# 第1章 简介

## 1.1 背景知识

库(library)是一系列组件的集合，它们可以在不同的程序中重复使用。C++语言按照传统的习惯，提供了各种各样的函数(function)所组成的库，用以完成诸如输入/输出、数学计算等功能。这其中的某些函数直接来源于标准的C库(如printf)，其他则专属于标准的C++库(如set\_new\_handler)。然而，不论其来源如何，库函数都遵守以下规则：接受一些预先符合指定类型的参数，返回一个特定类型的值或改变一些已有的值。设计一个能被广泛使用的C或C++库的一个重要组成部分就是，估计出这些函数中最有可能出现的参数组合情况。

C++与C相比较而言有着重大的改进。它将C中的数据结构的概念进行了扩展，使得它不但能包含成员对象，还能包含成员函数。C++中的类(class)可以更好地将一些相关的数据及在其上进行的操作封装在一起。设计一个能被广泛使用的C++库的另外一个重要组成部分就是估计出这些类中最有可能出现的成员对象类型的组合情况。

在其最新的发展过程中，C++新增了模板(template)这个特性。通过使用模板，C++允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化(specialize)的时候。可以定义模板类及模板函数。在一定的限制下，模板类或模板函数可以允许程序员针对不同的类型提取出特定的、一般化的代码为这些类型服务。正如你所想到的，模板库相对于传统的、由函数及类组成的库来说，可以提供更好的代码重用机制。当然，标准化委员会也不会忽视这些机制，所以它也被标准化委员会添加在C++标准中。

#### 1.1.1 ANSI X3J16 ISO WG21

ANSI(American National Standards Instiute，美国国家标准学会)负责制定计算机编程语言的美国标准。X3J16是经ANSI组织授权制定C++语言标准的标准化委员会的名称，它于1989年正式启动，现在我们称它为J16。ISO(International Organisation for Standardization，国际标准化组织)是一个与ANSI相似的组织，不过它负责的是制定全球化的标准。ISO于1991年成立了技术委员会JTC1/SC22/WG21，它协同X3J16一起工作，目的是制定出一个既符合ANSI要求，又符合ISO要求的C++标准。在本书中，“标准化委员会”同时指代X3J16/J16与WG21。在过去的8年间，标准化委员会每年都举行三次会议，一次持续一周，每次会议都会对C++标准的制定过程产生重大的影响。

我们现在所见的C++标准的最后一次技术上的修改完成于1997年11月，其官方名称为ISO 14882。这意味着从现在起，这份文档将进入一个相对稳定的阶段——一般来说，每一份经由ISO制定出来的标准都将在以后的5年间(至少是5年)被冻结。然而，在这份草案被制定为标准前，标准化委员会还是对它做了一些较大的改动，扩充了其中有关库的部分条款。其中一个比较大的改动就是将模板引入标准库，使用模板类来代替传统的C++中定义的类。但是对标准库的最大的一个改动出现在1994年7月。1994年7月，标准化委员会通过投票决定，将STL加入到C++的标准中。

#### 1.1.2 STL

STL(Standard Template Library，标准模板库)是惠普实验室开发的一系列软件的统称。它是由Alexander Stepanov(现在AT&T实验室工作)、Meng Lee(仍在惠普实验室工作)和David R Musser(现在Rensselaer Polytechnic Institute工作)在惠普实验室工作时所开发出来的(S&L95和M&S96)。现在虽说它主要出现在C++中，但在被引入到C++之前该技术就已经存在了很长的一段时间。实际上，Musser和Stepanon在十多年前就已经向世人展示了它的一个早期实现，那时他们用以完成STL的工具是Ada中的generics(这也是模板的一种形式，M&S87和M&S89)。这些工作也是基于他们两人在更早的时期所完成的一些工作。

如同我们在这里所展示的一样，惠普实验室最初的工作产生了超过6000行的、非常精致的代码。几乎所有的代码都采用了模板类及模板函数的方式。可以为这些模板指定各种类型来进行特化，这些类型可以是语言本身提供的内置(built-in)类型，也可以是自己所定义的类。通过指定类型对模板进行特化所产生的代码与自己所写的代码非常接近。而且对于STL中所提供的数据结构和算法，你也很难自己编写代码来正确地完成它们。因此我们可以尽情地享受STL带来的好处，使用它所提供的精巧的方案来解决常见问题，从而获得在代码空间及执行时间上的效率。

STL的代码从广义上大致分为三类：算法(algorithm)、容器(container)和迭代器(iterator)。在这些分类中，迭代器可能是其中最有趣、最有创造性的一类，不过也是最难解释清楚的一类。因此，我们在此先介绍算法和容器。

