图书在版编目(CIP)数据

C++程序设计与应用/张耀仁 著
武汉:华中科技大学出版社,2002年11月
ISBN 7-5609-2866-8

Ⅰ.C…
Ⅱ.张…
Ⅲ.C++语言-程序语言 教学参考资料
Ⅳ.TP312

责任编辑:周 筠 (http://yeka.xilubbs.com)
技术编辑:任颂华 (TR@SOE)
责任校对:陈元玉
出版发行:华中科技大学出版社 (武昌喻家山 邮编:430074 电话:(027)87541791)
录 排:华中科技大学惠友科技文印中心
印 刷:湖北新华印务有限公司
开 本:787×1092 1/16 印 张:46.75 插 页:2 字 数:700 000
版 次:2002年11月第1版 印 次:2002年11月第1次印刷
印 数:1—5 000 定 价:59.80元(含1CD)
ISBN 7-5609-2866-8/TP·488

第23章

泛型程序设计简介

C++ 标准模板链接库 (STL) 内的数据结构以“模板类”的形式定义，称为“容器类”(container class)，而算法则是以“模板函数”的形式定义，称为“泛型算法”(generic algorithms)。对于相同的容器类，不管容器内对象的数据是什么类型，都可以运用相同的泛型算法在其上进行同样的处理，这就是泛型程序设计 (generic programming) 的主要精神。

23.1 C++ 标准模板链接库 (STL)

所有学习 C++ 的人都有撰写函数 (function) 和类（class）的经验，为了让好不容易写成的函数和类能适用于不同的数据类型，我们可以使用第 10 章介绍的模板函数 (function template) 和第 22 章介绍的模板类 (class template)，以将它们转成通用的算法 (algorithms) 和数据结构 (data structures)。

但是，这些模板函数和模板类是否非自己亲手制作不可呢？有没有现成的程序代码可以利用呢？除了可以另外购买特殊用途的链接库以外，事实上每一个标准的 C++编译器都有一套名为标准模板链接库（the Standard Template Library，简称 STL）的内置链接库可以利用。

使用现成的模板程序代码写作应用程序称为泛型程序设计 (generic programming)。以 C++ 而言，泛型程序设计包括模板函数，模板类以及迭代子 (iterator) 的应用。

第一个实验性的泛型链接库（generic library）并不是以 C++ 写成，而是由 David R.Musser 和 Alexander A. Stepanov 在 1989 年使用程序语言 Ada 完成的“Ada Generic Library”。而 Alexander A. Stepanov 和 Meng Lee 随后在惠普实验室（Hewlett-Packard Laboratories）为 C++ 设计的 STL（第一版在 1994 年完成）则是最早被广为接受的泛型程序设计典范。目前，在 David Musser 的增修后，STL 已经是 ANSI/ISO C++ 标准链接库（the C++ Standard Library）的一部分。

23.2 STL 的主要内容

当 STL 成为 C++ 标准的一部分时，C++ 使用者很快地就了解到它是一个高效率的链接库，特别是将它视为由许多容器 (container) 组成的类链接库。因此，STL 的模板类又称为容器类 (container class)。

提示

在泛型程序设计的术语里，容器 (container) 是数据结构的广义称呼。Container 用来承装对象，就好比一般的容器是用来承装物品一样。第 9 章的字符串 (string) 是一

种容器，第 15 章提到的列表(list)、堆栈(stack)、二叉树 (binary tree)以及第 22 章的模板类 Vector 和 Matrix 都是容器。

STL 除了将所有的容器类 (container classes) 都写成模板类（class templates）以提供最通用的数据结构之外，还提供了能有效地在这些数据结构上进行处理的算法，并且都写成模板函数（function templates）的形式。

C++ 的标准模板链接库（STL）包括下列主要的三大类：

（一）七种通用的数据结构 (data structures)

