程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2025 年 4 月 18 日

大纲

泛型程序设计

C -> C++:

泛型程序设计

C –> C++:
C++ 语言核心优势之一就是便于软件的重用

泛型程序设计

C –> C++:
C++ 语言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:

泛型程序设计

C –> C++:
C++ 语言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：

- 面向对象的思想：继承和多态，标准类库
- 泛型程序设计 (generic programming) 的思想：模板机制，以及标准模板库 STL

泛型程序设计

简单地说就是使用模板的程序设计法，编写不依赖于具体数据类型的程序。

泛型程序设计

简单地说就是使用模板的程序设计法，编写不依赖于具体数据类型的程序。

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类型对象，算法针对什么类型的对象，则都不必重新实现数据结构，重新编写算法。

泛型程序设计

简单地说就是使用模板的程序设计法，编写不依赖于具体数据类型的程序。

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类型对象，算法针对什么类型的对象，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。

泛型程序设计

简单地说就是使用模板的程序设计法，编写不依赖于具体数据类型的程序。

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类型对象，算法针对什么类型的对象，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。

STL 中的基本的概念

- 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
  - `sort()`来对一个`vector`中的数据进行排序
  - `find()`来搜索一个`list`中的对象

STL 中的基本的概念

- 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
  - sort()来对一个vector中的数据进行排序
  - find()来搜索一个list中的对象

  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数据结构上使用。

```
1  int array[100];
2  //该数组就是容器，
3  //而 int * 类型的指针变量就可以作为迭代器，
4  //sort 算法可以作用于该容器上，对其进行排序：
5  sort(array,array+70); //将前 70 个元素排序
```

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，都是类模版，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`

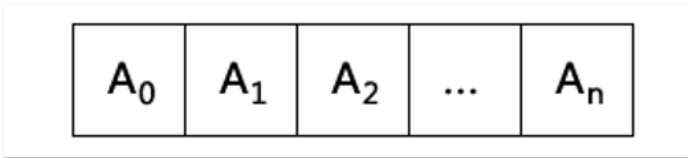
顺序容器

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.

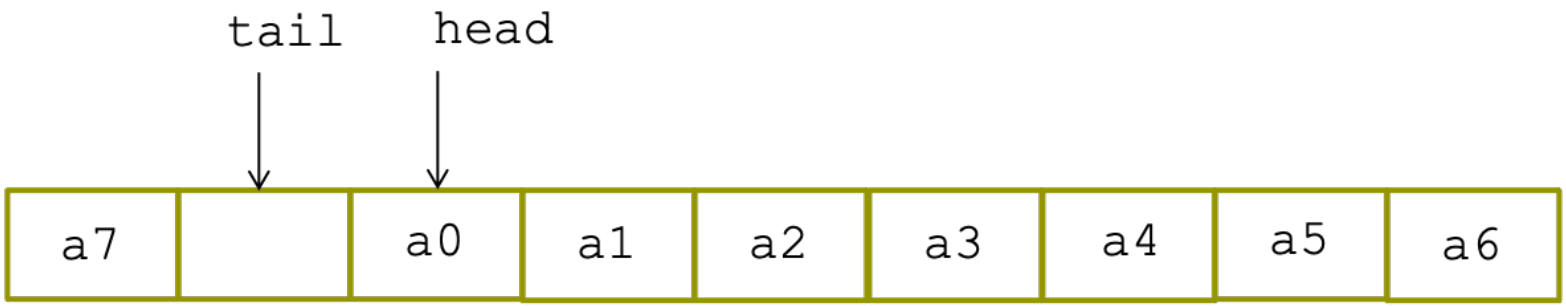
- vector 头文件 <vector>
  动态数组。元素在内存连续存放。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能 (大部分情况下是常数时间)。
- deque 头文件 <deque>
  双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成 (但次于vector)。在两端增删元素具有较佳的性能 (大部分情况下是常数时间)。
- list 头文件 <list>
  双向链表。元素在内存不连续存放。在任何位置增删元素都能在常数时间完成。不支持随机存取。

