第八周 标准模板库 STL（一）

第一节 string 类

1.关于 string 类

string 类是模板类：typedef basic_string<char>string;实例出来的类。使用 string 类要包含头文件<string>。

string 对象的初始化有以下几种类型：

```
string s1(“Hello);
string month = “March”;
string s2(8,’x’);
```

以下的初始方法错误：

```
string error1 = 'c';//error
string error2('u');//error
string error3 = 22//error
string error4(22);//error
```

但是可以将字符赋值给 string 对象，如下：

```
string s;
s = 'n';//OK
```

string 对象的长度用成员函数 length()读取，如下：

```
string s(“hello”);
cout<<s.length()<<endl;
```

string 支持流读取运算符，如下：

```
string stringObject;
cin>> stringObject;
```

string 支持 getline 函数：

```
string s;
getline(cin,s);
```

2.string 的赋值和连接

（1）用=赋值：

```
string s1(“cat”),s2;
s2 = s1;
```

（2）用 assign 成员函数复制

```
string s1(“cat”),s3;
s3.assign(s1);
```

（3）用 assign 成员函数部分复制

```
string s1(“catpig”),s3;
s3.assign(s1,1,3);//从 s1 中下标为 1 的字符开始复制 3 个字符给 s3
```

（4）单个字符复制

```
s2[5]=s1[3]=’a’;
```

（5）逐个访问 string 对象中的字符

```
string s1("Hello");
for(int i = 0;i<s1.length();i++)
    cout<<s1.at(i)<<endl;
```

成员函数 at 会做范围检查，如果超出范围，会抛出 out_of_range 异常，而下标运算符[]不做范围检查。

（6）用+运算符连接字符串

```
string s1(“good”),s2(“morning”);
s1+=s2;
cout<<s1;//输出 goodmorning
```

（7）用成员函数 append

```
s2.append(s1,3,s1.size());//s1.size()为 s1 的字符数
```

下标为 3 开始，s1.size()个字符数，如果字符串内没有足够字符，则复制到字符串最后一个字符。

3.比较 string

（1）用关系运算符比较 string 的大小：==/>/>=/</<=/!=

返回值都是 bool 类型，按照字典序。

（2）用成员函数 compare 比较 string 的大小

```
string s1("hello"),s2("hello"),s3("hell");
int f1 = s1.compare(s2);//0
int f2 = s1.compare(s3);//1
int f3 = s3.compare(s1);//-1
int f4 = s1.compare(1,2,s3,0,3);//-1,比较 s1 的 1-2 与 s3 的 0 到 3
int f5 = s1.compare(0,s1.size(),s3);//1，比较 s1 和 s3 的 0-end
```

4.成员函数

（1）成员函数 substr

```
string s1("hello world"),s2;
s2 = s1.substr(4,5);//下标 4 开始 5 个字符
cout << s2 <<endl;
```

输出：o wor

（2）成员函数 swap

```
string s1("hello"),s2("really");
s1.swap(s2);
```

（3）成员函数 find()

```
string s1("hello world");
s1.find("lo");
```

在 s1 中从前向后查找 lo 第一次出现的地方，如果找到则返回 lo 开始的位置（即 l 的下标）；如果找不到，返回 string::npos（string 定义的静态常量）。

（4）成员函数 rfind()

```
string s1("hello world");
s1.rfind("lo");
```

在 s1 中从后向前查找 lo 第一次出现的地方，如果找到则返回 lo 开始的位置（即 l 的下标）；如果找不到，返回 string::npos（string 定义的静态常量）。

（5）成员函数 find_first_of()

```
string s1("hello world");
s1.find_first_of("abcd");
```

在 s1 中从前向后找 abcd 中任何一个字符第一次出现的地方，如果找到返回字母的位置；如果找不到返回 string::npos。

（6）成员函数 find_last_of()

```
string s1("hello world");
s1.find_last_of("abcd");
```