数据结构名称（容器类）	说　明
vector	可动态地调整向量元素的数量 使用时要在程序前加上 #include <vector>
list	双向链结的列表（doubly-linked list）可使用指针向前后搜索 使用时要在程序前加上 #include <list>
deque	可允许从数组的头端或尾端加入新元素，而不需要额外复制中间的元素 使用时要在程序前加上 #include <deque>
stack	亦即具有LIFO（后进先出）特性的“堆栈”，只能对最上面的元素进行存取（push和pop）的动作 使用时，要在程序前加上指令 #include <stack>
queue	就是具有FIFO（先进先出）特性的队列 使用时，要在程序前加上指令 #include<queue>
set	由一群 key（键值）构成，且经过排序的数据集合。例如，所有“可疑车号”的set，可以供警方在路边抽检时迅速对照使用时，要在程序前加上指令 #include <set>
map	由成对的key和某种数据构成的集合。例如，以“车号”为key，以“车号和车身颜色”为配合的数据所形成的 map，可供警方在输入车号后，迅速检查车牌和汽车是否一致，以筛选犯罪车辆使用时，要在程序前加上指令 #include <map>

(二) 算法 (algorithms)

又称为泛型算法 (generic algorithms)，是以模板函数的形式包装起来的各种算法，可以配合上述各种数据结构使用。使用这些算法时，必须在程序前加入以下的预处理指令：

```
#include <algorithm>
```

按照功能的区别可以将算法分为下列三大类：

1. 只检查不更动的算法

   例如 find()，search()，count()，min()，max()等，以分别进行寻找，对比，计数，寻找最大值，寻找最小值等处理。

2. 会更动容器内元素的算法

   例如 copy()，swap()，replace()，fill()，remove()，reverse()，random_shuffle()，merge()，sort()等，以进行复制，交换，替换，加入，删除，颠倒次序，打乱重排，合并，排序等操作。

3. 和数值计算相关的算法

   例如 accumulate()，inner-product()，partial_sum()，adjacent_difference()等，以进行元素总合，内积，部分元素的累加，隔邻元素差值等简单的数值运算。一般而言，这类演算不符合真正数值演算的需要。

(三) 迭代子 (iterators)

“迭代子”是一种对象，用来指向其它对象，它本来就是广义的指针，也可以视为指针对象 (pointer object)。(注：iteration 是“重复执行”的意思，由于指针对象在容器内来来回回地进行对对象地址的取用工作，因此得名。)

在 STL 里面，迭代子担任算法和数据结构之间信号传递的联系工作。由于并不是所有的算法和数据结构都可以自由配对，迭代子同时也担任限制在容器上可以执行的算法种类之工作。此外，所有迭代子的声明都包括在算法里面，因此不需要额外的预处理指令。

迭代子可以分为 5 类（另外一种讲法是：可以由五种“概念，concepts”规范）：

1. 输入型迭代子（input iterator）

   如果 p 是一个输入型迭代子，则其行为就好像可以从键盘或文件输入数据一样，具有以下的特性：

   - 可以比较彼此的相同性，例如“p1 != p2”。

- 可以取得指针所指处的对象，例如 “float x = *p;”。
- 可以进行累加运算，亦即“++p”和“p++”。
- 仅可以读取目标对象，亦即 *p 被限制为只能是右值（rvalue），不可以进行诸如“*p = x;”的运算。

2. 输出型迭代子（output iterator）

其特质就好像可以将数据送往输出数据流（output stream）一样，是“只写”型的迭代子。如果 p 是一个输出型迭代子，则它具有以下的特性：

- 迭代子间可以复制和赋值。亦即可以进行“p1(p2);”和“p1=p2;”的运算。
- 可以进行“++p”和“p++”的累加运算。
- 仅可以输出目标对象的值，亦即*p 被限制为只能是左值（lvalue），不可以进行诸如“x = *p;”的运算。

3. 前进型迭代子（forward iterator）

它同时具有输入型迭代子和输出型迭代子的特征，可以对目标对象进行读写，亦即前进型迭代子可以是左值也可以是右值。“x = *p;”和“*p = y;”都是合法的语句。由于只能进行累加“++p”和“p++”的操作，不能进行累减“--p”和“p--”的操作，因此得名。

4. 双向型迭代子（bi-directional iterator）

除了前进型迭代子所具有的特性外，同时可以进行反向累减的操作，亦即“--p”和“p--”都是合法的语句。

5. 随机存取型迭代子（random access iterator）

它除了具有上述的所有特质外，还可以进行任意大小距离的跳跃动作。也就是，不只能够累加或累减，还可以进行诸如：

“p += 2;” 和 “*(p+3) = y;”

等操作。

泛型程序设计 (generic programming) 是基于模板函数和模板类的重要技术，和对象导向程序设计 (object-oriented programming) 两者具有不同的想法，但一样都是增进程序开发效率的有效方式。限于篇幅，本书无法对标准模板链接库，亦即 STL，进行全面性的介绍，