顺序容器

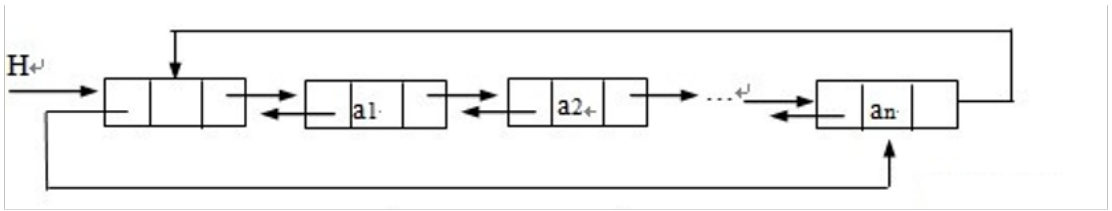
- vector



- deque



- list



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
在查找时具有非常好的性能, 通常以平衡二叉树方式实现，插入和检索的时间都是 $O(log(N))$

- set/multiset 头文件 <set>
  set 即集合。set中不允许相同元素，multiset中允许存在相同的元素。
- map/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，并可快速地根据first来检索元素。map与multimap的不同在于是否允许相同first值的元素。

容器适配器

适配器的含义

简单的理解容器适配器，其就是将不适用的序列式容器（包括 vector、deque 和 list）变得适用。即通过封装某个序列式容器，并重新组合该容器中包含的成员函数，使其满足某些特定场景的需要。

容器适配器本质上还是容器，只不过此容器模板类的实现，利用了大量其它基础容器模板类中已经写好的成员函数。当然，如果必要的话，容器适配器中也可以自创新的成员函数。

容器适配器

- `stack`：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（栈顶的项）。后进先出。
- `queue` 头文件 `<queue>`
  队列。插入只可以在尾部进行。删除只能在头部进行。检索和修改只能在头或尾进行。先进先出。
- `priority_queue` 头文件 `<queue>`
  优先级队列。最高优先级元素总是第一个出列

顺序容器和关联容器中都有的成员函数

- `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` 返回指向容器中第一个元素前面的位置的迭代器
- `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

顺序容器的常用成员函数

- `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`：在容器末尾增加新元素
- `pop_back`：删除容器末尾的元素
- `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

单选题

对于有 n 个元素的以下容器，大多数情况下，哪种操作速度最快?

A. 在deque头部添加一个元素
B. 在set中查找元素
C. 在vector头部添加一个元素
D. 在map中删除元素

单选题

对于有 n 个元素的以下容器，大多数情况下，哪种操作速度最快?

A. 在deque头部添加一个元素
B. 在set中查找元素
C. 在vector头部添加一个元素
D. 在map中删除元素

答案：A

迭代器

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

迭代器

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

```
//定义一个容器类的迭代器的方法可以是:
 容器类名::iterator    变量名;
 容器类名::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

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

```
//定义一个容器类的迭代器的方法可以是:
 容器类名::iterator    变量名;
 容器类名::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。

迭代器示例

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4  int main()  {
5    vector<int> v; //一个存放 int 元素的数组，一开始里面没有元素
6    v.push_back(1);  v.push_back(2);   v.push_back(3);  v.push_back(4);
7    vector<int>::const_iterator i;   //常量迭代器
8    for(i = v.begin(); i != v.end(); ++i)
9      cout << * i << ",";
10   cout << endl;
11   vector<int>::reverse_iterator r;  //反向迭代器
12   for (r = v.rbegin(); r != v.rend(); ++r)
13     cout << * r << ",";
14   cout << endl;
15   vector<int>::iterator j;   //非常量迭代器
16   for (j = v.begin(); j != v.end(); ++j)
17     * j =  100;
18   for (i = v.begin(); i != v.end(); ++i)
19     cout << * i << ",";
20   return 0;
21 }
```

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 指向容器中上一个元素
* p//取 p 指向的元素
p = p1//赋值
p == p1 , p!= p1//判断是否相等、不等
```

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：

```
1  //双向迭代器的所有操作
2  p += i //将 p 向后移动 i 个元素
3  p -= i //将 p 向向前移动 i 个元素
4  p + i  //值为：指向 p 后面的第 i 个元素的迭代器
5  p - i  //值为：指向 p 前面的第 i 个元素的迭代器
6  p[i]  //  值为：p 后面的第 i 个元素的引用
7  p < p1, p <= p1, p > p1, p>= p1
8  p - p1  //p1 和 p 之间的元素个数
```

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	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

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	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

