STL 简介，标准模板库

作者：Scott Field

这篇文章是关于C++语言的一个新的扩展——标准模板库的（Standard Template Library），也叫STL。

当我第一次打算写一篇关于STL的文章的时候，我不得不承认我当时低估了这个话题的深度和广度。有很多内容要含盖，也有很多详细描述STL的书。因此我重新考虑了一下我原来的想法。我为什么要写这篇文章，又为什么要投稿呢？这会有什麽用呢？有再来一篇关于STL的文章的必要吗？

当我翻开Musser and Saini的页时，我看到了编程时代在我面前消融。我能看到深夜消失了，目标软件工程出现了。我看到了可维护的代码。一年过去了，我使用STL写的软件仍然很容易维护。让人吃惊的是其他人可以没有我而维护的很好！

然而，我也记得在一开始的时候很难弄懂那些技术术语。一次，我买了Musser&Saini，每件事都依次出现，但是在那以前我最渴望得到的东西是一些好的例子。

当我开始的时候，作为C++一部分的Stroustrup还没出来，它覆盖了STL。

因此我想写一篇关于一个STL程序员的真实生活的文章可能会有用。如果我手上有一些好的例子的话，特别是象这样的新题目，我会学的更快。

另外一件事是STL应该很好用。因此，理论上说，我们应该可以马上开始使用STL。

什麽是STL呢？STL就是Standard Template Library，标准模板库。这可能是一个历史上最令人兴奋的工具的最无聊的术语。从根本上说，STL是一些“容器”的集合，这些“容器”有list,vector,set,map等，STL也是算法和其他一些组件的集合。这里的“容器”和算法的集合指的是世界上很多聪明人很多年的杰作。

STL的目的是标准化组件，这样你就不用重新开发它们了。你可以仅仅使用这些现成的组件。STL现在是C++的一部分，因此不用额外安装什麽。它被内建在你的编译器之内。因为STL的list是一个简单的容器，所以我打算从它开始介绍STL如何使用。如果你懂得了这个概念，其他的就都没有问题了。另外，list容器是相当简单的，我们会看到这一点。

这篇文章中我们将会看到如何定义和初始化一个list，计算它的元素的数量，从一个list里查找元素，删除元素，和一些其他的操作。要作到这些，我们将会讨论两个不同的算法，STL通用算法都是可以操作不止一个容器的，而list的成员函数是list容器专有的操作。

这是三类主要的STL组件的简明纲要。STL容器可以保存对象，内建对象和类对象。它们会安全的保存对象，并定义我们能够操作的这个对象的接口。放在蛋架上的鸡蛋不会滚到桌上。它们很安全。因此，在STL容器中的对象也很安全。我知道这个比喻听起来很老土，但是它很正确。

STL算法是标准算法，我们可以把它们应用在那些容器中的对象上。这些算法都有很著名的执行特性。它们可以给对象排序，删除它们，给它们记数，比较，找出特殊的对象，把它们合并到另一个容器中，以及执行其他有用的操作。

STL iterator就象是容器中指向对象的指针。STL的算法使用iterator在容器上进行操作。Iterator设置算法的边界，容器的长度，和其他一些事情。举个例子，有些iterator仅让算法读元素，有一些让算法写元素，有一些则两者都行。 Iterator也决定在容器中处理的方向。

你可以通过调用容器的成员函数begin()来得到一个指向一个容器起始位置的iterator。你可以调用一个容器的 end() 函数来得到过去的最后一个值（就是处理停在那的那个值）。

这就是STL所有的东西，容器、算法、和允许算法工作在容器中的元素上的iterator。算法以合适、标准的方法操作对象，并可通过iterator得到容器精确的长度。一旦做了这些，它们就在也不会“跑出边界”。还有一些其他的对这些核心组件类型有功能性增强的组件，例如函数对象。我们将会看到有关这些的例子，现在 ，我们先来看一看STL的list。

定义一个list

我们可以象这样来定义一个STL的list：

```
#include <string>
#include <list>
int main (void) {
  list<string> Milkshakes;
}
```

这就行了，你已经定义了一个list。简单吗？list<string> Milkshakes这句是你声明了list<string>模板类 的一个实例，然后就是实例化这个类的一个对象。但是我们别急着做这个。在这一步其实你只需要知道你定义了一个字符串的list。你需要包含提供STL list类的头文件。我用gcc 2.7.2在我的Linux上编译这个测试程序，例如：

```
g++ test1.cpp -otest1
```

注意iostream.h这个头文件已经被STL的头文件放弃了。这就是为什么这个例子中没有它的原因。

现在我们有了一个list，我们可以看实使用它来装东西了。我们将把一个字符串加到这个list里。有一个非常重要的东西叫做list的值类型。值类型就是list中的对象的类型。在这个例子中,这个list的值类型就是字符串,string ， 这是因为这个list用来放字符串。