#### 1.1.3 算法

严格地说，算法是用以决定结果的一个数学过程。它与数学中的函数在某些方面是有区别的。可能对于我们来说最重要的还是，算法是由有限次数的操作所完成的。相比之下，函数在概念上来说可以执行无限次数的操作，然后得到一个定义好的结果。

在实际的编程过程中，我们经常随意使用“算法”这个术语。在C/C++中，我们经常用它来表示那些能够得到有用结果的函数。好的算法在完成一件工作时能够有效地减少所需要的操作步骤和所需要使用的存储空间，并且还能够提供一个可以预期的时间复杂度。它应该谨慎地处理一些特殊情况，如进行0次的重复计数等。它还应该避免出现在执行期间的溢出或其他不正常的计算情况。它应该是高内聚的，对外只提供一个合理的尽量小的接口。

对于程序员来说，写出好的算法是一笔很重要的财富。例如，我们一直都在大量地使用标准C库。所有数学函数、字符串操作符及I/O实现程序都使用那些被时间证明具有非常好的可重用性的算法。

正如我们在最开始所说的，函数对数据类型的选择在其可重用性方面起着关键性的作用。一个用来求平方根的函数，在使用double作为其参数类型的情况下，其可重用性肯定比使用int作为它的参数类型要高。同样，由一系列char所组成的字符串或者文件，其可重用性也肯定比一系列float所组成的序列要高。现在，可以设想一下，当我们对这些函数所使用的数据有自己设定的类型时，它们的适用性又如何呢？

这时我们就需要借助模板的强大功能了。仅仅是多了一点点限制，就可以使用一个模板函数或类来为不同的类型服务，至于填空的工作交给翻译器就可以了。使用模板时需要遵循的限制条件是由能够怎样操作的参数类型所决定的。举例来说，如果模板需要对参数类型的对象“加一”的话，那么你所选择的类型对于那种类型的对象“加一”意味着什么应该有着良好的定义。

STL的一个重大成就在于，它提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的——你可以将所有的类型划分为少数的几类，然后就可以在模板的参数中使用一种类型轻易地替换掉同一种类中的其他类型。每一个种类都拼写的很清楚，编写的模板定义也十分健壮。也就是说，对于给定算法可能会使用到的类型通常都能够得到正确的结果，而且通常代码也能完成你想要它完成的功能。

STL提供了大约100个实现算法的模板函数。其中简单的如for\_each，它将为指定序列中的每一个元素调用指定的函数；复杂的如stable\_sort，它以你所指定的规则对序列进行排序，如果在序列中有两个元素在给定规则下相等，那么使用stable\_srot排序不会导致它们在序列中的前后关系发生改变。

熟悉了STL后，你就会发现：以前说的许多“有趣功能”的代码现在只需要用短短几行就能完全替换了。通过使用一两个算法模板，就可以得到一些优雅且高效的代码，它同样也能够完成你所需要的功能，并且比你自己的代码所完成的还要好。

#### 1.1.4 容器

在实际的开发过程中，组织数据和选择操作与这些数据上的算法几乎一样重要。组织一系列元素的方式将直接影响到诸如添加、删除、多模式存取以及重新组织它们等操作的复杂程序和时间消耗。事实上，当程序中存在着对时间要求很高的部分时，数据结构的选择就显得尤为重要，有时它们甚至会直接导致程序的成败。

在物理学中也存在着这样类似的问题。在物理学中有一些计算非常复杂，尤其是那些需要在边界匹配两种解决方法的运算更是如此。对于这种问题的一种解决方法是改变当前所使用的坐标系统，使得原来系统中复杂且难处理的边界值在新的系统中变得简单而且易于处理。当然，坐标系统之间的转换可能会比较复杂。这样做的优势在于，你要解决的问题可能在以前就有人解决过了，你所需要的只不过是将已有的解法拿来重新利用一番而已。

在计算过程中，我们不断地重复发现一些有用的方法用于组织一系列数据。也可以把它们想象成和上面讨论的坐标系统类似的系统。数组能够很方便地完成随机存取的操作，不过它不能简单地实现其本身的增长，因而往数组中插入元素简直就是一件麻烦事。链表对于添加和插入操作只需要做少量的动作，但存取任意元素的复杂度却成线性增长，而不像随机存取那样是一个常数，等等，

令人悲哀的是，在过去的时间中，我们不断地重复实现一些诸如向量(vector)、列表(list)等常见的数据结构。这些代码都十分相似，只是为了适应不同数据的变化而在一些细节方面有一些不同之处。那么我们自然可以这样想：如果有人能够实现一个链表解决上述问题的话，我们是不是可以重复利用这个已有的实现来实现我们自己的链表呢？如果能的话，那样做是不是很棒呢？

