

 码龄6年  暂无认证


104
原创


1905
积分


 


[私信](#)

热门文章

关于C/C++ stdin缓冲区以及对字符输入的一些经验和心得  5775

数据结构与算法专题之串——字符串及KMP算法  4275

UVA 11809 Floating-Point Numbers（浮点数）  3282

UVA 508 Morse Mismatches（莫尔斯电码）  3167

SDUT 2055 来溜博旅游  2446

最新评论

数据结构与算法专题之线性表——找及其...

Tisfy: 深得人心，正如古人云：忆昔霍将军，

，连年此征讨。

数据结构与算法专题之串——字符串及K...

qq_33360282: 讲的好好哦，但是我有个小

小的疑问: 就是您的next值是不用+1的 ...

POJCodeSubmitter_Update

karmalk: 请问博主这个软件是使用Python+

PyQt写的么？代码能开源学习一下么

UVa 11809 Floating-Point Numbers（浮...

wx17777: 老哥你在那找到的

POJCodeSubmitter_Update

xtttgo: 博主大好人，再也不用忍受POJ那速

度了！赞一个

最新文章

数据结构与算法专题之图——最短路径算法

数据结构与算法专题之图——欧拉回路与拓扑排序

数据结构与算法专题之图——连通分量与最小生成树

2017年 21篇	2016年 7篇
2015年 64篇	2014年 13篇

原创 ShannonNansen 2017-08-16 18:47:05 673 收藏 版权

分类专栏: 2017年暑假集训 算法模板 心得体会 文章标签: C++ stl

重载运算符，是C++语言特色之一。对于构造数据类型来说，通过运算符的重载，可以使程序代码更加简洁清晰，功能更加丰富。

本文不过多地介绍运算符重载和STL，只是介绍一下STL有序容器与重载运算符之间的一点小应用。下面的代码我都简单写了，实际上应该做好封装的。

重载运算符

为什么要重载运算符

1. 为了代码书写方便

比方说，我们定义一个**复数类**，由于复数类是我们自己构造的数据类型，它是无法简单地通过+来实现复数加法的，因为编译器和计算机根本不知道咋加，所以我们需要手动实现一个**add()**方法来加，但是在C++中，我们可以通过重载运算符，来实现直接使用+连接两个复数来进行加法运算。假设有复数类（结构体）：

```
1 struct Complex
2 {
3     int a,b;
4 };
```

我们用a,b分别代表复数的实部和虚部，我们知道复数相加实际就是实部和虚部分别相加，我们先来写一下add方法吧：

[点赞](#)
[评论](#)
[分享](#)
[收藏](#)
[举报](#)
[关注](#)
[一键三连](#)

```
3 |     Complex res;  
4 |     res.a = v1.a + v2.a;  
5 |     res.b = v1.b + v2.b;  
6 |     return res;  
7 | }
```

这样，我们就可以通过add方法来实现两个复数相加得到一个新的复数了。

当然更简单的方法是重载 '+' 加号：

```
1 Complex& operator + (const Complex &v1, const Complex &v2)
2 {
3     Complex res;
4     res.a = v1.a + v2.a;
5     res.b = v1.b + v2.b;
6     return res;
7 }
```

可以看到，内部代码其实是一样的，只不过头部的声明有变化而已，其实它也是个函数，实际上就是告诉编译器，我要重载+加号，它的两边都是Complex类型，并且他们运算后的结果也是Complex类型，然后函数体里定义了他们的运算过程。这样我们就可以直接使用+来连接两个复数进行运算了。

试想一下，假设有Complex a, b, c;，那么c = a.add(b);和c = a + b;哪一段代码更直观易懂呢？答案肯定是后者。

当然我们也可以重载减号啊乘号啊等等.....当然，我们也可以定义‘+’为加法运算，‘*’为乘法运算，反正只要你开心，怎么定义都可以，不必拘泥于运算符本来的含义，重载嘛，本来就是给符号赋予新的生命。

就像我们日常书写中，习惯把“^”符号当做“幂”，而在计算机中这个符号却是“异或”，当然对于基本数据类型（int, double, char, long, float等）来说，编译器已经把它们的符号定义固定死了，我们无法更改，但是对于构造类型，我们有绝对的决定权，所以我们可以重载运算符让某些符号实现我们自己想要的功能。