I使用list的成员函数push_back和push_front插入一个元素到list中

```
#include <string>
#include <list>
#
int main (void) {
  list<string> Milkshakes;
  Milkshakes.push_back("Chocolate");
  Milkshakes.push_back("Strawberry");
  Milkshakes.push_front("Lime");
  Milkshakes.push_front("Vanilla");
}
```

We now have a list with four strings in it. The list member function push_back() places an object onto the back of the list. The list member function push_front() puts one on the front. I often push_back() some error messages onto a list, and then push_front() a title on the list so it prints before the error messages. 我们现在有个4个字符串在list中。list的成员函数push_back()把一个对象放到一个list的后面，而push_front()把对象放到前面。我通常把一些错误信息push_back()到一个list中去，然后push_front()一个标题到list中，这样它就会在这个错误消息以前打印它了。

The list member function empty()list的成员函数empty()

知道一个list是否为空很重要。如果list为空，empty()这个成员函数返回真。我通常会这样使用它。通篇程序我都用push_back()来把错误消息放到list中去。然后，通过调用empty() 我就可以说出这个程序是否报告了错误。如果我定义了一个list来放信息，一个放警告，一个放严重错误，我就可以通过使用empty()轻易的说出到底有那种类型的错误发生了。

我可以整理这些list，然后在打印它们之前，用标题来整理它们，或者把它们排序成类。

这是我的意思：

```
/*
|| Using a list to track and report program messages and status
*/
#include <iostream.h>
#include <string>
#include <list>
#
int main (void) {
  #define OK 0
  #define INFO 1
  #define WARNING 2
#
  int return_code;
#
  list<string> InfoMessages;
```

```
  list<:string> WarningMessages;
#
  // during a program these messages are loaded at various points
  InfoMessages.push_back("Info: Program started");
  // do work...
  WarningMessages.push_back("Warning: No Customer records have been fo
  // do work...
 #
  return_code = OK;
 #
  if  (!InfoMessages.empty()) {          // there were info messages
     InfoMessages.push_front("Informational Messages:");
     // ... print the info messages list, we'll see how later
     return_code = INFO;
  }
#
  if  (!WarningMessages.empty()) {       // there were warning message
     WarningMessages.push_front("Warning Messages:");
     // ... print the warning messages list, we'll see how later
     return_code = WARNING;
  }
#
  // If there were no messages say so.
  if (InfoMessages.empty() && WarningMessages.empty()) {
     cout << "There were no messages " << endl;
  }
#
  return return_code;
}
```

用for循环来处理list中的元素

我们想要遍历一个list，比如打印一个中的所有对象来看看list上不同操作的结果。要一个元素一个元素的遍历一个list，我们可以这样做：

```
/*
|| How to print the contents of a simple STL list. Whew!
*/
#include <iostream.h>
#include <string>
#include <list>
#
```

```
int main (void) {
list<string> Milkshakes;
list<string>::iterator MilkshakeIterator;
#
  Milkshakes.push_back("Chocolate");
  Milkshakes.push_back("Strawberry");
  Milkshakes.push_front("Lime");
  Milkshakes.push_front("Vanilla");
 #
  // print the milkshakes
  Milkshakes.push_front("The Milkshake Menu");
  Milkshakes.push_back("*** Thats the end ***");
  for (MilkshakeIterator=Milkshakes.begin();
         MilkshakeIterator!=Milkshakes.end();
          ++MilkshakeIterator) {
    // dereference the iterator to get the element
    cout << *MilkshakeIterator << endl;
  }
}
```

这个程序定义了一个iterator，MilkshakeIterator。我们把它指向了这个list的第一个元素。这可以调用Milkshakes.begin()来作到，它会返回一个指向list开头的iterator。然后我们把它和Milkshakes.end()的返回值来做比较，当我们到了那儿的时候就停下来。

容器的end()函数会返回一个指向容器的最后一个位置的iterator。当我们到了那里，就停止操作。我们不能不理容器的end()函数的返回值。我们仅知道它意味着已经处理到了这个容器的末尾，应该停止处理了。所有的STL容器都要这样做。

在上面的例子中，每一次执行for循环，我们就重复引用iterator来得到我们打印的字符串。

在STL编程中，我们在每个算法中都使用一个或多个iterator。我们使用它们来存取容器中的对象。要存取一个给定的对象，我们把一个iterator指向它，然后间接引用这个iterator。

