数。 `map` 与 `multimap` 迭代器用法与 `vector` 迭代器相同。

4. 插入和删除元素

`map` 和 `multimap` 的成员函数 `insert()` 用于插入元素， `insert()` 函数主要有三种重载形式，分别如下所示：

```
m.insert(elem);                         //在容器中插入元素elem
m.insert(pos, elem);                    //在pos位置插入元素elem
m.insert(begin, end);                   //在容器中插入[begin, end)区间的值
```

由于 `map` 和 `multimap` 中存储的是**键值对**，因此插入函数与其他容器稍有不同，每次插入一对元素，参数中的 `elem` 指的是一对元素。在插入元素时，使用 `pair<>` 语句或 `make_pair()` 函数创建一个 `pair` 对象，将 `pair` 对象作为 `insert()` 函数的参数，插入到容器中。

与 `insert()` 函数相对应， `map` 与 `multimap` 也提供了 `erase()` 函数用于删除容器中的元素， `erase()` 函数主要有三种重载形式，分别如下所示：

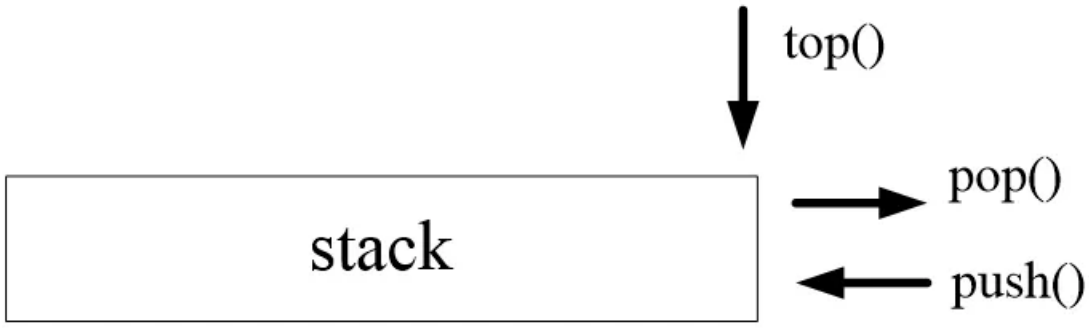
```
m.erase(pos);                           //删除位置pos上的元素
m.erase(begin, end);                    //删除[begin, end)范围内的元素
m.erase(key);                           //删除键为key的元素对
```

11.4 容器适配器

C++标准库中提供了三个容器适配器： `stack` （栈）、 `queue` （队列）和 `priority queue` （优先队列）

11.4.1 stack

`stack` 存储的元素具有**后进先出**的特点， `stack` 提供了 `push()` 函数向容器中插入元素，同时提供了 `pop()` 函数将最后插入的元素从容器中移除。 `stack` 容器适配器存储结构如下图。



1. 创建stack容器

创建 `stack` 容器适配器的构造函数原型如下：

创建 `stack` 容器主要有两种方式，分别如下所示：

（1）创建空的 `stack` 容器

创建空的 `stack` 容器，用于存储基本数据类型的元素，示例代码如下：

```
stack<int> st;
```

上述代码创建了一个空的 `stack` 容器 `st` ，向容器 `st` 中插入元素可以调用 `push()` 函数。

（2）创建存储序列容器的 `stack` 容器

创建存储序列容器的 `stack` 容器， `stack` 容器的类型参数有两个，第一个类型参数为**元素的数据类型**，第二个类型参数为**容器的类型**，示例代码如下所示：

```
vector <int> v = { 1,2,3 };                 //创建vector容器v
stack<int,vector<int> > s(v);               //创建stack容器s，存储容器v
```

2. 元素访问

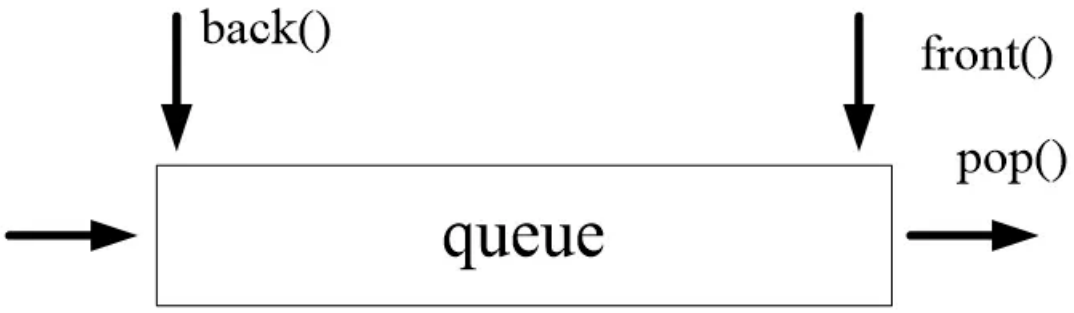
`stack` 容器适配器除了具有 `vector` 容器相同功能的成员函数之外，如 `empty()` 、 `size()` 、 `emplace()` 和 `swap()` 函数，还具有以下操作函数。