有的算法，例如sort，binary_search需要通过随机访问迭代器来访问容器中的元素，那么list以及关联容器就不支持该算法！

vector的迭代器

vector的迭代器是随机迭代器，遍历 vector 可以有以下几种做法(deque亦然)：

```
1  vector<int> v(100);
2  int i;
3  for (i = 0;i < v.size(); i ++)
4    cout << v[i]; //根据下标随机访问
5  vector<int>::const_iterator ii;
6  for (ii = v.begin(); ii != v.end(); ++ii)
7    cout << * ii;
8  for (ii = v.begin(); ii < v.end(); ++ii )
9    cout << * ii;
10
11 //间隔一个输出：
12 ii = v.begin();
13 while (ii < v.end()) {
14   cout << * ii;
15   ii = ii + 2;
16 }
```

list 的迭代器

list 的迭代器是双向迭代器，正确的遍历list的方法：

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

错误的做法：

```
1  for (ii = v.begin(); ii < v.end(); ++ii)
2    cout << * ii;
3  //双向迭代器不支持 <,list 没有 [] 成员函数
4  for (int i = 0; i < v.size(); i++)
5    cout << v[i];
```

单选题

下列类模板不支持迭代器的是

A. `list`
B. `stack`
C. `deque`
D. `set`

单选题

下列类模板不支持迭代器的是

A. `list`
B. `stack`
C. `deque`
D. `set`

答案：B

单选题

以下说法哪个是正确的?

A. 双向迭代器可以和整数相加
B. 两个双向迭代器可以相减
C. 两个双向迭代器可以比较大小
D. `list`上面的迭代器不能比较大小

单选题

以下说法哪个是正确的?

A. 双向迭代器可以和整数相加
B. 两个双向迭代器可以相减
C. 两个双向迭代器可以比较大小
D. `list`上面的迭代器不能比较大小

答案：D

算法简介

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()

```
1  template<class InIt, class T>
2  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

算法示例:find()

```
1  int main()  {   //find 算法示例
2    int array[10] = {10, 20, 30, 40};
3    vector<int> v;
4    v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4);
5    vector<int>::iterator p;
6    p = find(v.begin(), v.end(), 3);
7    if (p != v.end())
8      cout << * p << endl; //输出 3
9    p = find(v.begin(), v.end(), 9);
10   if (p == v.end())
11     cout << "not found " << endl;
12   p = find(v.begin()+1, v.end()-2, 1); //整个容器: [1,2,3,4], 查找区间: [2,3)
13   if (p == v.end() - 2)
14     cout << "not found" << endl;
15   int * pp = find(array, array+4, 20);//数组名是迭代器
16   cout << * pp << endl;
17   return 0;
18 }
```

输出:

```
3
not found
not found
20
```

STL 中“大”“小”的概念

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：`binary_search`

有些算法会对区间进行从小到大排序，称为“排序算法”，例：`sort`

还有一些其他算法会用到“大”，“小”的概念
使用 STL 时，在缺省的情况下，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

STL 中“相等”的概念

有时，“x和y相等”等价于“x==y为真”
例：在未排序的区间上进行的算法，如顺序查找find

有时“x和y相等”等价于“x小于y和y小于x同时为假”
例：有序区间算法，如binary_search
关联容器自身的成员函数find

STL 中“相等”概念演示

```
1  class A  {
2    int v;
3  public:
4    A(int n):v(n) { }
5    bool operator<(const A & a2) const {
6    //必须为常量成员函数
7      cout << v << "<" << a2.v << "?" << endl;
8      return false;//直接返回了 false
9    }
10   bool operator==(const A & a2) const {
11     cout << v << "==" << a2.v << "?" << endl;
12     return v == a2.v;
13   }
14 };
15 int main() {
16   A a [] = { A(1), A(2), A(3), A(4), A(5) };
17   cout << binary_search(a, a+4, A(9));  //折半查找
18   return 0;
19 }
```

输出结果：