这个list容器，就象你所想的，它不支持在iterator加一个数来指向隔一个的对象。就是说，我们不能用Milkshakes.begin()+2来指向list中的第三个对象，因为STL的list是以双链的list来实现的，它不支持随机存取。vector和deque(向量和双端队列)和一些其他的STL的容器可以支持随机存取。

上面的程序打印出了list中的内容。任何人读了它都能马上明白它是怎麽工作的。它使用标准的iterator和标准的list容器。没有多少程序员依赖它里面装的东西， 仅仅是标准的C++。这是一个向前的重要步骤。这个例子使用STL使我们的软件更加标准。

用STL的通用算法for_each来处理list中的元素

使用STL list和 iterator，我们要初始化、比较和给iterator增量来遍历这个容器。STL通用的for_each 算法能够减轻我们的工作。

```
/*
|| How to print a simple STL list MkII
*/
#include <iostream.h>
#include <string>
#include <list>
#include <algorithm>
#
PrintIt (string& StringToPrint) {
  cout << StringToPrint << endl;
}
#
int main (void) {
  list<string> FruitAndVegetables;
  FruitAndVegetables.push_back("carrot");
  FruitAndVegetables.push_back("pumpkin");
  FruitAndVegetables.push_back("potato");
  FruitAndVegetables.push_front("apple");
  FruitAndVegetables.push_front("pineapple");
 #
  for_each  (FruitAndVegetables.begin(), FruitAndVegetables.end(), Pri
}
```

在这个程序中我们使用STL的通用算法for_each()来遍历一个iterator的范围，然后调用PrintIt()来处理每个对象。我们不需要初始化、比较和给iterator增量。for_each()为我们漂亮的完成了这些工作。我们执行于对象上的操作被很好的打包在这个函数以外了，我们不用再做那样的循环了，我们的代码更加清晰了。

for_each算法引用了iterator范围的概念，这是一个由起始

iterator和一个末尾iterator指出的范围。起始iterator指出操作由哪里开始，末尾iterator指明到哪结束，但是它不包括在这个范围内。

用STL的通用算法count()来统计list中的元素个数。

STL的通用算法count()和count_it()用来给容器中的对象记数。就象for_each()一样，count()和count_if() 算法也是在iterator范围内来做的。

让我们在一个学生测验成绩的list中来数一数满分的个数。这是一个整型的List。

```
/*
|| How to count objects in an STL list
*/
#include <list>
#include <algorithm>
#
int main (void) {
  list<int> Scores;
#
  Scores.push_back(100); Scores.push_back(80);
  Scores.push_back(45); Scores.push_back(75);
  Scores.push_back(99); Scores.push_back(100);
#
  int NumberOf100Scores(0);
  count (Scores.begin(), Scores.end(), 100, NumberOf100Scores);
#
  cout << "There were " << NumberOf100Scores << " scores of 100" << er
}
```

The count() algorithm counts the number of objects equal to a certain value. In the above example it checks each integer object in a list against 100. It increments the variable NumberOf100Scores each time a container object equals 100. The output of the program is count()算法统计等于某个值的对象的个数。上面的例子它检查list中的每个整型对象是不是100。每次容器中的对象等于100，它就给NumberOf100Scores加1。这是程序的输出：

```
  There were 2 scores of 100
```

用STL的通用算法count_if()来统计list中的元素个数

count_if()是count()的一个更有趣的版本。他采用了STL的一个新组件，函数对象。count_if() 带一个函数对象的参数。函数对象是一个至少带有一个operator()方法的类。有些STL算法作为参数接收函数对象并调用这个函数对象的operator()方法。

函数对象被约定为STL算法调用operator时返回true或false。它们根据这个来判定这个函数。举个例子会说的更清楚些。count_if()通过传递一个函数对象来作出比count()更加复杂的评估以确定一个对象是否应该被记数。在这个例子里我们将数一数牙刷的销售数量。我们将提交包含四个字符的销售码和产品说明的销售记录。

```
/*
|| Using a function object to help count things
*/
#include <string>
#include <list>
#include <algorithm>
#
const string ToothbrushCode("0003");
#
class IsAToothbrush {
public:
bool operator() ( string& SalesRecord ) {
    return SalesRecord.substr(0,4)==ToothbrushCode;
  }
};
#
int main (void) {
  list<string> SalesRecords;
#
  SalesRecords.push_back("0001 Soap");
  SalesRecords.push_back("0002 Shampoo");
  SalesRecords.push_back("0003 Toothbrush");
  SalesRecords.push_back("0004 Toothpaste");
  SalesRecords.push_back("0003 Toothbrush");
 #
  int NumberOfToothbrushes(0);
  count_if (SalesRecords.begin(), SalesRecords.end(),
```

```
            IsAToothbrush(), NumberOfToothbrushes);
#
  cout << "There were "
       << NumberOfToothbrushes
       << " toothbrushes sold" << endl;
}
```