函数	含义
top( )	返回栈顶元素的引用，即最后一个进入stack容器适配器的元素
push(val)	将元素val插入到栈顶，无返回值

pop( )	删除栈顶的元素，无返回值
swap(s1,s2)	交换两个stack容器中的元素，是非成员函数重载(C++11)

11.4.2 queue

`queue` 具有**先进先出**的特点。容器中的元素只能从一端使用 `push()` 函数进行插入，从另一端使用 `pop()` 函数进行删除，`queue` 容器适配器**不允许一次插入或删除多个元素，且不支持迭代器**方法如 `begin()` 、 `rbegin()` 等。



`queue` 容器适配器具有 `vector` 容器相同功能的成员函数 `empty()` 、 `size()` 、 `emplace()` 和 `swap()` 函数等，容器适配器**创建**方式与 `stack` 容器适配器**相同**。 `queue` 容器适配器特有的成员函数如下表。

函数	含义
front()	返回queue容器适配器中第一个放入的元素
back()	返回queue容器适配器中最后一个插入的元素

C++

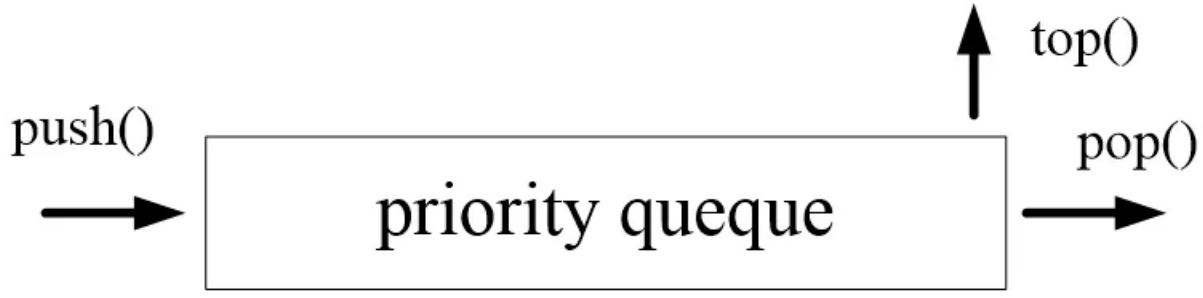
```cpp
#inlcude<iostream>
#inlcude<list>
#inlcude<queue>
using namespace std;

int main()
{
    list <int >l = { 1,2,3 };
    queue<int, list <int>> q(l);      //创建queue容器适配器q
    q.push(4);
    q.emplace(5);
    q.pop();
    cout << "第一个元素" << q.front() << endl;
    cout << "最后一个元素" << q.back() << endl;
    while(!q.empty()){
        cout << " "<<q.front();
        q.pop();}
    return 0;
}
```

输出结果 C++

```
第一个元素2
最后一个元素5
2 3 4 5
```

11.4.3 priority queue

`priority_queue` 是优先队列，它是队列的一种，但 `priority_queue` 可以按照**自定义的方式对队列中的数据进行动态排序**。向优先队列中插入或删除元素时，**优先队列会动态的调整，以保证队列有序**。



`priority_queue` 创建方式与 `queue` 相同，只是在创建 `priority_queue` 时，可以指定优先规则，即最后一个模版参数可以是一个函数对象，指定元素排序规则。创建 `priority_queue` 的示例如下：

```
priority_queue<int , vector<int> ,greater<int> > pq;
```

11.5 迭代器

STL提供了五种基本的迭代器：输入迭代器、输出迭代器、前向迭代器、双向迭代器和随机迭代器。

11.5.1 输入迭代器与输出迭代器

输入和输出迭代器是最基本、要求最低的迭代器，几乎所有的迭代器都具备这两个迭代器的功能。

1. 输入迭代器

输入迭代器(Inputiterator)，**只能一次一个的向前读取元素**，并按此顺序传回元素值，是不可重复的单向遍历。

方法	含义
*itr	读取实际元素
itr->member	读取实际元素的成员
++itr	向前前进一个单位，传回新的位置
itr++	向前前进一个单位，传回旧的位置
itr1==itr/itr!=itr2	判断迭代器是否相等
itr1=itr2	赋值迭代器构造器
拷贝构造	复制迭代器

输入迭代器**只能读取元素一次**，移动到迭代器到下一个位置后，不能保证之前的迭代器还有效。一个典型例子就是输入流迭代器，即从标准输入设备读取数据的迭代器，同一个值不会被读取两次，一旦从输