```
3<9?
2<9?
1<9?
9<1?
1
```

vector 示例程序

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  template<class T>
5  void PrintVector( T s, T e) {
6    for(; s != e; ++s)
7      cout << * s << " ";
8    cout << endl;
9  }
10 int main()  {
11   int a[5] = { 1,2,3,4,5 };
12   vector<int> v(a,a+5);  //将数组 a 的内容放入 v
13   cout << "1) " << v.end()  - v.begin() << endl;
14   //两个随机迭代器可以相减，输出 1) 5
15   cout << "2) "; PrintVector(v.begin(),v.end());
16   //2) 1 2 3 4 5
17   v.insert(v.begin() + 2, 13); //在 begin()+2 位置插入 13
18   cout << "3) "; PrintVector(v.begin(),v.end());
19   //3) 1 2 13 3 4 5
20   v.erase(v.begin() + 2); //删除位于 begin() + 2 的元素
21   cout << "4) "; PrintVector(v.begin(),v.end());
22   //4) 1 2 3 4 5
23   vector<int> v2(4, 100);  //v2 有 4 个元素，都是 100
24   v2.insert(v2.begin(),v.begin()+ 1,v.begin()+3);
25   //将 v 的一段插入 v2 开头
26   cout << "5) v2: "; PrintVector(v2.begin(),v2.end());
27   //5) v2: 2 3 100 100 100 100
28   v.erase(v.begin() + 1, v.begin() + 3);
29   //删除 v 上的一个区间，即 2,3
30   cout << "6) "; PrintVector(v.begin(),v.end());
31   //6) 1 4 5
32   return 0;
33 }
```

用vector实现二维数组

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main() {
5    vector<vector<int> > v(3);
6    //v 有 3 个元素，每个元素都是 vector<int> 容器
7    for(int i = 0;i < v.size(); ++i)
8      for(int j = 0; j < 4; ++j)
9        v[i].push_back(j);
10   for(int i = 0;i < v.size(); ++i) {
11     for(int j = 0; j < v[i].size(); ++j)
12       cout << v[i][j] << " ";
13     cout << endl;
14   }
15   return 0;
16 }
```

用vector实现二维数组

```cpp
#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;
}
```

程序输出结果:

```
0 1 2 3
0 1 2 3
0 1 2 3
```

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 $O(1)$

list 容器

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：

```
push_front: //在前面插入
pop_front:   //删除前面的元素
sort: //排序  ( list 不支持 STL 的算法 sort)
remove: // 删除和指定值相等的所有元素
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
        //  unique 之前还需要 sort)
merge:   //归并二个已排序链表为一个，链表应以升序排序
        //操作后容器 第二个变为空。若两个列表为同一对象则函数不做任何事
reverse: //颠倒链表
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
```

list示例程序

```
1  class A {
2  private:
3    int n;
4  public:
5    A(int n_ ) { n = n_; }
6    friend bool operator<(const A & a1, const A & a2);
7    friend bool operator==(const A & a1, const A & a2);
8    friend ostream & operator <<(ostream & o, const A & a);
9  };
10 bool operator<(const A & a1, const A & a2) {
11   return a1.n < a2.n;
12 }
13 bool operator==(const A & a1, const A & a2) {
14   return a1.n == a2.n;
15 }
16 ostream & operator <<(ostream & o, const A & a) {
17   o << a.n;
18   return o;
19 }
20 template <class T>
21 void PrintList(const list<T> & lst) {
22 //不推荐的写法，还是用两个迭代器作为参数更好
23   int tmp = lst.size();
24   if (tmp > 0) {
25     typename list<T>::const_iterator i;
26     i = lst.begin();
27     for (i = lst.begin(); i != lst.end(); ++i)
28       cout << * i << ",";
29   }
30 }// typename 用来说明 list<T>::const_iterator 是个类型
```

list示例程序

```
31  int main()  {
32    list<A>  lst1,lst2;
33    lst1.push_back(1);lst1.push_back(3); lst1.push_back(2);lst1.push_back(4);
34    lst1.push_back(2);
35    lst2.push_back(10);lst2.push_front(20);
36    lst2.push_back(30);lst2.push_back(30);
37    lst2.push_back(30);lst2.push_front(40);
38    lst2.push_back(40);
39    cout << "1) "; PrintList( lst1); cout << endl;
40    // 1) 1,3,2,4,2,
41    cout << "2) "; PrintList( lst2); cout << endl;
42    // 2) 40,20,10,30,30,30,40,
43    lst2.sort();
44    cout << "3) "; PrintList( lst2); cout << endl;
45    //3) 10,20,30,30,30,40,40,
46    lst2.pop_front();
47    cout << "4) "; PrintList( lst2); cout << endl;
48    //4) 20,30,30,30,40,40,
49    lst1.remove(2); //删除所有和 A(2) 相等的元素
50    cout << "5) "; PrintList( lst1); cout << endl;
51    //5) 1,3,4,
52    lst2.unique();  //删除所有和前一个元素相等的元素
53    cout << "6) "; PrintList( lst2); cout << endl;
54    //6) 20,30,40,
55    lst1.sort();
56    lst1.merge (lst2);  //归并 lst2 到 lst1, 并清空 lst2
57    cout << "7) "; PrintList( lst1); cout << endl;
58    //7) 1,3,4,20,30,40,
59    cout << "8) "; PrintList( lst2); cout << endl;
60    //8)
```

list示例程序