这是这个程序的输出：

```
There were 2 toothbrushes sold（一共卖了两把牙刷）
```

这个程序是这样工作的：定义一个函数对象类IsAToothbrush，这个类的对象能判断出卖出的是否是牙刷。如果这个记录是卖出牙刷的记录的话，函数调用operator()返回一个true，否则返回false。

count_if()算法由第一和第二两个iterator参数指出的范围来处理容器对象。它将对每个 IsAToothbrush()返回true的容器中的对象增加NumberOfToothbrushes的值。

最后的结果是NumberOfToothbrushes这个变量保存了产品代码域为"0003"的记录的个数，也就是牙刷的个数。

注意count_if()的第三个参数IsAToothbrush()，它是由它的构造函数临时构造的一个对象。你可以把IsAToothbrush类的一个临时对象传递给count_if()函数。count_if()将对该容器的每个对象调用这个函数。

使用count_if()的一个更加复杂的函数对象。

我们可以更进一步的研究一下函数对象。假设我们需要传递更多的信息给一个函数对象。我们不能通过调用operator来作到这点，因为必须定义为一个list的中的对象的类型。 然而我们通过为IsAToothbrush指出一个非缺省的构造函数就可以用任何我们所需要的信息来初始化它了。 例如，我们可能需要每个牙刷有一个不定的代码。我们可以把这个信息加到下面的函数对象中：

```
/*
|| Using a more complex function object
*/
#include <iostream.h>
#include <string>
```

```
#include <list>
#include <algorithm>
#
class IsAToothbrush {
public:
  IsAToothbrush(string& InToothbrushCode) :
      ToothbrushCode(InToothbrushCode) {}
  bool operator() (string& SalesRecord) {
    return SalesRecord.substr(0,4)==ToothbrushCode;
}
private:
  string ToothbrushCode;
};
#
int main (void) {
  list<string> SalesRecords;
#
  SalesRecords.push_back("0001 Soap");
  SalesRecords.push_back("0002 Shampoo");
  SalesRecords.push_back("0003 Toothbrush");
  SalesRecords.push_back("0004 Toothpaste");
  SalesRecords.push_back("0003 Toothbrush");
 #
  string VariableToothbrushCode("0003");
#
  int NumberOfToothbrushes(0);
  count_if (SalesRecords.begin(), SalesRecords.end(),
              IsAToothbrush(VariableToothbrushCode),
                 NumberOfToothbrushes);
  cout << "There were  "
       << NumberOfToothbrushes
       << " toothbrushes matching code "
       << VariableToothbrushCode
       << " sold"
       << endl;
}
```

程序的输出是：

```
There were  2 toothbrushes matching code 0003 sold
```

这个例子演示了如何向函数对象传递信息。你可以定义任意你想要的构造函数，你可以再函数对象中做任何你想做的处理，都可以合法编译通过。

你可以看到函数对象真的扩展了基本记数算法。

到现在为止，我们都学习了：

- 定义一个list
- 向list中加入元素
- 如何知道list是否为空
- 如何使用for循环来遍历一个list
- 如何使用STL的通用算法for_each来遍历list
- list成员函数begin() 和 end() 以及它们的意义
- iterator范围的概念和一个范围的最后一个位置实际上并不被处理这一事实
- 如何使用STL通用算法count()和count_if()来对一个list中的对象记数
- 如何定义一个函数对象

我选用这些例子来演示list的一般操作。如果你懂了这些基本原理，你就可以毫无疑问的使用STL了建议你作一些练习。我们现在用一些更加复杂的操作来扩展我们的知识，包括list成员函数和STL通用算法。

使用STL通用算法find()在list中查找对象

我们如何在list中查找东西呢？STL的通用算法find()和find_if()可以做这些。就象for_each(), count(), count_if() 一样，这些算法也使用iterator范围，这个范围指出一个list或任意其他容器中的一部分来处理。通常首iterator指着开始的位置，次iterator指着停止处理的地方。由次iterator指出的元素不被处理。

这是find()如何工作：

```
/*
|| How to find things in an STL list
*/
#include <string>
#include <list>
#include <algorithm>
#
int main (void) {
  list<string> Fruit;
```

```
  list<string>::iterator FruitIterator;
#
  Fruit.push_back("Apple");
  Fruit.push_back("Pineapple");
  Fruit.push_back("Star Apple");
#
  FruitIterator = find (Fruit.begin(), Fruit.end(),
"Pineapple");
#
  if (FruitIterator == Fruit.end()) {
    cout << "Fruit not found in list" << endl;
  }
  else {
   cout << *FruitIterator << endl;
  }
}
```