在 s1 中查找 abcd 中任何一个字符最后一次出现的地方，如果找到，返回找到字母的位置；如果找不到，返回 string::npos。

（7）成员函数 find_first_not_of()

```
string s1("hello world");
s1.find_first_not_of ("abcd");
```

在 s1 中从前向后查找不在 abcd 中的字母第一次出现的地方，如果找到，返回找到字母的位置（在此例中为 h，返回 0）；如果找不到，返回 string::npos。

（8）成员函数 find_last_not_of()

```
string s1("hello world");
s1.find_last _not_of ("abcd");
```

在 s1 中凑后往前查找不在 abcd 中的字母第一次出现的地方，如果找到，返回找到字母的位置（此处为 l，返回地址为 9）；如果找不到，返回 string::npos。

（9）成员函数 erase()。删除 string 中的字符。

（10）成员函数 replace()。替换 string 中的字符。例如：

```
string s1("hello world");
s1.replace(2,3,"haha");
cout << s1;
//将 s1 下标 2 开始的三个字符替换为 haha
```

如果被替换区间大小小于替换长度，那么直接在后面加，如本例输出为 hehaha world

（11）成员函数 insert()。在 string 中插入字符。

例程：

```
string s1("hello world");
string s2("show insert");
s1.insert(5,s2); //  将 s2 插入 s1 下标 5 的位置
cout << s1 << endl;
s1.insert(2,s2,5,3);
//将 s2 中下标 5 开始的 3 个字符插入 s1 下标 2 的位置
cout << s1 << endl;
```

输出结果为：

```
helloshow insert world
heinslloshow insert world
```

（12）成员函数 c_str。转换成 C 语言式 char*字符串

```
string s1(“hello world”);
printf(“%s\n,s1.c_str());//为了兼容 C 语言
// s1.c_str()  返回传统的 const char *  类型字符串，且该字符串以 ‘\0’结尾。
```

（13）成员函数 data()

```
string s1("hello world");
const char * p1=s1.data();
for(int i=0;i<s1.length();i++)
    printf("%c",*(p1+i));
//s1.data()  返回一个 char *  类型的字符串，对 s1  的修改可能会使 p1 出错。
```

（14）成员函数 copy()。字符串拷贝函数。

```
string s1("hello world");
int len = s1.length();
char * p2 = new char[len+1];
s1.copy(p2,5,0);
p2[5]=0;
cout << p2 << endl;
```

// s1.copy(p2,5,0) 从 s1 的下标 0 的字符开始制作一个最长 5 个字符长度的字符串副本并将其赋值给 p2。返回值表明实际复制字符串的长度。

输出结果为：hello

5.字符串流处理

除了标准流和文件流输入输出外，还可以从 string 进行输入输出；

类似 istream 和 osteram 进行标准流输入输出，我们用 istringstream 和 ostringstream 进行字符串上的输入输出，也称为内存输入输出。

用到的头文件：

```
#include <string>
#include <iostream>
#include <sstream>
```

例程 1：字符串输入流 istringstream

```
string input("Input test 123 4.7 A");
istringstream inputString(input);
string string1, string2;
int i;
double d;
char c;
inputString >> string1 >> string2 >> i >> d >> c;
cout << string1 << endl << string2 << endl;
cout << i <<endl << d << endl << c << endl;
long L;
if(inputString >> L)
    cout << "long\n";
else cout << "empty\n";
```

输出结果为：

```
Input test 123 4.7 A empty
```

例程 2：字符串输出流 istringstream

```
ostringstream outputString;
int a = 10;
outputString << "This " << a << " ok" << endl;
cout << outputString.str();
```

输出：This 10 ok

第二节 标准模板库 STL 概述（一）

1.泛型程序设计

C++语言的核心优势之一就是便于软件的重用。C++中有两个方面体现重用：

（1）面向对象的思想：继承和多态，标准类库。

（2）泛型程序设计的思想：模板机制，以及标准模板库 STL。

将一些常用的数据结构（比如链表，数组，二叉树）和算法（比如排序，查找）写成模板，以后则不论数据结构里放的是什么对象，算法针对什么样的对象，则都不必重新实现数据结构，重新编写算法。