入流读入一个数据后，下次再读取时会传回另一个值。如果要复制输入迭代器，原有的迭代器和新产生的副本都向前移动，那么这两个迭代器会读取不同的值。

```C++
#include <iostream>
#include <iterator>
#include <algorithm>

using namespace std;

int main() {
    cout << "请输入一些整数，以空格分隔，输入非整数结束输入：" << endl;
    // 创建标准输入流迭代器
    istream_iterator<int> input_it(cin);
    istream_iterator<int> end_it;

    // 遍历输入，直到遇到非整数输入
    while (input_it != end_it) {
        cout << "读取到的整数: " << *input_it << endl;
        ++input_it;
    }

    return 0;
}
```

如果有两个输入迭代器 `it1` 和 `it2`，且有 `it1==it2`，但这并不保证 `++it11==++it2`，更不能保证 `*(++it1) == *(++it2)`，因此用输入迭代器读入的序列不能保证是可重复的。

2. 输出迭代器

输出迭代器(Outputiterator)与输入迭代器**相反**，其作用是将元素值逐个写入，即只能逐个元素赋值。输出迭代器也支持对序列进行单向遍历，当把迭代器移到下一个位置后，也不能保证之前的迭代器是有效的。

方法	含义
*itr=val	将元素写入到迭代器位置
++itr	向前前进一个单位，传回新的位置
itr++	向前前进一个单位，传回旧的位置

itr1=itr2	赋值迭代器构造器
拷贝构造	复制迭代器

11.5.2 前向迭代器

前向迭代器(Forwarditerator)是输入迭代器和输出迭代器的**集合**，具有输入迭代器和输出迭代器的**全部功能**。前向迭代器支持对序列进行可重复的单向遍历，可以多次解析一个迭代器指定的位置，因此可以对一个值进行多次读写。

前向迭代器去掉了输入迭代器与输出迭代器的一些不确定性，例如，如果有两个前向迭代器 `it1` 和 `it2` ，且有 `it1==it2` ，那么 `++it11==++it2` 一定是成立的，这就意味着，前后两次使用相等的前向迭代器读取同一个序列，只要序列的值在这个过程中没有被改写，就一定会得到相同的结果，因此前向迭代器对序列的遍历是**可重复的**。

11.5.3 双向迭代器与随机访问迭代器

双向迭代器是在单向迭代器的基础上增加了一个反向操作，就是它**既可以前进，又可以后退**，因此它比单向迭代器新增一个功能，**进行自减操作**，例如 `it--` 或者 `--it` 。

随机访问迭代器在双向迭代器的基础上，又支持直接将迭代器向前或向后移动 `n` 个元素，而且还支持比较运算的操作，因此随机访问迭代器的功能几乎和指针一样。

方法	含义
itr[n]	获取索引位置为n的元素
itr+=n itr−=n	向前或者向后跳n个元素的位置
(1)itr+n (2)n+itr (3)itr−n	(1)返回itr后第n个元素的位置 (2)返回itr后第n个元素的位置 (3)返回itr之前第n个元素的位置

itr1–itr2	返回itr1与itr2之间的距离
itr1<itr2 itr1>=itr2	判断itr1是否在itr2之前
itr[n]	获取索引位置为n的元素
itr+=n itr–=n	向前或者向后跳n个元素的位置
(1)itr+n (2)n+itr (3)itr–n	(1)返回itr后第n个元素的位置 (2)返回itr后第n个元素的位置 (3)返回itr之前第n个元素的位置
itr1–itr2	返回itr1与itr2之间的距离

若在实战中使用到迭代器相关内容，我们将**特别篇——深究STL**中再探讨，以上内容有印象即可。

11.6 算法

在学习STL算法时，不必知道算法时如何设计的，但需要知道如何在程序中使用这些算法。

11.6.1 算法概论

1. 算法的头文件

STL中提供的所有算法都包含在三个头文件中：`<algorithm>`、`<numeric>`、`<functional>`。

- `algorithm`是最大的一个头文件，它由一大堆函数模板组成，其中涉及到的功能有比较、交换、查找、遍历、复制、修改、删除、合并、排序等。
- 头文件`numeric`很小，只包括几个在序列中进行简单数学运算的函数模板，以及加法和乘法在序列中的一些操作。

- 头文件 `functional` 中则定义了一些类模板，用于声明一些函数对象。

2. 算法的分类

STL中的算法大致可分为4类：

（1）**不可变序列算法**：此类算法不改动容器中元素的次序，也不改动元素值，一般通过 `input` 迭代器和 `forward` 迭代器完成工作。不可变序列算法可用于所有的标准容器。

（2）**可变序列算法**：此类算法一般不直接修改容器中的元素值，它有可能在将元素复制到另一区间的过程中改变元素的值。可变序列算法还包括了移除性质和删除性质的算法，移除一般只是在逻辑上“移除”元素，不改变容器的大小和容器中元素的个数，“移除”和“删除”是不同的算法。