读者可以参阅下列两本专门介绍 STL 的权威著作：

1. Matthew H. Austern: *Generic Programming and the STL*
   Addison Wesley Longman, Inc., 1999. (书目编号 QA76.73.C153 A97)
2. Nicolai M. Josuttis: *The C++ Standard Library: a Tutorial and Reference*
   Addison Wesley Longman, Inc., 1999. (书目编号 QA76.73.C153 J69)

为了对 STL 有具体的认识，我们将在以下的范例中介绍如何使用 STL 以对容器中的对象进行下述几种简单的操作：

1. 寻找符合特定值的第一个出现的元素。使用 `search()`。
2. 排序。使用 `sort()`。
3. 打乱重排，好比桥牌中“洗牌”的动作，将次序打乱。使用 `random_shuffle()`。
4. 设定迭代子的位置。使用 `at()`。
5. 插入新的元素。使用 `insert()`。

23.3 使用 STL 的 vector 容器类

在 C++ STL 中最基本的容器是 **vector** (向量，亦即一维数组)，几乎所有的泛型算法都可以在 vector 上操作，因此不需包含其它的头文件。(注意，我们在 22-2 节自己写了一个叫做 **Vector** 的模板类，具有一般数学上“向量”的特性，而 STL 中的 vector 只是单纯的一维数组，除了索引运算符 `[]` 外，没有重载其它运算符。)

vector 对象的定义

定义 vector 对象时使用的是模板类对象的定义语法。例如：

```
vector<float> V1(5);

vector<char>  V2(20);
```

分别定义了一个名叫 V1，每个元素的数据类型都是 float，长度为 5 的一维数组；以及一个名叫 V2，每个元素的数据类型都是 char，长度为 20 的一维数组。对于数值型的 vector 对象，预设的元素值都是 0。

如果我们要设定初始值，可以使用标准的一维数组，将含有初始值的数组的开头和结尾地址当作参数一次给予。例如:

```
float A[5] = {1.5, 2.9, 3.8, 4.2, 5.7};
char  B[5] = {'b', 'a', 'i', 'r', 'w'};
vector<float> Vf(A, A + 5);  // A 为开头地址
vector<char>  Vc(B, B + 5);  // B 为开头地址
```

vector 对象的输出

要将 vector 对象输出有两种形式，分别为：(1) 使用输出数据流对象 cout，(2) 合并使用函数 copy()和迭代子：

(1) 使用输出数据流对象 cout

例如，要输出 Vf 的各元素值时，可以使用:

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

函数 size()可以得知 vector 对象的长度。

(2) 合并使用函数 copy()和 ostream_iterator 对象

例如，要输出 Vf 的各元素值时，要先定义可以输出 float 型数据的 ostream_iterator 对象，命名为 IntOut (第二个参数用来指定各元素之间的输出对象，本例使用字符串" "表示以空白隔开各元素值):

```
ostream_iterator <float> IntOut(cout, "  ");
```

然后使用函数 copy()将地址 Vf.begin()到地址 Vf.end()之间的所有数据输出到 IntOut。其中 vector 的成员函数 begin()和 end()分别可以用来取得 vector 对象的开头和结尾 (最末一个元素之后) 的地址。

```
copy(Vf.begin(), Vf.end(), IntOut);
```

插入额外的元素

要在 vector 对象中插入额外元素时，要使用函数 insert()。这个动作会让元素的数目增加一个。例如：

```
Vf.insert(Vf.begin() + 3, 1.2);
```

在 Vf[2]和 Vf[3]之间插入一个数值为 1.2 的元素。

设定特定元素的值

要在 vector 对象中设定特定某元素的值时，有两种做法：

第一种做法是使用索引。例如，要将 Vf[3] 的元素值设定为 2.8 可以写成

```
Vf[3] = 2.8;
```

第二种做法是使用函数 at()来设定地址。例如，要将 Vf[2] 的元素值设定为 9.9 可以写成

```
Vf.at(2) = 9.9;
```

使用函数 at()的好处是它会检查要设定元素值的位置有没有超过目前这个 vector 对象的范围；如果使用索引则不做检查，一旦发生错误时很难查出确实的原因。

将 vector 对象各元素的值依大小排序

要将 vector 对象各元素的值依大小排序时，可以使用成员函数 sort()。例如

```
sort(Vf.begin(), Vf.end());
```

将依 Vf 的元素值排序。如果各元素的数据类型是 char，则排序的根据是 ASCII 值。

将 vector 对象各元素打乱重排

如果要将 vector 对象各元素打乱重排，好比桥牌中“洗牌”的动作将次序打乱时，可以使用成员函数 random_shuffle()。例如

```
random_shuffle(Vf.begin(), Vf.end());
```