目录

```

□□□□
    □□□□□□□□
STL□□□□□□□
    □□
    sort□□
    priority_queue□□□□
    □□

```

分类专栏

拿线性代数中的矩阵来说，我们知道矩阵是有幂运算的，那么假设有矩阵**a**，要求**a**的**b**次方，我们在程序中直接写**a^b**肯定是不行的，但是我们可以重载呀，假设有类**Matrix**（就不详细写了），我们可以这样写运算符重载的头部：

```
1 Matrix& operator ^ (Matrix &m, int &n)
2 {
3     .....
4 }
```

这样重载过后，我们假设有**Matrix a**；**int b**；然后我们就可以**a^b**来得到**a**矩阵的**b**次幂了。有兴趣的可以看一下我之前写过的矩阵快速幂的模板，里面就重载了两矩阵的基本操作：[传送门>>](#)

2. 为了使用有序容器或排序

这点类似java里实现Comparator接口来使对象可入有序容器。对于内置数据类型如int来说，它的大小关系是**确定的**，比如1和6，我们明确地知道1小6大，计算机、编译器也都知道。但是对于构造类型来说，比如构造一个学生类，问学生**a**和学生**b**谁大谁小？What？什么大？**年龄？身高？体重？XX？**所以计算机编译器无法确定构造类型的大小关系，他们的关系，是创造这个类型的编码人来定义确定的，所以这时候我们就需要通过书写**方法**或者**重载运算符**来告知编译器该如何确定他们的大小关系，例如下面这个结构体（类）：

```
1 struct student
2 {
3     int age;
4     int height;
5     int weight;
6 };
```

那么，我们认为，通过年龄来定义学生的大小，年龄相等的，就通过身高比较，身高相同的，就通过体重比较。那么，如果我们通过方法体来写，应该这样：

```
1 int stucmp(student a, student b)
2 {
3     if(a.age != b.age)
4         return a.age < b.age ? -1 : 1;
5     if(a.height != b.height)
6         return a.height < b.height ? -1 : 1;
7     if(a.weight != b.weight)
8         return a.weight < b.weight ? -1 : 1;
9     return 0;
10 }
```

类比**strcmp**，差不多就是这个道理，我们可以通过调用该函数来比较出学生的大小，当然更直观简便的方式是重载运算符，看下面：

```
1 bool operator < (const student &a, const student &b)
2 {
3     if(a.age != b.age)
4         return a.age < b.age;
5     if(a.height != b.height)
6         return a.height < b.height;
7     return a.weight < b.weight;
8 }
```

跟上面的代码类似，看起来也很像一个函数，对于这个函数，我们有三点需要注意。

第一，**返回值**：

可以看出，这里返回值是**bool**，布尔型，我们知道，对于比较运算符来说，它所组成的表达式的结果是**bool**值，比如1<2, **true**, 6<3, **false**，所以重载'<'，返回值得是**bool**。

第二，**运算符**：

要重载哪个运算符，就要写上**operator**关键字，然后跟上要重载的算符。

第三，**参数列表**：

我们知道，小于号是**双目运算符**，那么我们参数列表里就需要**两个参数**，**从左到右分别代表了小于号的左右操作数**，也就是说，我们声明了这个“函数”，实际上就是告诉编译器，我们要告诉你**a<b**是如何定义的。

如果这个“函数”返回值时**true**，代表小于号成立；否则就是不成立。这样重载完成后，假设有**student a,b;**，我们就可以简单地通过**a<b**来得到**ab**的大小关系了。

当我们重载了比较运算符后，我们**C++STL**内置的一些有序容器就可以使用这些构造类型了，因为如果你不重载规定的比较符的话，编译器无法得知构造类型的大小关系，自然就不存在有序这一说，只有重载了比较运算符，编译器才可以通过你重载的运算符确定大小关系，自然就可以实现有序。这些下面会继续介绍。

C++可重载的运算符很多，我们可以根据自己实际的需要来重载运算符，其实重载运算符是相当灵活的一个功能，我们可以随意定义它的返回值和功能，但是不能改变它的单双目性质。不可以重载基本数据类型的运算符。