输出是:

```
Pineapple
```

如果没有找到指出的对象，就会返回Fruit.end()的值，要是找到了就返回一个指着找到的对象的iterator

使用STL通用算法find_if()在list中搜索对象

这是find()的一个更强大的版本。这个例子演示了find_if()，它接收一个函数对象的参数作为参数，并使用它来做更复杂的评价对象是否和给出的查找条件相付。

假设我们的list中有一些按年代排列的包含了事件和日期的记录。我们希望找出发生在1997年的事件。

```
/*
|| How to find things in an STL list MkII
*/
#include <string>
#include <list>
#include <algorithm>
#
```

```
class EventIsIn1997 {
public:
 bool operator () (string& EventRecord) {
   // year field is at position 12 for 4 characters in EventRecord
   return EventRecord.substr(12,4)=="1997";
  }
};
#
int main (void) {
  list<string> Events;
#
// string positions 012345678901234567890123456789012345678901234567890123456789012345
  Events.push_back("07 January  1995  Draft plan of house prepared");
  Events.push_back("07 February 1996  Detailed plan of house prepared");
  Events.push_back("10 January  1997  Client agrees to job");
  Events.push_back("15 January  1997  Builder starts work on bedroom");
  Events.push_back("30 April    1997  Builder finishes work");
 #
  list<string>::iterator EventIterator =
      find_if (Events.begin(), Events.end(), EventIsIn1997());
#
  // find_if completes the first time EventIsIn1997()() returns true
  // for any object. It returns an iterator to that object which we
  // can dereference to get the object, or if EventIsIn1997()() never
  // returned true, find_if returns end()
  if (EventIterator==Events.end()) {
    cout << "Event not found in list" << endl;
  }
  else {
   cout << *EventIterator << endl;
  }
}
```

这是程序的输出：

```
10 January  1997  Client agrees to job
```

使用STL通用算法search在list中找一个序列

一些字符在STL容器中很好处理，让我们看一看一个难处理的字符序列。我们将定义一个list来放字符。

```
  list<char> Characters;
```

现在我们有了一个字符序列，它不用任何帮助就知道然后管理内存。它知道它是从哪里开始、到哪里结束。它非常有用。我不知道我是否说过以null结尾的字符数组。

让我们加入一些我们喜欢的字符到这个list中：

```
  Characters.push_back('\0');
  Characters.push_back('\0');
  Characters.push_back('1');
  Characters.push_back('2');
```

我们将得到多少个空字符呢？

```
  int NumberOfNullCharacters(0);
  count(Characters.begin(), Characters.end(), '\0', NumberOfNullCharac
  cout << "We have " << NumberOfNullCharacters << endl;
```

让我们找字符'1'

```
  list<char>::iterator Iter;
  Iter = find(Characters.begin(), Characters.end(), '1');
  cout << "We found " << *Iter << endl;
```

这个例子演示了STL容器允许你以更标准的方法来处理空字符。现在让我们用STL的search算法来搜索容器中的两个null。

就象你猜的一样，STL通用算法search()用来搜索一个容器，但是是搜索一个元素串，不象find()和find_if() 只搜索单个的元素。

```
/*
|| How to use the search algorithm in an STL list
*/
#include <string>
#include <list>
#include <algorithm>
#
int main ( void ) {
#
  list<char> TargetCharacters;
```

```
  list<char> ListOfCharacters;
#
  TargetCharacters.push_back('\0');
  TargetCharacters.push_back('\0');
#
  ListOfCharacters.push_back('1');
  ListOfCharacters.push_back('2');
  ListOfCharacters.push_back('\0');
  ListOfCharacters.push_back('\0');
#
  list<char>::iterator PositionOfNulls =
    search(ListOfCharacters.begin(), ListOfCharacters.end(),
           TargetCharacters.begin(), TargetCharacters.end());
#
  if (PositionOfNulls!=ListOfCharacters.end())
    cout << "We found the nulls" << endl;
}
```

The output of the program will be 这是程序的输出：

```
We found the nulls
```

search算法在一个序列中找另一个序列的第一次出现的位置。在这个例子里我们在ListOfCharacters中找TargetCharacters这个序列的第一次出现，TargetCharacters是包含两个null字符的序列。

search的参数是两个指着查找目标的iterator和两个指着搜索范围的iterators。因此我们我们在整个的ListOfCharacters的范围内查找TargetCharacters这个list的整个序列。

如果TargetCharacters被发现，search就会返回一个指着ListOfCharacters中序列匹配的第一个字符的iterator。如果没有找到匹配项，search返回ListOfCharacters.end()的值。