在程序 StdVector.cpp 中，我们使用 <vector> 去定义包含 int 和 double 两种数据类型元素的向量 Vf 和 Vc，并验证上述各种使用 <algorithm> 指令的正确性。

范例程序　文件 StdVector.cpp

```
// StdVector.cpp
#include <iomanip>
#include <vector>
#include <algorithm>
using std::vector;
using std::copy;
using std::sort;
using std::random_shuffle;
using std::ostream_iterator;
using std::cout;
using std::endl;
using std::setw;
using std::setprecision;

// ------------- 主程序 -----------------------------
int main()
{
  const int Size = 5;
  float A[Size] = {1.5, 2.9, 3.8, 4.2, 5.7};
  char  B[Size] = {'b', 'a', 'i', 'r', 'w'};
  // -- 定义 vector 对象 ---------------------------
  vector<float> Vf(A, A + Size);
  vector<char>  Vc(B, B + Size);
  // -- 定义 ostream_iterator 对象 -----------------
  ostream_iterator<float> IntOut(cout, "  ");
  ostream_iterator<char>  CharOut(cout, "  ");
  cout << std::showpoint
       << std::setprecision(4);
  cout << "\n--------测试 vector<float>---------------"
       << endl;
  cout << "\nVf 为: "  << endl;
  // -- vector 对象的输出 --------------------------
  for (int i = 0; i < Vf.size(); i++)
    cout  << Vf[i] << "  ";
```

```
    // -- 设定特定元素的值 ------------------------------
    Vf.at(2) = 9.9;
    cout << endl;
    cout << "在执行“Vf.at(2) = 9.9”之后, Vf 变成为:"
         << endl;
    // -- vector 对象的输出 ----------------------------
    copy(Vf.begin(), Vf.end(), IntOut);
    cout << endl;
    Vf[3] = 2.8;
    cout << "在执行“Vf[3] = 2.8;”之后, Vf 变成为:"
         << endl;
    copy(Vf.begin(), Vf.end(), IntOut);
    // -- 插入额外的元素 --------------------------------
    Vf.insert(Vf.begin() + 3, 1.2);
    cout << "在执行“Vf.insert(Vf.begin() + 3, 1.2)” "
         <<  "\n 之后, Vf 变成为:"<< endl;
    copy(Vf.begin(), Vf.end(), IntOut);
    // -- 将 vector 对象各元素的值依大小排序 ------------
    sort(Vf.begin(), Vf.end());
    cout << "\n 排序之后, Vf 变成为:"     << endl;
    copy(Vf.begin(), Vf.end(), IntOut);
    // -- 将 vector 对象各元素打乱重排 ------------------
    random_shuffle(Vf.begin(), Vf.end());
    cout << "\n 打乱重排之后, Vf 变成为:" << endl;
    // -- vector 对象的输出 ----------------------------
    copy(Vf.begin(), Vf.end(), IntOut);
    cout << endl;
    cout <<
  "\n--------测试 vector<char>-------------------------"
         << endl;
    cout << "\nVc 为: "  << endl;
    // -- vector 对象的输出 ----------------------------
    for (int i = 0; i < Vc.size(); i++)
      cout << Vc[i] << "  ";
    cout << endl;
    // -- 设定特定元素的值 ------------------------------
    Vc.at(2) = 'h';
```

```
    cout << "在执行“Vc.at(2) = 'h'”之后，Vc 变成为:"
         << endl;
    copy(Vc.begin(), Vc.end(), CharOut);
    cout << endl;
    Vc[3] = 'G';
    cout << "在执行“Vc[3] = 'G';”之后，Vf 变成为:"
         << endl;
    copy(Vc.begin(), Vc.end(), CharOut);
    // -- 插入额外的元素 ---------------------------------
    Vc.insert(Vc.begin() + 3, 'p');
    cout << "在执行“Vc.insert(Vc.begin() + 3, 'p')”"
         <<  "\n 之后，Vc 变成为:"<< endl;
    copy(Vc.begin(), Vc.end(), CharOut);
    // -- 将 vector 对象各元素的值依大小排序 -------------
    sort(Vc.begin(), Vc.end());
    cout << "\n 排序之后，Vc 变成为:"
         << endl;
    copy(Vc.begin(), Vc.end(), CharOut);
    // -- 将 vector 对象各元素打乱重排 -------------------
    random_shuffle(Vc.begin(), Vc.end());
    cout << "\n 打乱重排之后，Vc 变成为:"
         << endl;
    copy(Vc.begin(), Vc.end(), CharOut);
    return 0;
}
```