```
61      lst1.reverse();
62      cout << "9) "; PrintList( lst1); cout << endl;
63      //9) 40,30,20,4,3,1,
64      lst2.push_back (100);lst2.push_back (200);
65      lst2.push_back (300);lst2.push_back (400);
66      list<A>::iterator p1,p2,p3;
67      p1 = find(lst1.begin(),lst1.end(),3);
68      p2 = find(lst2.begin(),lst2.end(),200);
69      p3 = find(lst2.begin(),lst2.end(),400);
70      lst1.splice(p1,lst2,p2, p3);
71      //将 [p2,p3) 插入 p1 之前，并从 lst2 中删除 [p2,p3)
72      cout << "10) "; PrintList( lst1); cout << endl;
73      //10) 40,30,20,4,200,300,3,1,
74      cout << "11) "; PrintList( lst2); cout << endl;
75      //11) 100,400,
76      return 0;
77  }
```

单选题

关于顺序容器迭代器的操作，不正确的是

A. `vector<int>::iterator iter1, iter2; iter1 < iter2;`

B. `list<int>::iterator iter1, iter2; iter1 < iter2;`

C. `vector<int>::iterator iter1, iter2; iter1 -= 3;`

D. `deque<int>::iterator iter1; iter1 += 5;`

单选题

关于顺序容器迭代器的操作，不正确的是

A. `vector<int>::iterator iter1, iter2; iter1 < iter2;`

B. `list<int>::iterator iter1, iter2; iter1 < iter2;`

C. `vector<int>::iterator iter1, iter2; iter1 -= 3;`

D. `deque<int>::iterator iter1; iter1 += 5;`

答案：B

函数对象

是个对象，但是用起来看上去象函数调用，实际上也执行了函数调用。

```
1  class CMyAverage {
2  public:
3    double operator()(int a1, int a2, int a3) {
4      //重载 () 运算符
5      return (double)(a1 + a2 + a3) / 3;
6    }
7  };
8  CMyAverage average;  //函数对象
9  cout << average(3, 2, 3); // average.operator()(3,2,3) 用起来看上去象函数调
   ↪  用
10                           //输出 2.66667
```

函数对象的应用

STL 里有以下模板：

```
1  template<class InIt, class T, class Pred>
2  T accumulate(InIt first, InIt last, T val, Pred pr);
```

pr 就是个函数对象。
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 源代码参考

形式 1

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

形式 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;
}
```

accumulate函数示例

```
1  int SumSquares( int total, int value){
2    return total + value * value;
3  }
4  template <class T>
5  void PrintInterval(T first, T last){ //输出区间 [first,last) 中的元素
6    for( ; first != last; ++ first)
7      cout << * first << " ";
8    cout << endl;
9  }
10 template<class T>
11 class SumPowers {
12 private:
13   int power;
14 public:
15   SumPowers(int p):power(p) { }
16   const T  operator() (const T & total, const T & value) const {
17     //计算 value 的 power 次方, 加到 total 上
18     T v = value;
19     for (int i = 0; i < power - 1; ++ i)
20       v = v * value;
21     return total + v;
22   }
23 };
```

accumulate函数示例

```
1  int main() {
2    const int SIZE = 10;
3    int a1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
4    vector<int> v(a1, a1 + SIZE);
5    cout << "1) ";
6    PrintInterval(v.begin(), v.end());
7    int result = accumulate(v.begin(), v.end(), 0, SumSquares);
8    cout << "2) 平方和: " << result << endl;
9    result = accumulate(v. begin(),v.end(), 0, SumPowers<int>(3));
10   cout << "3) 立方和: " << result << endl;
11   result = accumulate(v.begin(), v.end(), 0, SumPowers<int>(4));
12   cout << "4) 4 次方和: " << result;
13   return 0;
14 }
```