标准模板库 (Standard Template Library) 就是一些常用数据结构和算法的模板的集合。有了 STL，不必再写大多的标准数据结构和算法，并且可获得非常高的性能。

2.STL 中的基本概念

容器：可容纳各种数据类型的通用数据结构,是类模板。

迭代器：可用于依次存取容器中元素，类似于指针。

算法：用来操作容器中的元素的函数模板。

算法本身与他们操作的数据的类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用。

举个例子，比如说 int array[100];该数组就是容器，而 int*类型的指针变量就可以作为迭代器，sort 算法可以作用于该容器上，对其进行排序：

sort(array,array+70);

3.容器概述

可以用于存放各种类型的数据（基本类型的变量、对象等）的数据结构，都是类模板，分为三种：

①顺序容器：vector，deque，list。

②关联容器：set，multiset，map，multimap。

③容器适配器：stack，queue，priority_queue。

对象被插入容器中时，被插入的是对象的一个**复制品**。许多算法，比如排序，查找，要求对容器中的元素进行比较，有的容器本身就是排序的，所以，放入容器的对象所属的类，往往还应该重载 == 和 < 运算符。

（1）顺序容器

容器并非排序的，元素的插入位置同元素的值无关。有 vector，deque，list 三种。

①vector 头文件<vector>

动态数组。元素在内存连续存放。随机存取任何元素都能在**常数时间**完成。在尾端增删元素具有较佳的性能(大部分情况下是常数时间)。

②deque 头文件<deque>

双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成（但次于 vector）。在两端增删元素具有较佳的性能（大部分情况下是常数时间）。

③list 头文件<list>

双向链表。元素在内存不连续存放。在任何位置增删元素都能在常数时间完成。**不支持随机存取**。

（2）关联容器

元素是排序的，插入任何元素，都按相应的排序规则来确定其位置。在查找时有非常好的性能，通常以平衡二叉树方式实现，**插入和检索时间都是 O(log(N))**。

①set/multiset 头文件<set>

set 即集合。set 中不允许相同元素，multiset 中允许相同的元素。

②map/multimap 头文件<map>

map 与 set 的不同在于 map 中存放的元素有且仅有两个成员变量，一个名为 first，另一个名为 second，map 根据 first 值对元素进行从小到大排序，并可快速根据 first 来检索元素。

map 同 multimap 的不同在于是否允许相同 first 值的元素。

（3）容器适配器

①stack：头文件 <stack>

栈。是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项。**后进先出**。

②queue 头文件<queue>

队列。插入只可以在尾部进行，删除、检索和修改只允许从头部进行。**先进先出**。

③priority_queue 头文件<queue>

优先级队列。最高优先级元素总是第一个出列。

4.顺序容器和关联容器中都有的成员函数

成员函数	功能
begin	返回指向容器中第一个元素的迭代器。
end	返回指向容器中最后一个元素**后面的**位置的迭代器。
rbegin	返回指向容器中最后一个元素的迭代器。
rend	返回只想容器中第一个元素前面的位置迭代器。
erase	从容器中删除一个或几个元素。
clear	从容器中删除所有元素。

5.顺序容器的常用成员函数

成员函数	功能
front	返回容器中第一个元素的引用。
back	返回容器中最后一个元素的引用。
push_back	在容器末位增加新元素。
pop_back	删除容器模块的元素。
erase	删除迭代器指向的元素(可能会使该迭代器失效)，或删除一个区间，返回被删除元素后面的那个元素的迭代器。

第三节 标准模板库 STL 概述（二）

1.迭代器

（1）基本概念

- 用于指向顺序容器和关联容器中的元素
- 迭代器用法和指针类似
- 有 const 和非 const 两种
- 通过迭代器可以读取它指向的元素
- 通过非 const 迭代器还能修改其指向的元素