程序执行结果

```
--------测试 vector<float>-------------
Vf 为:
1.500  2.900  3.800  4.200  5.700
在执行“Vf.at(2) = 9.9”之后，Vf 变成为:
1.500  2.900  9.900  4.200  5.700
在执行“Vf[3] = 2.8;”之后，Vf 变成为:
1.500  2.900  9.900  2.800  5.700
在执行“Vf.insert(Vf.begin() + 3, 1.2)”
```

```
之后，Vf 变成为：
1.500  2.900  9.900  1.200  2.800  5.700
排序之后，Vf 变成为：
1.200  1.500  2.800  2.900  5.700  9.900
打乱重排之后，Vf 变成为：
2.900  1.200  9.900  2.800  1.500  5.700

-------测试 vector<char>-----------------
Vc 为：
b  a  i  r  w
在执行“Vc.at(2) = 'h'”之后，Vc 变成为：
b  a  h  r  w
在执行“Vc[3] = 'G';”之后，Vf 变成为：
b  a  h  G  w
在执行“Vc.insert(Vc.begin() + 3, 'p')”
之后，Vc 变成为：
b  a  h  p  G  w
排序之后，Vc 变成为：
G  a  b  h  p  w
打乱重排之后，Vc 变成为：
b  G  p  h  a  w
```

23.4 使用 STL 处理字符串数组

在 23.3 节中，我们在 vector 对象中所存的数据类型只是 float 或 char，程序的结构及算法大致上使用相同的语法；我们在本节中进一步讨论由字符串（string）组成的一维数组，以进行深入的了解。

使用 vector 容器类

首先我们定义一个由字符串组成的数组 S：

```
string A[] =
```

```
{"John","Memi","Brigitte","Daiana","Lulu","Lisa"};
vector <string>  S(A, A + 6);
```

对于这个名叫 S 的 Vector 对象而言，除了元素的数据类型为 string 以外，能够在其上进行的处理，包括 ostream_iterator 的定义，成员函数 at()，以及泛型算法：copy()，sort()，remove()，和 random_shuffle，都和 23.3 节 StdVector.cpp 的语法完全一样。对于相同的容器类 (例如这里的 vector)，不管容器内对象的数据类型为何，都可以运用相同的泛型算法在其上进行同样的处理，这就是泛型程序设计的主要特色。

在程序 StringVector.cpp 中，我们分别对这个名叫 S 的 Vector 对象进行 23.3 节 StdVector.cpp 中相同的处理，包括插入额外的元素，设定特定元素的值，将 vector 对象各元素的值依大小排序，以及将各元素打乱重排等操作。

程序 StringVector.cpp 和程序 StdVector.cpp 有以下四项主要的语法差异：

(1) 对于泛型算法不使用 using 声明

也就是说，由于大部分的泛型算法在本程序中只使用一次，因此在程序开头处不使用诸如下列的 using 声明：

```
using std::copy;
using std::sort;
using std::random_shuffle;
```

而在每个使用的地方加上“std::”的完整名称，代表使用标准链接库的函数：

```
std::copy(S.begin(), S.end(), StringOut);
std::sort(S.begin(), S.end());
std::random_shuffle(S.begin(), S.end());
```

(2) 使用成员函数 find()以寻找存有某特定值的元素地址，并以函数 insert()在此处加入新元素。例如

```
S.insert(find(S.begin(), S.end(), "Daiana"), "Joanne");
```

可以完成在 Daiana 前插入 Joanne 的工作。

(3) 使用泛型算法 remove()以删除存有某特定值的元素。例如

```
std::remove( S.begin(), S.end(), "Brigitte");
```