使用list的成员函数sort()排序一个list。

要排序一个list，我们要用list的成员函数sort()，而不是通用算法sort()。所有我们用过的算法都是通用算法。然而，在STL中有时容器支持它自己对一个特殊算法的实现，这通常是为了提高性能。

在这个例子中，list容器有它自己的sort算法，这是因为通用算

法仅能为那些提供随机存取里面元素的容器排序，而由于list是作为一个连接的链表实现的，它不支持对它里面的元素随机存取。所以就需要一个特殊的 sort()成员函数来排序list。

由于各种原因，容器在性能需要较高或有特殊效果需求的场合支持外部函数(extra functions)， 这通过利用构造函数的结构特性可以作到。

```
/*
|| How to sort an STL list
*/
#include <string>
#include <list>
#include <algorithm>
#
PrintIt (string& StringToPrint) { cout << StringToPrint << endl;}
#
int main (void) {
  list<string> Staff;
  list<string>::iterator PeopleIterator;
#
  Staff.push_back("John");
  Staff.push_back("Bill");
  Staff.push_back("Tony");
  Staff.push_back("Fidel");
  Staff.push_back("Nelson");
#
  cout << "The unsorted list " << endl;
  for_each(Staff.begin(), Staff.end(), PrintIt );
#
  Staff.sort();
#
  cout << "The sorted list " << endl;
  for_each(Staff.begin(), Staff.end(), PrintIt);
}
```

输出是:

```
The unsorted list
John
Bill
Tony
Fidel
```

```
Nelson
The sorted list
Bill
Fidel
John
Nelson
Tony
```

用list的成员函数插入元素到list中

list的成员函数push_front()和push_back()分别把元素加入到list的前面和后面。你可以使用insert() 把对象插入到list中的任何地方。

insert()可以加入一个对象，一个对象的若干份拷贝，或者一个范围以内的对象。这里是一些插入对象到list中的例子：

```
/*
|| Using insert to insert elements into a list.
*/
#include <list>
#
int main (void) {
  list<int> list1;
#
  /*
  || Put integers 0 to 9 in the list
  */
  for (int i = 0; i < 10; ++i)  list1.push_back(i);
#
  /*
  || Insert -1 using the insert member function
  || Our list will contain -1,0,1,2,3,4,5,6,7,8,9
  */
  list1.insert(list1.begin(), -1);
#
  /*
  || Insert an element at the end using insert
  || Our list will contain -1,0,1,2,3,4,5,6,7,8,9,10
  */
  list1.insert(list1.end(), 10);
 #
```

```
  /*
  || Inserting a range from another container
  || Our list will contain -1,0,1,2,3,4,5,6,7,8,9,10,11,12
  */
  int IntArray[2] = {11,12};
  list1.insert(list1.end(), &IntArray[0], &IntArray[2]);
#
  /*
  || As an exercise put the code in here to print the lists!
  || Hint: use PrintIt and accept an interger
  */
}
```

注意，insert()函数把一个或若干个元素插入到你指出的iterator的位置。你的元素将出现在 iterator指出的位置以前。

List 构造函数

我们已经象这样定义了list：

```
  list<int> Fred;
```

你也可以象这样定义一个list，并同时初始化它的元素：

```
  // define a list of 10 elements and initialise them all to 0
  list<int> Fred(10, 0);
  // list now contains 0,0,0,0,0,0,0,0,0,0
```

或者你可以定义一个list并用另一个STL容器的一个范围来初始化它，这个STL容器不一定是一个list，仅仅需要是元素类型相同的的容器就可以。

```
  vector<int> Harry;
  Harry.push_back(1);
  Harry.push_back(2);
#
  // define a list and initialise it with the elements in Harry
  list<int> Bill(Harry.begin(), Harry.end());
  // Bill now contains 1,2
```

使用list成员函数从list中删除元素

list成员函数pop_front()删掉list中的第一个元素，pop_back()删掉最后一个元素。函数erase()删掉由一个iterator指出的元素。还有另一个erase()函数可以删掉一个范围的元素。

```
/*
|| Erasing objects from a list
*/
#include <list>
#
int main (void) {
  list<int> list1;   // define a list of integers
#
  /*
  || Put some numbers in the list
  || It now contains 0,1,2,3,4,5,6,7,8,9
  */
  for (int i = 0; i < 10; ++i)  list1.push_back(i);
#
  list1.pop_front();    // erase the first element 0
#
  list1.pop_back();     // erase the last element 9
 #
  list1.erase(list1.begin());  // erase the first element (1) using ar
#
  list1.erase(list1.begin(), list1.end());  // erase all the remaining
#
  cout << "list contains " << list1.size() << " elements" << endl;
}
```