（3）**排序算法**：STL中一系列算法都与排序有关，其中包括对序列进行排序、合并的算法、搜索算法、有序序列的集合操作以及堆操作相关算法，所有这些算法都是通过对序列元素的比较操作完成的。排序一般是通过对容器中元素的赋值和交换，改变元素顺序。排序算法的复杂度通常低于线性算法，要运用随机存取迭代器。

（4）**数值算法**：数值算法主要是对容器中的元素进行数值计算，例如，区间内容的累积、相邻元素差等。

11.6.2 常用的算法

1. for_each()算法

`for_each()` 属于非可变序列算法，此算法非常灵活，可以同时处理**修改每一个元素**，其函数定义如下所示：

```
template<typename InputIterator, typename Function>
```

```
for_each(InputIterator begin, InputIterator end, Function func);
```

`for_each()` 算法对 `[begin, end)` 区间中的每个元素都调用 `func` 函数（对象）进行操作，它不会对区间中的元素做任何修改，也不会改变原来序列的元素次序。

2. find()算法

`find()` 也属于非可变序列算法，用于在指定区间查找某一元素**是否存在**，其函数原型如下所示：

```
template<typename InputIterator, typename T>
```

```
InputIterator find(InputeIterator first, InputIterator last, const T& valu
e);
```

`find()` 算法用于在 `[begin, last)` 区间查找 `value` 元素是否存在，如果存在，就返回指向这个元素的迭代器，如果不存在，就返回 `end` 。

3. copy()算法

对于 `copy()` 函数，我们并不陌生，在讲解迭代器几次都用到了 `copy()` 函数，它的功能是完成**元素的复制**，其函数原型如下所示：

```
template<typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator DestBeg);
```

`copy()` 函数实现将 `[first, last)` 区间的元素复制到另一个地方，这个地方的起始位置为 `DestBeg` 。由于在讲解迭代器时，已经多次调用 `copy()` 函数将元素复制到 `cout` 流对象中输出到屏幕，因此这里就不再举例演示 `copy()` 函数的用法。

4. sort()算法

`sort()` 属于可变序列算法，它支持对容器中的所有元素**进行排序**，函数的原型有如下两种形式：

```
template<typename RanIt>                                    //第一种形式
void sort(RanIt first, RanIt last);
template<typename RanIt, typename Pred>                     //第二种形式
void sort(RanIt first, RanIt last, Pred op);
```

其中，第一种形式是默认的按从小到大的顺序排列，第二种形式比第一种形式更加通用，它可以指定排序规则。

`sort()` 算法所使用的迭代区间必须是随机迭代器，因此只能适用于 `vector` 与 `deque` 容器， `list` 容器不支持随机迭代器，不能使用该算法，但 `list` 容器提供了 `sort()` 成员函数，用于自身的元素排序。

5. accumulate()算法

`accumulate()` 算法属于**数值算法**，它的原型如下所示：

```
template<typename InputIterator, typename T>                //第一种形式
T accumulate(InputIterator first, InputIterator last, T t);
template<typename InputIterator, typename T,typename Pred>  //第二种形式
T accumulate(InputIterator first, InputIterator last, T t, Pred op);
```

`accumulate()` 函数的功能是将 `[first, last)` 区间内的数值累加，累加的初始值为 `t` ，它的返回值是元素累加结果。第二种形式可以按照指定的规则将元素相加。

并不是让大家把原型全都记下来，而是**会用即可。**

下面是简单的例子：

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

// 打印元素的函数
void printElement(int element) {
    cout << element << " ";
}

int main() {
    vector<int> numbers = {5, 2, 9, 1, 5, 6};

    // for_each() 算法示例
    cout << "for_each 输出元素: ";
    for_each(numbers.begin(), numbers.end(), printElement);
    cout << endl;

    // find() 算法示例
    auto it = find(numbers.begin(), numbers.end(), 9);
    if (it != numbers.end()) {
        cout << "找到了元素 9, 位置是: " << (it - numbers.begin()) << endl;
    } else {
        cout << "未找到元素 9" << endl;
    }

    // copy() 算法示例
    vector<int> copiedNumbers(numbers.size());
    copy(numbers.begin(), numbers.end(), copiedNumbers.begin());
    cout << "复制后的元素: ";
    for_each(copiedNumbers.begin(), copiedNumbers.end(), printElement);
    cout << endl;

    // sort() 算法示例
    sort(numbers.begin(), numbers.end());
    cout << "排序后的元素: ";
    for_each(numbers.begin(), numbers.end(), printElement);
    cout << endl;

    // accumulate() 算法示例
    int sum = accumulate(numbers.begin(), numbers.end(), 0);
    cout << "元素的总和是: " << sum << endl;

```

```
46         return 0;
47     }
48
```