可以从 vector 对象 S 中删除 Brigitte. 它的返回值是一个迭代子，因此，我们可以使用下列语句定义迭代子 Last:

```
vector<string>::iterator Last;
```

并把从 remove()传回且删除后的 vector 对象尾端的地址以下列语句存起来:

```
Last = std::remove( S.begin(), S.end(), "Brigitte");
```

以备以下操作使用. 不过 remove()并不会改变 vector 对象的长度，只是将所删除元素之后的各元素往前移，因此还需要使用成员函数 resize()以调整对象的长度 (原来的长度是 S.size()):

```
S.resize(S.size()-1);
```

(4) 使用成员函数 find()以寻找存有某特定值的元素地址，并以取值运算符“*”在此处存入新的值. 例如

```
*find(S.begin(), S.end(), "Daiana")= "William";
```

可以完成将 Daiana 取代为 William 的工作.

这些语法同样适用于程序 StdVector.cpp. 完整的范例程序文件 StringVector.cpp 如下.

范例程序 文件 StringVector.cpp

```
// StringVector.cpp
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
using std::vector;
using std::string;
using std::ostream_iterator;
using std::cout;
using std::endl;

// ------ 主程序 ----------------------------------
int main()
```

```
{
  string A[] =
{"John","Memi","Brigitte","Daiana","Lulu","Lisa"};
  vector<string>  S(A, A + 6);
  // -- 定义 ostream_iterator 对象 ----------------
    ostream_iterator<string>  StringOut(cout, "  ");
  cout << "\nvector S 原来为: "  << endl;
  for (int i = 0; i < S.size(); i++)
   cout << S[i] << "  ";
  cout << endl;
  // 插入
  S.insert(find(S.begin(), S.end(), "Daiana"), "Joanne");
  cout << "\n在 Daiana 前插入 Joanne"
       << " 之后, vector S 为: "  << endl;
  for (int i = 0; i < S.size(); i++)
      cout << S[i] << "  ";
  cout << endl;
  // 删除
  std::remove( S.begin(), S.end(), "Brigitte");
  S.resize(S.size()-1);
  cout << "\n在删除 Brigitte"
       << " 之后, vector S 为: "  << endl;
  std::copy(S.begin(), S.end(), StringOut);
  cout << endl;
  // 取代
  *find(S.begin(), S.end(), "Daiana")= "William";
  cout << "\n将 Daiana 取代为 William"
      << " 之后, vector S 为: "  << endl;
  for (int i = 0; i < S.size(); i++)
   cout << S[i] << "  ";
  cout << endl;
  // 设定特定元素的值
  S.at(2) = "Peter";
  Cout << "\n在执行 "S.at(2) = Peter" 之后, S 变成为:"
       << endl;
  std::copy(S.begin(), S.end(), StringOut);
  cout << endl;
```

```
    // 插入额外的元素
    S.insert(S.begin() + 3, "Albert");
    Cout << "\n 在执行 "S.insert(S.begin() + 3, Albert)" "
         << "\n 之后, S 变成为:"<< endl;
    std::copy(S.begin(), S.end(), StringOut);
    // 将 vector 对象各元素的值依大小排序
    cout << endl;
    std::sort(S.begin(), S.end());
    cout << "\n 排序之后, S 变成为:"
         << endl;
    std::copy(S.begin(), S.end(), StringOut);
    cout << endl;
    // 将 vector 对象各元素打乱重排 (random shuffle)
    std::random_shuffle(S.begin(), S.end());
    cout << "\n 打乱重排之后, S 变成为:"
         << endl;
    std::copy(S.begin(), S.end(), StringOut);
    cout << endl;
    return 0;
}
```

程序执行结果

```
vector S 原来为:
John  Memi  Brigitte  Daiana  Lulu  Lisa

在 Daiana 前插入 Joanne 之后, vector S 为:
John  Memi  Brigitte  Joanne  Daiana  Lulu  Lisa

在删除 Brigitte 之后, vector S 为:
John  Memi  Joanne  Daiana  Lulu  Lisa

将 Daiana 取代为 William 之后, vector S 为:
John  Memi  Joanne  William  Lulu  Lisa
在执行 "S.at(2) = Peter" 之后, S 变成为:
John  Memi  Peter  William  Lulu  Lisa
```