输出是:

```
list contains 0 elements
```

用list成员函数remove()从list中删除元素。

list的成员函数remove()用来从list中删除元素。

```
/*
|| Using the list member function remove to remove elements
*/
#include <string>
#include <list>
```

```
#include <algorithm>
#
PrintIt (const string& StringToPrint) {
  cout << StringToPrint << endl;
}
#
int main (void) {
  list<string> Birds;
#
  Birds.push_back("cockatoo");
  Birds.push_back("galah");
  Birds.push_back("cockatoo");
  Birds.push_back("rosella");
  Birds.push_back("corella");
#
  cout << "Original list with cockatoos" << endl;
  for_each(Birds.begin(), Birds.end(), PrintIt);
 #
  Birds.remove("cockatoo");
#
  cout << "Now no cockatoos" << endl;
  for_each(Birds.begin(), Birds.end(), PrintIt);

}
```

输出是:

```
Original list with cockatoos
cockatoo
galah
cockatoo
rosella
corella
Now no cockatoos
galah
rosella
corella
```

使用STL通用算法remove()从list中删除元素

通用算法remove()使用和list的成员函数不同的方式工作。一般情况下不改变容器的大小。

```
/*
|| Using the generic remove algorithm to remove list elements
*/
#include <string>
#include <list>
#include <algorithm>
#
PrintIt(string& AString) { cout << AString << endl; }
#
int main (void) {
  list<string> Birds;
  list<string>::iterator NewEnd;
#
  Birds.push_back("cockatoo");
  Birds.push_back("galah");
  Birds.push_back("cockatoo");
  Birds.push_back("rosella");
  Birds.push_back("king parrot");
#
  cout << "Original list" << endl;
  for_each(Birds.begin(), Birds.end(), PrintIt);
#
  NewEnd = remove(Birds.begin(), Birds.end(), "cockatoo");
#
  cout << endl << "List according to new past the end iterator" << enc
  for_each(Birds.begin(), NewEnd, PrintIt);
#
  cout << endl << "Original list now. Care required!" << endl;
  for_each(Birds.begin(), Birds.end(), PrintIt);
}
```

The output will be

```
Original list
cockatoo
galah
cockatoo
rosella
king parrot


List according to new past the end iterator
galah
rosella
king parrot
```

```
Original list now. Care required!
galah
rosella
king parrot
rosella
king parrot
```

通用remove()算法返回一个指向新的list的结尾的iterator。从开始到这个新的结尾（不含新结尾元素）的范围包含了remove后剩下所有元素。你可以用list成员函数erase函数来删除从新结尾到老结尾的部分。

使用STL通用算法stable_partition()和list成员函数splice()来划分一个list

我们将完成一个稍微有点复杂的例子。它演示STL通用算法stable_partition()算法和一个list成员函数 splice()的变化。注意函数对象的使用和没有使用循环。 通过简单的语句调用STL算法来控制。

stable_partition()是一个有趣的函数。它重新排列元素，使得满足指定条件的元素排在不满足条件的元素前面。它维持着两组元素的顺序关系。

splice 把另一个list中的元素结合到一个list中。它从源list中删除元素。

在这个例子中，我们想从命令行接收一些标志和四个文件名。文件名必须’按顺序出现。通过使用stable_partition() 我们可以接收和文件名混为任何位置的标志，并且不打乱文件名的顺序就把它们放到一起。

