C++迭代器（STL迭代器）iterator详解

原创 @最爱吃兽奶 于 2023-02-25 15:07:34 发布

阅读量1.1w 收藏 421 点赞数 62

分类专栏： C++ 文章标签： c++ 开发语言

要访问顺序容器和关联容器中的元素，需要通过“迭代器（iterator）”进行。迭代器是一个变量，相当于容器和操纵容器的算法之间的中介。迭代器可以指向容器中的某个元素，通过迭代器就可以读写它指向的元素。从这一点上看，迭代器和指针类似。

迭代器按照定义方式分成以下四种。

1) 正向迭代器，定义方法如下：

容器类名::iterator 迭代器名;

2) 常量正向迭代器，定义方法如下：

容器类名::const_iterator 迭代器名;

3) 反向迭代器，定义方法如下：

容器类名::reverse_iterator 迭代器名;

4) 常量反向迭代器，定义方法如下：

容器类名::const_reverse_iterator 迭代器名;

迭代器用法示例

通过迭代器可以读取它指向的元素，*迭代器名就表示迭代器指向的元素。通过非常量迭代器还能修改其指向的元素。

迭代器都可以进行++操作。反向迭代器和正向迭代器的区别在于：

- 对正向迭代器进行++操作时，迭代器会指向容器中的后一个元素；
- 而对反向迭代器进行++操作时，迭代器会指向容器中的前一个元素。

下面的程序演示了如何通过迭代器遍历一个 vector 容器中的所有元素。

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main()
5  {
6      vector<int> v;  //v是存放int类型变量的可变长数组，开始时没有元素
7      for (int n = 0; n<5; ++n)
8          v.push_back(n);  //push_back成员函数在vector容器尾部添加一个元素
9      vector<int>::iterator i;  //定义正向迭代器
10     for (i = v.begin(); i != v.end(); ++i) {  //用迭代器遍历容器
11         cout << *i << " ";  //*i 就是迭代器i指向的元素
12         *i *= 2;  //每个元素变为原来的2倍
13     }
14     cout << endl;
15     //用反向迭代器遍历容器
16
   for (vector<int>::reverse_iterator j = v.rbegin(); j != v.rend(); ++j)
17         cout << *j << " ";18     return 0;
19 }
```

程序的输出结果是：
0 1 2 3 4
8 6 4 2 0

第 6 行，vector 容器有多个构造函数，如果用无参构造函数初始化，则容器一开始是空的。

第 10 行，begin 成员函数返回指向容器中第一个元素的迭代器。++i 使得 i 指向容器中的下一个元素。end 成员函数返回的不是指向最后一个元素的迭代器，而是指向最后一个元素后面的位置的迭代器，因此循环的终止条件是`i != v.end()`。

第 16 行定义了反向迭代器用以遍历容器。反向迭代器进行++操作后，会指向容器中的上一个元素。rbegin 成员函数返回指向容器中最后一个元素的迭代器，rend 成员函数返回指向容器中第一个元素前面的位置的迭代器，因此本循环实际上是从后往前遍历整个数组。

如果迭代器指向了容器中最后一个元素的后面或第一个元素的前面，再通过该迭代器访问元素，就有可能导致程序崩溃，这和访问 NULL 或未初始化的指针指向的地方类似。

第 10 行和第 16 行，写++i、++j相比于写i++、j++，程序的执行速度更快。回顾++被重载成前置和后置运算符的例子如下：