于是STL容器就闪亮登场了。STL容器是一些模板类，它们支持半打左右组织数据的最通常的方法。这些模板的参数允许我们指定容器中元素的数据类型。所有用于增长、收缩、插入以及存取容器中所有元素的代码一旦提供，就可以被所有容器重用。

再重复一次，慢慢地你将会发现，许多要为一个新程序所写的代码其实只是对于一些通用的数据组织方法的簿记工作，STL中提供的容器就能够完全搞定这些单调乏味的工作。

#### 1.1.5 迭代器

迭代器在STL中用来将算法和容器联系起来，起着“胶合剂”的作用。通过为C++中新近定义的类重载一些操作符，迭代器泛化了C中指针的概念。例如，考虑一下first和last这两个通常用于指定序列边界的指针：

for(sum=0; first != last; ++first)

    sum += \*first;

毫无疑问，first在此处看起来就是一个指针。不过在C++中，它不需要一定为一个指针。实际上，它可以是任意符合下列调减的类型：

* 比较两个值是否相等(first != last)
* 间接取值(dereference)，以存取某个值(\*first)
* (前)递增(++first)

当然，你也可以在这段代码中使用传统的对象指针。

通过在上例中添加几行代码，可以得到一个模板函数，然后就可以利用这个模板函数来对任意类型T所形成的元素序列求和：

template<class T, class It>

inline T Sum\_all(It first, It last)

{

    T sum;

    for(sum=0; first != last; ++first)

        sum += \*first;

    return sum;

}

如果在上述代码中再添加一些需求，我们就可以得到一个典型迭代器的规范。实际上，上面给出的这个模板函数与第8章中描述的模板函数accumulate十分相似。

几乎所有STL提供的算法都是通过迭代器存取元素序列进行工作的。对于不同的迭代器，STL中提供了一些分类方法，那些算法函数也都是围绕这分些类来定义的。正是由于这样的组织原则，使得STL具有更强的功能以及更大的弹性。

STL中的每一个容器都定义了其本身所专有的迭代器，用以存取容器中的元素。当然，也可以为自己定义的数据结构提供自己的迭代器。值得注意的是，普通的指针也可以像迭代器一样正常地工作。如果你没有兴趣，也可以不用去定义一堆奇特的类。另外，你也可以很容易地扩展已有的算法和容器，并使之在STL框架中正常工作。

#### 1.1.6 惠普实验室的代码

惠普实验室已经将其STL实现免费向公众开放，这是一件值得庆幸的好事情。从此你就可以自由地使用这些代码以及附带的文档，甚至还可以将其作为商业化产品的一部分重新发行。为此不需要付任何版税。所有的要求就是：在使用这些代码时必须同时附上他们提供的版权及责任说明，这完全是无可厚非的。

惠普实验室同样也保留了STL中的几个自适应算法的专利。他们也声明了如何使用这些专利——我们可以在没有预先获得许可及不支付版税的前提下使用它们。对于该公司在此领域所做出的巨大贡献，我们不得不给予高度评价。

然而，请注意：那些公众可获得的HP STL版本及其最新的经过重新组织的版本并没有被ISO 14482(也就是C++标准)所采纳。如同被标准化委员会所采纳的其他建议一样，相对于其最初提交给委员会的版本来说，STL的规范几经修订，早已不复当时的面貌了。那些众所周知的大大小小的变化对于STL代码的影响已经遍布了STL中几乎所有的角落。这就导致了这样的一个问题：使用现有的包编写的程序要想移植到兼容C++标准的实现中必须做一些改动。

同时，STL的组织方式也有了改变。举例来说，在惠普最初发行的版本中，STL组织为下列的48个头文件：

algo.h algobase.h bool.h bvector.h

defalloc.h deque.h faralloc.h fdeque.h

flist.h fmap.h fmltmap.h fmultset.h

fset.h function.h hdeque.h heap.h

hlist.h hmap.h hmultmap.h hmultset.h

hset.h hugalloc.h hvector.h iterator.h

lbvector.h ldeque.h list.h llist.h

lmap.h lmultmap.h lmultset.h lngalloc.h

lset.h map.h multimap.h multiset.h

neralloc.h nmap.h nmultmap.h nmultset.h

nset.h pair.h projectn.h set.h

stack.h temcbuf.h tree.h vector.h

而在C++标准中，它们被重新组织为下面的13个头文件：

algorithm deque functional iterator list

map memory numeric queue set

stack utility vector

这样的改动所带来的影响非常大，但却很容易解决。如果你碰巧有一些代码使用了HP所提供的STL版本，那么将它们改成符合标准的形式并不太难。所需要做的只是将代码所包含的文件列表稍加修改，不断地向其中加入新的头文件直到编译器不再报错为止。