accumulate函数示例

```
1  int main() {
2    const int SIZE = 10;
3    int a1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
4    vector<int> v(a1, a1 + SIZE);
5    cout << "1) ";
6    PrintInterval(v.begin(), v.end());
7    int result = accumulate(v.begin(), v.end(), 0, SumSquares);
8    cout << "2) 平方和: " << result << endl;
9    result = accumulate(v. begin(),v.end(), 0, SumPowers<int>(3));
10   cout << "3) 立方和: " << result << endl;
11   result = accumulate(v.begin(), v.end(), 0, SumPowers<int>(4));
12   cout << "4) 4 次方和: " << result;
13   return 0;
14 }
```

输出

```
1) 1 2 3 4 5 6 7 8 9 10
2) 平方和: 385
3) 立方和: 3025
4) 4 次方和: 25333
```

accumulate函数示例

```
1  int result = accumulate(v.begin(), v.end(), 0, SumSquares);
```

实例化出:

```
1  int accumulate(vector<int>::iterator first, vector<int>::iterator last,
2                 int init,int ( * op)( int,int)) {
3    for ( ; first != last; ++first)
4      init = op(init, *first);
5    return init;
6  }
```

STL 中函数对象类模板

STL 的<functional> 里还有以下函数对象类模板：

- equal_to
- greater
- less
- ...

这些模板可以用来生成函数对象。

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;
  }
};
```

binary_function定义：

```
template<class Arg1, class Arg2, class Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};
```

注：这是 C++98 中的定义，C++11 中已更新

greater 的应用

list 有两个sort函数，前面例子中看到的是不带参数的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：

```
1  template <class T2>
2  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。

greater 的示例

```
1  class MyLess {
2  public:
3    bool  operator()(const int & c1, const int & c2 ) const {
4      return (c1 % 10) < (c2 % 10);
5    }
6  };
7  int main() {
8    const int SIZE = 5;
9    int a[SIZE] = {5, 21, 14, 2, 3};
10   list<int> lst(a, a+SIZE);
11   lst.sort(MyLess());
12   ostream_iterator<int> output(cout, ",");
13   copy(lst.begin(), lst.end(), output); cout << endl;
14   lst.sort(greater<int>()); //greater<int>() 是个对象, 本句进行降序排序
15   copy(lst.begin(), lst.end(), output); cout << endl;
16   return 0;
17 }
```

输出:

```
21,2,3,14,5,
21,14,5,3,2,
```

ostream_iterator

copy类似于

```
1  template <class T1, class T2>
2  void copy(T1 s, T1 e, T2 x) {
3    for(; s != e; ++s,++x)
4      *x = *s;
5  }
```

ostream_iterator 如何编写?

ostream_iterator

copy类似于

```
1  template <class T1, class T2>
2  void copy(T1 s, T1 e, T2 x) {
3    for(; s != e; ++s,++x)
4      *x = *s;
5  }
```

ostream_iterator 如何编写?

如果要实现ostream_iterator类应该重载以下运算符中的几个?

=, *, ++, !=

A. 1

B. 2

C. 3

D. 4

ostream_iterator

copy类似于

```
1  template <class T1, class T2>
2  void copy(T1 s, T1 e, T2 x) {
3    for(; s != e; ++s,++x)
4      *x = *s;
5  }
```

ostream_iterator 如何编写?

如果要实现ostream_iterator类应该重载以下运算符中的几个?

=, *, ++, !=

A 1

B 2

C 3

D 4

答案: C

ostream_iterator

copy 类似于

```
1  template <class T1,class T2>
2  void copy(T1 s,T1 e, T2 x) {
3    for(; s != e; ++s,++x)
4      *x = *s;
5  }
```

ostream_iterator类，将输出的工作放在哪个运算符中做最合适?

A ++

B =

C *

D !=

ostream_iterator

copy 类似于

```
1  template <class T1,class T2>
2  void copy(T1 s,T1 e, T2 x) {
3    for(; s != e; ++s,++x)
4      *x = *s;
5  }
```

ostream_iterator类，将输出的工作放在哪个运算符中做最合适?