STL有序容器和算法

算子

在介绍它们之前，我们先介绍两种重要的算子类，**less**和**greater**，它们是**泛型的**。我们先来看一下它们两个的头文件里的原型：

```
1  /// One of the @link comparison_function comparison functors@endlink.
2  template<typename _Tp>
3      struct greater : public binary_function<_Tp, _Tp, bool>
4      {
5          bool
6          operator()(const _Tp& __x, const _Tp& __y) const
7              { return __x > __y; }
8      };
9
10  /// One of the @link comparison_function comparison functors@endlink.
11  template<typename _Tp>
12      struct less : public binary_function<_Tp, _Tp, bool>
13      {
14          bool
15          operator()(const _Tp& __x, const _Tp& __y) const
16              { return __x < __y; }
17      };
```

可以看出，它们两个实际上是两个 **已经封装好** 的大小比较而已，至于它们内部为什么要重载 **()** 一对括号，这是**STL**内部的事情，为了使用起来可以像函数一样（下面会用例子解释到），我们现在只需要知道，**less**封装了小于，**greater**封装了大于。

通过原型也不难看出，想要使用这两个泛型类，就必须重载对应的运算符，比如，拿上面的**student**来说，我们已经重载了小于，所以我们可以通过**less<student> a;**来声明一个**less**对象**a**，但是**greater<student> b;**就会报错：no match for 'operator' (operand types are 'const student' and 'const student')，明显可以看出来，**student**缺少对大于号的重载。

那么这两个类到底有啥子用？为什么要把比较封装起来？直接大于小于搞起来不好吗？

好，问得好，那么我们继续看下面的内容……

sort算法

这是我们比较常用的一个排序算法，由于内部根据数据的组织形式灵活实现使用各种排序，所以性能已经是相当好的了。它的使用也相当简单，传入两个参数，容器的开始指针（或迭代器）和容器的结束指针（或迭代器）即可。当然也可以对数组排序，对于数组来说，就是传入**首地址**和**最后一个元素的下一位置地址**。比如需要对**int s[100]**整个数组排序，只需**sort(s, s + 100)**，执行完后**s**即变为有序。

sort默认是升序排序，也就是小的在前，如果我们要降序怎么办呢？别急，**sort**还有三个参数的版本，看一下原型：

```
1  template<typename _RAIter, typename _Compare>
2      void
```

```
3 |         sort(_RAIter, _RAIter, _Compare);
```

我们看到第三个参数叫_Compare，顾名思义，也就是使用_Compare对象进行比较排序，我们给他取个不专业的名字“比较器”，也就是我们上面提到的less和great（其实还有很多，比如等于equal_to，大于等于greater_equal，小于等于less_equal等），当然，这个_Compare也是可以自己定义的，只要能体现出大小比较，就可以（可以是 **返回值为bool型的两个参数的函数**，也可以是 **重载了>()双括号参数为2个且返回值为bool的一个类**）。默认升序，也就是小的在前，使用了less，如果我们要升序排序，自然使用greater，那么我们上面对于s[100]的排序就变成了sort(s, s + 100, greater<int>());，注意，因为我们在给sort传参数，所以第三个参数要传变量进去，所以不要写成greater<int>而落掉后面的双括号，因为写了括号是声明，不写括号就单纯的是个类型名（注意 **greater<int>** 和 **greater<int>()** 的区别）。

同样地，我们要对学生stu[100]排序的话，要**升序排序，就要重载小于号**，调用sort(stu, stu+100)或者sort(stu, stu+100, less<student>()); 如果要**降序，则需要重载大于号**并调用sort(stu, stu+100, greater<student>())。

priority_queue优先队列

对于优先队列，这里没什么要讲的，我们只讲使用。优先队列是有序的容器，队头始终是整个队列中**最大或最小**的元素，既然提到了最大最小，所以对于队列中的元素，一定要有确定的大小关系。优先队列原型部分如下：

```
1 | template<typename _Tp, typename _Sequence = vector<_Tp>,
2 |         typename _Compare = less<typename _Sequence::value_type> >
3 |     class priority_queue
4 |     {
5 |
6 |         .....
7 |
8 |     };
```

可以看出，泛型部分的声明，第二部分和第三部分都有默认值，分别是vector和less，这说明优先队列默认情况下时用vector组织，并且以less为比较运算规则，也就是较大元素在前，这点要区别于sort排序。如下：

```
1 |     priority_queue<int> q1; // 大的在前
2 |     priority_queue<int, vector<int>, less<int> > q2; // 大的在前，要注意最后<int>与>这里要隔开一个空格，不然会被当做右移符号
3 |     priority_queue<int, vector<int>, greater<int> > q3; // 小的在前
4 |     priority_queue<int, greater<int> > q4; // 报错，这样写编译器会认为greater是第二参数，不匹配
```