（2）迭代器的定义

定义一个容器类的迭代器的方法可以是：

容器名称::iterator 变量名;
或
容器名称::const_iterator 变量名;
访问一个迭代器指向的元素：
*迭代器变量名
迭代器上可以执行 ++ 操作，以使其指向容器中的下一个元素。如果迭代器到达了容器中的最后一个元素的后面，此时再使用它，就会出错，类似于使用 NULL 或未初始化的指针一样。
例程：

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v; //一个存放 int 元素的数组，一开始里面没有元素
    v.push_back(1);v.push_back(2);v.push_back(3);v.push_back(4);
    vector<int>::const iterator i;//常量迭代器
    for(int i = v.begin();i!=v.end;++i)
        cout<<*i<<",";
    cout<<endl;
    //输出结果 1,2,3,4,
    vector<int>::reserve.iterator r;//反向迭代器
    for(r = v.rbegin();r!=v.rend();r++)
        cout<<*r<<",";
    cout<<endl;
    //输出结果：4,3,2,1,
    vector<int>::iterator j;//非常量迭代器
    for(j = v.begin();j!=v.end();j++)
        *j = 100;
    for(i = v.begin();i!=v.end();i++)
        cout<<*i<<",";
    //输出结果：100,100,100,100,
    return 0;
}
```

（3）双向迭代器
若 p 和 p1 都是双向迭代器，则可对 p、p1 可进行以下操作：
++p,p++：使 p 指向容器中下一个元素；
--p,p--：使 p 指向容器的上一个元素；
*p：取 p 指向的元素；
p = p1：赋值；
p==p1,p!=p1：判断是否相等
（4）随机访问迭代器
若 p 和 p1 都是随机访问迭代器，则可对 p、p1 可进行以下操作：
双向迭代器的所有操作
p += i 将 p 向后移动 i 个元素

p -= i 将 p 向向前移动 i 个元素
p + i 值为: 指向 p 后面的第 i 个元素的迭代器
p - i 值为: 指向 p 前面的第 i 个元素的迭代器
p[i] 值为: p 后面的第 i 个元素的引用
p < p1, p <= p1, p > p1, p>= p1
p – p1 : p1 和 p 之间的元素个数

表 3.1 容器和容器上面的迭代器

容器	容器上的迭代器类别
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
queue	不支持
priority_queue	不支持

例程 1：vector 的遍历，deque 也是这样

```
vector<int> v(100);
int i;
for(i = 0;i < v.size() ; i ++)
    cout<<v[i];//根据下标随机访问
vector<int>::const_iterator ii;
for( ii = v.begin(); ii != v.end ();++ii)
    cout << * ii;
for( ii = v.begin(); ii < v.end ();++ii )
    cout << * ii;
```

例程 2：list 的遍历（双向迭代器）

```
list<int> v;
list<int>::const_iterator ii;
for(ii=v.begin();ii!=v.end();++ii)
    cout<<*ii;
```

2.算法

算法就是一个个**函数模板**，大多数在<algorithm> 中定义。STL 中提供能在各种容器中通用的算法，比如查找，排序等。算法通过迭代器来操纵容器中的元素。许多算法可以对容器中的一个局部区间进行操作，因此需要两个参数，一个是起始元素的迭代器，一个是终止元素的后面一个元素的迭代器。比如，排序和查找。有的算法返回一个迭代器。比如 find() 算法，在容器中查找一个元素，并返回一个指向该元素的迭代器。算法可以处理容器，也可以处理普通数组

算法示例：find()