A ++

B =

C *

D !=

答案：B

例题 MyMax

```cpp
#include <iostream>
#include <iterator>
using namespace std;
class MyLess {
public:
  bool operator() (int a1, int a2) const {
    if ((a1 % 10) < (a2 % 10))
      return true;
    else
      return false;
    }
};
bool MyCompare(int a1, int a2) {
  if ((a1 % 10) < (a2 % 10))
    return false;
  else
    return true;
}
int main() {
  int a[] = {35, 7, 13, 19, 12};
  cout << MyMax(a, 5, MyLess()) << endl;
  cout << MyMax(a, 5, MyCompare) << endl;
  return 0;
}
```

输出:

```
19
12
```

要求写出 MyMax

例题 MyMax

```
1  template <class T, class Pred>
2  T MyMax(T *p, int n, Pred myless) {
3    T tmpmax = p[0];
4    for(int i = 1; i < n; i++)
5      if(myless(tmpmax, p[i]))
6        tmpmax = p[i];
7    return tmpmax;
8  };
```

引入函数对象后，STL 中的“大”，“小”关系

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

- x小于y
- op(x,y)返回值为true
- y大于x

比较规则的注意事项

```
struct 结构名 {
  bool operator()( const T & a1,const T & a2) const {
    //若 a1 应该在 a2 前面，则返回 true。
    //否则返回 false。
  }
};
```

- 比较规则返回 true，意味着 a1 必须在 a2 前面
- 返回 false，意味着 a1 并非必须在 a2 前面
- 排序规则的写法，不能造成比较 a1,a2 返回 true 比较 a2,a1 也返回 true
- 否则会出问题，比如sort会 runtime error，
- 比较 a1,a2 返回 false 比较 a2,a1 也返回 false，则没有问题

用sort对结构数组进行排序

```
1  struct Student {
2    char name[20];
3    int id;
4    double gpa;
5  };
6  Student students [] = {{"Jack",112,3.4},{"Mary",102,3.8},{"Mary",117,3.9},
7    {"Ala",333,3.5},{"Zero",101,4.0}};
8  struct StudentRule1 { //按姓名从小到大排
9    bool operator() (const Student & s1, const Student & s2) const {
10     if( stricmp(s1.name,s2.name) < 0)
11       return true;
12     return false;
13   }
14 };
15 struct StudentRule2 { //按 id 从小到大排
16   bool operator() (const Student & s1, const Student & s2) const {
17     return s1.id < s2.id;
18   }
19 };
20 struct StudentRule3 {//按 gpa 从高到低排
21   bool operator() (const Student & s1, const Student & s2) const {
22     return s1.gpa > s2.gpa;
23   }
24 };
25 void PrintStudents(Student s[], int size){
26   for(int i = 0;i < size;++i)
27     cout << "(" << s[i].name << ","  << s[i].id <<"," << s[i].gpa << ") " ;
28   cout << endl;
29 }
```

用sort对结构数组进行排序

```
30  int main(){
31    int n = sizeof(students) / sizeof(Student);
32    sort(students,students+n, StudentRule1()); //按姓名从小到大排
33    PrintStudents(students,n);
34    sort(students,students+n, StudentRule2()); //按 id 从小到大排
35    PrintStudents(students,n);
36    sort(students,students+n, StudentRule3()); //按 gpa 从高到低排
37    PrintStudents(students,n);
38    return 0;
39  }
```

用sort对结构数组进行排序

```
30  int main(){
31    int n = sizeof(students) / sizeof(Student);
32    sort(students,students+n, StudentRule1()); //按姓名从小到大排
33    PrintStudents(students,n);
34    sort(students,students+n, StudentRule2()); //按 id 从小到大排
35    PrintStudents(students,n);
36    sort(students,students+n, StudentRule3()); //按 gpa 从高到低排
37    PrintStudents(students,n);
38    return 0;
39  }
```

输出结果:

```
(Ala,333,3.5) (Jack,112,3.4) (Mary,102,3.8) (Mary,117,3.9) (Zero,101,4)
(Zero,101,4) (Mary,102,3.8) (Jack,112,3.4) (Mary,117,3.9) (Ala,333,3.5)
(Zero,101,4) (Mary,117,3.9) (Mary,102,3.8) (Ala,333,3.5) (Jack,112,3.4)
```