如果我们要使得优先队列变成较小元素在前，自然就要使第三个参数变成greater。注意，按照参数默认值的规则，第二第三参数**要么写2不写3，要么都不写，要么都写，不存在**只写3而不写2的（因为计算机不知道你写的到底是2还是3，它只认从左到右依次取参数的顺序）。

可以看出，less和greater决定了有序容器和算法的排序规则，它们是两个泛型类，也是用来比较两数据的工具，使用它们之前必须先对源数据模型进行小于号和大于号的重载，这与java中的Comparator泛型接口是一个道理的，只有我们对构造类型定义清楚了它的比较规则，我们的其他算法或泛型容器才可以正确处理我们的数据。

总结

算子那部分的问题，现在已经可以解答了。我们可以看出，stl给我们提供了很多的算法和容器，我们要使得stl更加灵活，就要指定它的一些属性，那属性是如何指定的？无非两种，一种是在**声明的时候指定泛型类型**，如上面的优先队列；一种是在**调用的时候指定参数**，如sort排序。而这两种方式**一个要求指定类型，一个要求传入实参**，而如果我们要指定排序规则，难不成要指定一个单纯的小于号或者传入一个小于号？这显然是不符合编程规范的，如果我们以字符传入，那么可能会导致运行时错误（比如传入非法字符），如果我们以符号传入，好像没有这种操作。所以我们要将其封装成类，在需要的时候**指定其类名**或者**声明一个对应的对象**。

下面我手写一个使用了less或greater等内置比较器的泛型冒泡排序，可能会有助于大家理解本章内容：

```
1 | template <typename _Iter, typename _Compare>
2 | void bubble_sort(_Iter first, _Iter last, _Compare compare)
3 | {
4 |     for(int i = 0 ; i < (int)(last - first) ; i++)
5 |     {
```

```
6         for(_Iter j = first ; j < (_Iter)(last - i - 1) ; j++)
7         {
8             if(!compare(*j,*j + 1)) // 这里使用比较器，像调用函数一样
9             {
10                 swap(*j, *(j + 1)); // 交换两数
11             }
12         }
13     }
14 }
```

这样我们如果要对int数组排序，可以这样：

```
1     int s[] = {1,4,3,8,6,5,3,0};
2     bubble_sort(s, s + 8, less<int>()); // 升序排序
3     bubble_sort(s, s + 8, greater<int>()); // 降序排序
```

看到上面的排序代码比较那部分，现在可以知道为什么less和greater原型里要重载 () 双括号了吧？就是为了让less和greater 的实例能像函数一样用括号括起来使用，像这样 less<int> cmpLess; cmpLess(a,b) 或 greater<int> cmpGreater; cmpGreater(a,b)，这样的好处是，_Compare可以使用函数来自定义，这样就算_Compare传入的是个函数，也可以正常执行，例如：

```
1 bool cmp(int a, int b)
2 {
3     return a > b;
4 }
```

然后排序的时候就可以这样写：

```
bubble_sort(s, s + 8, cmp); // 自定义函数的降序
```

这样就更加灵活，可以直接通过函数来声明排序的大小关系优先级，也是比较方便的。

也可以手写一个类似less和greater的类，重载一下()双括号：

```
1 struct cmpGreater
2 {
3     bool operator () (const int &a, const int &b) const
4     {
5         return a > b;
6     }
7 };
```

然后排序可这样写：（注意细节）

```
bubble_sort(s, s + 8, cmpGreater()); // 注意双括号！
```

跟上面一样，这里是 函数调用，是传参数，传实参，参数要么是 函数名（函数指针），要么是 变量（对象），所以不加双括号就变成了类型，是不合法的，加了括号才是个变量（对象）。

本章内容可能比较抽象，加上感觉我自己讲的也比较乱糟糟……是临时插入的一章，感觉在日常编写C++代码时，STL用的还是比较多的。

如有表述不明或错误的地方，欢迎指正和交流。也欢迎大家继续跟进数据结构与算法的相关学习博客~