由于记数和查找算法都很易用，我们调用这些算法来决定哪个标志被设置而哪个标志未被设置。我发现容器用来管理少量的象这样的动态数据。

```
/*
|| Using the STL stable_partition algorithm
```

```
|| Takes any number of flags on the command line and
|| four filenames in order.
*/
#include <string>
#include <list>
#include <algorithm>
#
PrintIt ( string& AString ) { cout << AString << endl; }
#
class IsAFlag {
public:
  bool operator () (string& PossibleFlag) {
    return PossibleFlag.substr(0,1)=="-";
  }
};
#
class IsAFileName {
public:
  bool operator () (string& StringToCheck) {
    return !IsAFlag()(StringToCheck);
  }
};
#
class IsHelpFlag {
public:
  bool operator () (string& PossibleHelpFlag) {
    return PossibleHelpFlag=="-h";
  }
};
#
int main (int argc, char *argv[]) {
#
list<string> CmdLineParameters;       // the command line parameters
list<string>::iterator StartOfFiles;  // start of filenames
list<string> Flags;                   // list of flags
list<string> FileNames;               // list of filenames
#
for (int i = 0; i < argc; ++i) CmdLineParameters.push_back(argv[i]);
#
CmdLineParameters.pop_front(); // we don't want the program name
#
// make sure we have the four mandatory file names
int NumberOfFiles(0);
count_if(CmdLineParameters.begin(), CmdLineParameters.end(),
       IsAFileName(), NumberOfFiles);
```

```
#
cout << "The "
     << (NumberOfFiles == 4 ? "correct " : "wrong ")
     << "number ("
     << NumberOfFiles
     << ") of file names were specified" << endl;
 #
// move any flags to the beginning
StartOfFiles =
  stable_partition(CmdLineParameters.begin(), CmdLineParameters.end(),
         IsAFlag());
#
cout << "Command line parameters after stable partition" << endl;
for_each(CmdLineParameters.begin(), CmdLineParameters.end(), PrintIt);
#
// Splice any flags from the original CmdLineParameters list into Flag
Flags.splice(Flags.begin(), CmdLineParameters,
       CmdLineParameters.begin(), StartOfFiles);
#
if (!Flags.empty()) {
  cout << "Flags specified were:" << endl;
  for_each(Flags.begin(), Flags.end(), PrintIt);
}
else {
  cout << "No flags were specified" << endl;
}
#
// parameters list now contains only filenames. Splice them into FileN
FileNames.splice(FileNames.begin(), CmdLineParameters,
       CmdLineParameters.begin(), CmdLineParameters.end());
#
if (!FileNames.empty()) {
  cout << "Files specified (in order) were:" << endl;
  for_each(FileNames.begin(), FileNames.end(), PrintIt);
}
else {
  cout << "No files were specified" << endl;
}
#
// check if the help flag was specified
if (find_if(Flags.begin(), Flags.end(), IsHelpFlag())!=Flags.end()) {
  cout << "The help flag was specified" << endl;
}
#
// open the files and do whatever you do
```

```
#
}
```

给出这样的命令行：

```
test17 -w linux -o is -w great
```

输出是：

```
The wrong number (3) of file names were specified
Command line parameters after stable partition
-w
-o
-w
linux
is
great
Flags specified were:
-w
-o
-w
Files specified (in order) were:
linux
is
great
```

结论

我们仅仅简单的谈了谈你可以用list做的事情。我们没有说明一个对象的用户定义类，虽然这个不难。

如果你懂了刚才说的这些算法背后的概念，那么你使用剩下的那些算法就应该没有问题了。使用STL 最重要的东西就是得到基本理论。

STL的关键实际上是iterator。STL算法作为参数使用iterator，他们指出一个范围，有时是一个范围，有时是两个。STL容器支持iterator，这就是为什么我们说 list<int>::iterator, 或 list<char>::iterator, 或 list<string>::iterator.

iterator有很好的定义继承性。它们非常有用。某些iterator仅

支持对一个容器只读，某些仅支持写，还有一些仅能向前指，有一些是双向的。有一些iterator支持对一个容器的随机存取。

STL算法需要某个iterator作为“动力” 如果一个容器不提供iterator作为“动力”，那么这个算法将无法编译。例如，list容器仅提供双向的 iterator。通常的sort()算法需要随机存取的iterator。这就是为什么我们需要一个特别的list成员函数sort()。

要合适的实际使用STL，你需要仔细学习各种不同的iterator。你需要知道每种容器都支持那类iterator。你还需要知道算法需要那种iterator，你当然也需要懂得你可以有那种iterator。

在field中使用STL

去年，我曾用STL写过几个商业程序。它在很多方面减少了我的工作量，也排除了很多逻辑错误。

最大的一个程序有大约5000行。可能最惊人的事情就是它的速度。它读入并处理一个1-2兆的报告文件仅花大约20秒。我是在linux上用gcc2.7.2开发的，现在运行在HP-UX机器上。它一共用了大约50和函数对象和很多容器，这些容器的大小从小list到一个有14,000个元素的map都有。

一个程序中的函数对象是处于一个继承树中，顶层的函数对象调用低层的函数对象。我大量的使用STL算法for_each() ,find (),find_if(),count()和count_if()，我尽量减少使用程序内部的函数，而使用STL的算法调用。

STL倾向于自动的把代码组织成清晰的控制和支持模块。通过小心使用函数对象并给它们起有意义的名字，我使它们在我的软件的控制流中流动。

还有很多关于STL编程要知道的东西，我希望你通过这些例子可以愉快的工作。

参考数目中的两本书在web上都有勘误表，你可以自己改正它们。

Stroustrup在每一章后面都有个建议栏，特别是对于出学者有用。正本书比早期的版本更加健谈。它也更大了。书店里还可以找到

其他几本关于STL的教科书。去看看，也许你能发现什么。

参考书目

The STL Tutorial and Reference Guide, David Musser and Atul Saini. Addison Wesley 1996. 《STL教程和参考手册》

The C++ Programming Language 3e, Bjarne Stroustrup. Addison Wesley 1997.