```
1  CDemo CDemo::operator++ ()
2  {  //前置++
3      ++n;
4      return *this;
5  }
6  CDemo CDemo::operator ++(int k)
7  {  //后置++
8      CDemo tmp(*this);  //记录修改前的对象
9      n++;
10     return tmp;  //返回修改前的对象
11 }
```

后置++要多生成一个局部对象 tmp，因此执行速度比前置的慢。同理，迭代器是一个对象，STL 在重载迭代器的++运算符时，后置形式也比前置形式慢。在次数很多的循环中，++i和i++可能就会造成运行时间上可观的差别了。因此，本教程在前面特别提到，对循环控制变量 i，要养成写++i、不写i++的习惯。

注意，容器适配器 stack、queue 和 priority_queue 没有迭代器。容器适配器有一些成员函数，可以用来对元素进行访问。

迭代器的功能分类

不同容器的迭代器，其功能强弱有所不同。容器的迭代器的功能强弱，决定了该容器是否支持 STL 中的某种算法。例如，排序算法需要通过随机访问迭代器来访问容器中的元素，因此有的容器就不支持排序算法。

常用的迭代器按功能强弱分为输入、输出、正向、双向、随机访问五种，这里只介绍常用的三种。

1) 正向迭代器。假设 p 是一个正向迭代器，则 p 支持以下操作：++p，p++，*p。此外，两个正向迭代器可以互相赋值，还可以用==和!=运算符进行比较。

2) 双向迭代器。双向迭代器具有正向迭代器的全部功能。除此之外，若 p 是一个双向迭代器，则--p和p--都是有定义的。--p使得 p 朝和++p相反的方向移动。

3) 随机访问迭代器。随机访问迭代器具有双向迭代器的全部功能。若 p 是一个随机访问迭代器，i 是一个整型变量或常量，则 p 还支持以下操作：

- p+=i：使得 p 往后移动 i 个元素。
- p-=i：使得 p 往前移动 i 个元素。
- p+i：返回 p 后面第 i 个元素的迭代器。
- p-i：返回 p 前面第 i 个元素的迭代器。
- p[i]：返回 p 后面第 i 个元素的引用。

此外，两个随机访问迭代器 p1、p2 还可以用 <、>、<=、>= 运算符进行比较。p1<p2的含义是：p1 经过若干次（至少一次）++操作后，就会等于 p2。其他比较方式的含义与此类似。

对于两个随机访问迭代器 p1、p2，表达式p2-p1也是有定义的，其返回值是 p2 所指向元素和 p1 所指向元素的序号之差（也可以说是 p2 和 p1 之间的元素个数减一）。

表1所示为不同容器的迭代器的功能。

表1：不同容器的迭代器的功能

容器	迭代器功能
vector	随机访问
deque	随机访问
list	双向
set / multiset	双向
map / multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

例如，vector 的迭代器是随机迭代器，因此遍历 vector 容器有以下几种做法。下面的程序中，每个循环演示了一种做法。

【实例】遍历 vector 容器。

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v(100); //v被初始化成有100个元素
    for(int i = 0;i < v.size() ; ++i) //size返回元素个数
        cout << v[i]; //像普通数组一样使用vector容器
    vector<int>::iterator i;
    for(i = v.begin(); i != v.end (); ++i) //用 != 比较两个迭代器
        cout << * i;
    for(i = v.begin(); i < v.end ();++i) //用 < 比较两个迭代器
        cout << * i;
    i = v.begin();
    while(i < v.end()) { //间隔一个输出
        cout << * i;
        i += 2; // 随机访问迭代器支持 "+= 整数"  的操作
    }
}
```

list 容器的迭代器是双向迭代器。假设 v 和 i 的定义如下：

```
list<int> v;
list<int>::const_iterator i;
```

则以下代码是合法的：

```
for(i=v.begin(); i!=v.end(); ++i)
cout << *i;
```

以下代码则不合法：

```
for(i=v.begin(); i<v.end(); ++i)
cout << *i;
```

因为双向迭代器不支持用“<”进行比较。以下代码也不合法：

```
for(int i=0; i<v.size(); ++i)
cout << v[i];
```

因为 list 不支持随机访问迭代器的容器，也不支持用下标随机访问其元素。

在 C++ 中，数组也是容器。数组的迭代器就是指针，而且是随机访问迭代器。例如，对于数组 int a[10]，int * 类型的指针就是其迭代器。则 a、a+1、a+2 都是 a 的迭代器。

迭代器的辅助函数

STL 中有用于操作迭代器的三个函数模板，它们是：

- advance(p, n)：使迭代器 p 向前或向后移动 n 个元素。
- distance(p, q)：计算两个迭代器之间的距离，即迭代器 p 经过多少次 + + 操作后和迭代器 q 相等。如果调用时 p 已经指向 q 的后面，则这个函数会陷入死循环。
- iter_swap(p, q)：用于交换两个迭代器 p、q 指向的值。

要使用上述模板，需要包含头文件 algorithm。下面的程序演示了这三个函数模板的 用法。

```
#include <list>
#include <iostream>
#include <algorithm> //要使用操作迭代器的函数模板，需要包含此文件
using namespace std;
int main()
{
    int a[5] = { 1, 2, 3, 4, 5 };
    list <int> lst(a, a+5);
    list <int>::iterator p = lst.begin();
    advance(p, 2);  //p向后移动两个元素，指向3
```

程序的输出结果是：
1) 3
2) 2
3) 3
4) 1 5 3 4 2