```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
```

first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找区间起点和终点[first,last)。区间的起点是位于查找范围之中的，而终点不是。find 在**[first,last)**查找等于 val

的元素。用 == 运算符判断相等。函数返回值是一个迭代器。如果找到，则该迭代器指向被找到的元素。如果找不到，则该迭代器等于 last。

例程：

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main()  {
    int array[10] = {10,20,30,40};
    vector<int> v;
    v.push_back(1);v.push_back(2);v.push_back(3);v.push_back(4);
    vector<int>::iterator p;
    p = find(v.begin(),v.end(),3);
    if(p!=v.end())
        cout<<*p<<endl;//输出 3
    p = find(v.begin(),v.end(),9)
    if(p == v.end())
        cout<<"not found"<<endl;//输出 not found
    p = find(v.begin+1,v.end()-2,1);
    //查找区间为[2,3)
    if(p!=v.end())
        cout<<*p<endl;//输出结果为 3，因为找不到就返回 v.end()，在这里相当于 v.end()的是 3 这个位置
    int *pp = find(array,array+4,20);//数组名是迭代器
    cout<<*pp<<endl;//输出 20
}
```

3.STL 中的“大”、“小”和“相等”

关联容器内部的元素是从小到大排序的。有些算法要求其操作的区间是从小到大排序的，称为“有序区间算法”，例如 binary_search（二分法查找）；有些算法会对区间进行从小到大（可以自己定义）排序，称为“排序算法”，例如 sort；还有一些其它算法会用到“大”“小”的概念，使用 STL 时，在**缺省**情况下，以下三个说法等价：

（1）x 比 y 小；

（2）表达式“x<y”为真；

（3）y 比 x 大。

有时，“x 和 y 相等”等价于“x==y 为真”，例如在未排序的区间上进行的算法，如顺序查找 find；有时候“x 和 y 相等”等价于“x 小于 y 和 y 小于 x 同时为假”（这里小于也是可以自定义的），例如有序区间算法，如 binary_search，关联容器自身的成员函数 find。

第四节 vector，deque 和 list

1.vector

vector 示例程序 1：

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

```
using namespace std;
template<class T>
void PrintVector( T s, T e){
	for(; s != e; ++s)
	cout << * s << " ";
	cout << endl;
}
int main(){
	int a[5] = {1,2,3,4,5};
	vector<int> v(a,a+5);//将数组 a 的内容放入 v
	cout<<"1)"<<v.end()-v.begin()<<endl;
	//两个随机迭代器可以相减，输出 1)5
	cout<<"2)";
	PrintVector(v.begin(),v.end());
	//2)1 2 3 4 5
	v.insert(v.begin()+2,13);
	cout<<"3)";
	PrintVector(v.begin(),v.end());
	//3)1 2 13 3 4 5
	v.erase(v.begin()+2);
	cout<<"4)";
	PrintVector(v.begin(),v.end());
	//4)1 2 3 4 5
	vector<int> v2(4,100);//v2 有 4 个元素，都是 100
	v2.insert(v2.begin(),v.begin()+1,v.begin()+3);
	cout<<"5)";
	PrintVector(v2.begin(),v2.end());
	//5)2 3 100 100 100 100
	v.erase(v.begin() + 1, v.begin() + 3);
	//删除 v 上的一个区间,即 2,3
	cout << "6) ";
	PrintVector(v.begin(),v.end());
	//6) 1 4 5
	return 0;
}
```

例程 2：用 vector 实现二维数组

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
	vector<vector<int> > v(3);
	//v 有 3 个元素，每个元素都是 vector<int>容器
	for(int i = 0;i<v.size();++i)
```

```
        for(int j = 0;j<4;++j)
            v[i].push_back(j);
    for(int i = 0;i<v.size();++i){
        for(int j = 0;j<v[i].size();++j)
            cout<<v[i][j]<<" ";
        cout<<endl;
    }
    return 0;
}
```

2.deque

所有适用于 vector 的操作都适用于 deque。

deque 还有 push_front（将元素插入到前面） 和 pop_front(删除最前面的元素）操作，复杂度是 O(1)。

3.双向链表

在任何位置插入删除都是常数时间，不支持随机存取。除了具有所有顺序容器都有的成员函数以外，还支持 8 个成员函数：

函数名	作用	函数名	作用
push_front	在前面插入	unique	删除所有**和前一个元素相同**的元素（要做到元素不重复，则在 unique 之前还需要 sort）
pop_front	删除最前面的元素	merge	合并两个链表，并清空被合并的那个
sort	排序（**但不支持 STL 的算法 sort**）	reverse	颠倒链表
remove	删除和指定值相等的所有元素	splice	在指定位置前面插入另一链表中的一个或多个元素，并在另一链表中删除被插入的元素

例程：双向链表的例程

```
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;
 class A {
   private:
      int n;
   public:
      A( int n_ ) { n = n_; }
      friend bool operator<( const A & a1, const A & a2);
      friend bool operator==( const A & a1, const A & a2);
      friend ostream & operator <<(  ostream & o, const A & a);
};
```

图 4.1 程序 1

```
bool operator<( const A & a1, const A & a2) {
    return a1.n < a2.n;
}
bool operator==( const A & a1, const A & a2) {
    return a1.n == a2.n;
}
ostream & operator <<(  ostream & o, const A & a) {
    o << a.n;
    return o;
}
```

图 4.2 程序 1 续 1

```
template <class T>
void PrintList(const list<T> & lst) {
//不推荐的写法，还是用两个迭代器作为参数更好
    int tmp = lst.size();
    if( tmp > 0 ) {
        typename list<T>::const_iterator i;
        i = lst.begin();
        for( i = lst.begin();i != lst.end(); i ++)
            cout<<*i<<",";
    }
}//typename 用来说明  list<T>::const_iterator 是个类型  //在 vs 中不写也可以
int main(){
    list<A> lst1,lst2;
    lst1.push_back(1);lst1.push_back(3); lst1.push_back(2);lst1.push_back(4);
    lst1.push_back(2);
    lst2.push_back(10);lst2.push_front(20);
    lst2.push_back(30);lst2.push_back(30);
    lst2.push_back(30);lst2.push_front(40); lst2.push_back(40);
    cout<<"1)";PrintList(lst1);cout<<endl;
    //1)1,3,2,4,2
    cout<<"2)";PrintList(lst2);cout<<endl;
    //2)40,20,10,30,30,30,40
    lst2.sort();
```

```
cout << "3) "; PrintList( lst2); cout << endl;
//3) 10,20,30,30,30,40,40,
lst2.pop_front();
cout << "4) "; PrintList( lst2); cout << endl;
//4) 20,30,30,30,40,40,
lst1.remove(2); //删除所有和A(2)相等的元素
cout << "5) "; PrintList( lst1); cout << endl;
//5) 1,3,4,
lst2.unique();  //删除所有和前一个元素相等的元素
cout << "6) "; PrintList( lst2); cout << endl;
//6) 20,30,40,
```

图 4.3 程序 1 续 2

```
lst1.merge (lst2);  //合并 lst2到lst1并清空lst2
cout << "7) "; PrintList( lst1); cout << endl;
//7) 1,3,4,20,30,40,
cout << "8) "; PrintList( lst2); cout << endl;
//8)
lst1.reverse();
cout << "9) "; PrintList( lst1); cout << endl;
//9) 40,30,20,4,3,1,
```

图 4.4 程序 1 续 3

```
lst2.push_back (100);lst2.push_back (200);
lst2.push_back (300);lst2.push_back (400);
list<A>::iterator p1,p2,p3;
p1 = find(lst1.begin(),lst1.end(),3);
p2 = find(lst2.begin(),lst2.end(),200);
p3 = find(lst2.begin(),lst2.end(),400);
lst1.splice(p1,lst2,p2, p3);
//将[p2,p3)插入p1之前，并从lst2中删除[p2,p3)
cout << "10) "; PrintList( lst1); cout << endl;
//10) 40,30,20,4,200,300,3,1,
cout << "11) "; PrintList( lst2); cout << endl;
//11) 100,400,
return 0;
```

图 4.5 程序 1 续 4

第五节 函数对象

1.函数对象定义

函数对象是个对象，但是用起来看上去像函数调用，实际上也执行了函数调用。若一个类重载了运算符“()”，则该类的对象就成为了函数对象。

例如：

class CMyAverage{

```
    public:
    double operator()(int a1, int a2, int a3){
        return (double)(a1+a2+a3)/3;
    }
};
CMyAverage agerage;
cout<<agerage(3,4,5);//输出为 4
```

2.STL 里有模板举例

（1）template<class InIt, class T, class Pred>
T accumulate(InIt first, InIt last, T val, Pred pr);
pr 就是一个函数对象，对[first,last)中的每个迭代器 I，执行 val = pr(val,*I),返回最终的 val。pr 也可以是个函数。

Dev C++中的 Accumulate 源代码 1：

```
template<typename _InputIterator, typename _Tp>
_Tp accumulate(_InputIterator __first, _InputIterator __last, _Tp __init){
    for(;__first!=__last;++__first)
        __init = __init + *__first;
    return __init;
}
//typename 等价于 class
```

Dev C++中的 Accumulate 源代码 2：

```
template<typename _InputIterator, typename _Tp, typename _BinaryOperation>
_Tp accumulate(_InputIterator __first, _InputIterator __last, _Tp __init,_BinaryOperation __binary_op){
    for(;__first!=__last;++__first)
        __init = __binary_op(__init + *__first);
    return __init;
}
```

调用 accumulate 时，和__binary_op 对应的实参可以是个**函数或函数对象**。

（2）greater 函数对象类模板

```
template<class T>
struct greater : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const {
        return x>y;
    }
};
```

可以看出，在这里只有 x>y 的时候才会返回 true。

list 有两个 sort 函数，第一个是前面所说的不带参数的 sort 函数，它将 list 中的元素按“规定的比较方法”升序排列。

list 还有另一个 sort 函数：

```
template <class T2>
void sort(T2 op);
```

可以用 op 来比较大小，即 op(x,y)为 true 则认为 x 应该排在前面。

例程：

```
#include <list>
#include <iostream>
#include <iterator>
using namespace std;
class MyLess {
    public:
    bool operator()(const int & c1, const int & c2)
    {
        return (c1%10)<(c2%10);
    }
};
int main()
{
    const int SIZE = 5;
    int a[SIZE] = {5,21,14,2,3};
    list<int> lst(a,a+SIZE);
    lst.sort(MyLess()); //调用的第二种 sort 排序，由 MyLess()来确定，准确的说是()确定
    Print(lst.begin(),lst.end());
    cout<<endl;
    lst.sort(greater<int>());//greater<int>()是个对象
    Print(lst.begin(),lst.end());
    cout<<endl;
    return 0;
}
```

输出结果：21,2,3,14,5,（按照个位数排序）

21,14,5,3,2,（倒序排列）

3.在 STL 中使用自定义的“大”“小”关系

关联容器和 STL 中许多算法，都是可以用函数或函数对象自定义比较器的。在自定义了比较器 op 的情况下，以下三种说法是等价的：

①x 小于 y；

②op(x,y)返回值为 true；

③y 大于 x。

例题：写出 MyMax 模板

```
#include <iostream>
#include <iterator>
using namespace std;
class MyLess {
    public:
    bool operator()(const int & c1, const int & c2)
    {
        if((c1%10)<(c2%10))
            return true;
        elsereturn false;
    }
```

```
    bool MyCompare(int a1, int a2)
    {
        if((a1%10)<(a2%10))
            return false;
        else
            return true;
    }
};
template <class T, class Pred>
T MyMax(T first, T last, Pred myless)
{
    T tmpMax= first;//tmpMax 是一个迭代器
    for(;first!=last;++first)
        if(myless(*tmpMax,*first))
            tmpMax = first;
    return tmpMax;
};


int main()
{
    int a[] = {35,7,13,19,12};
    cout<<*MyMax(a,a+5,MyLess())<<endl;//个位数最大
    cout<<*MyMax(a,a+5,MyCompare)<<endl;//个位数最小
    return 0;
}
```