更细微的变化体现在“容器是如何分配内存的”这样的地方。原则上来说，它们仅仅影响那些喜欢刨根问底的STL用户（也就是那些试图让容器使用它们自己的内存分配方法的用户，具体做法参见第4章）。而不幸的是，它们同样也会影响那些简单使用STL容器的用户。 随着STL的日益标准化，我们应该做好学习大量新条约的准备。

#### 1.1.7 商业编译器

现在的市场中存在着大量商业C++编译器，它们一般都有STL完全更新的版本。例如，本书中所提供的代码就是从Dinkum C++库的STL部分提取出来的，经过重新组织后在此处作为教学代码。这使得这些代码与Microsot Visual C++、IBM Visual Age以及其他厂商提供的C++编译器中的STL代码极为相似（造成它们之间还存在不同之处的主要原因是，代码必须能够无条件地符合编译器所提供的特性，即使是编译器本身并没有完全支持C++标准中所给定的全部语言特性）。虽然其他的厂商可能会提供一些有着不同语言的STL代码，但它们也必须符合C++标准。用于标准C++库的商业验证套件现在已经可以在市场获得了，提供这样的套件的公司包括：Dinkumware有限公司和Perennial Software等（这些套件的作用是给那些库厂商以某种荣誉。当升级以及除错工作都已经达到某种程度而不需要继续大力进行下去后，你就可以期待不同的商业化的标准C++库的新版本更加统一）。本书中所提供的代码已经通过了Dinkum C++ Proofer和Dinkum Abridged Proofer中所有的相关测试。上述的两种工具都是由Dinkumware有限公司提供的。

本书将STL视为标准C++所定义的那个巨大的库的一个相对独立的子集。而C++中的库又是被设计成为与ANSI/ISO标准C(ANSI89和ISO90)中的库一脉相承，可以协同其一起工作。因此，你可以将本书视为对P. J. Plauger以前所完成的两本书《The Standard C Library》(Pla92)和《The Draft Standard C++ Library》(Pla95)的一个后续版本。几乎所有C++程序员感兴趣的有关库的内容都可以在这三本书中找到。

本书每一章的组织结构都与本简介的结构差不多。

* 首先是一个讲述“背景知识”的小节，用于引入主题。
* “功能描述”小节用于提供一些简洁的参考信息。在阅读本书时，你可以从这节中发现很多有用的信息。因为在此节中指出了该章的教学主旨，并且通常还包含一些没有在该章中说明的参考信息。
* “使用...”小节介绍如何使用在该章中所描述的以及你可能希望使用的一些特性。
* 在“实现...”小节中讨论如何使这些特性在典型的STL实现中正常运行。在该节中将给出组成每个头文件的实际代码。
* 在“实现...”小节中讨论如何使这些特性在典型的STL实现中正常运行。在该节中将给出组成每个头文件的实际代码。
* 在“测试...”小节中讨论在测试一个STL实现的基本功能时产生的问题。在该节会给出针对每一个头文件的测试代码。也就是说，该节会提供一些简单但具有说明意义的示例代码。
* 每一章的最后是“习题”小节，它的目的是使你思考刚刚所阅读的信息。如果你把本书作为一门课程的一部分来阅读的话，那么它们也可以作为家庭作业用以检验学习成果。

与本章的主旨一致的是，本章剩下的几个小节也都是把STL作为一个整体而不是一些特定的部分来讨论的。对于各个部分的详细讨论将由本书的其他章节完成。

## 1.2 功能描述

#### 1.2.1 标准头文件

所有的标准C++库的实体都定义或声明在一个或多个标准头文件中。所有这些头文件包括那些在标准C中已经存在的头文件，以及一些由标准C++所引入的新的头文件。组成标准C++中的STL部分的那14个头文件已经在“惠普实验室的代码”一节中列出。可以使用一条include指令将标准头文件中的内容包含到程序中，如：

#include <algorithm> //include all algorithms

可以以任何顺序来包含那些标准头文件，或是将一个标准头文件包含多次，或是在两个或多个标准头文件中定义同一个宏或类型。但是，请不要在声明中包含标准头文件，不要在包含标准头文件前定义那些与关键字使用相同名字的宏。

一个标准的C++头文件通常还会包含其他一些标准的C++头文件，用以定义它所需要的数据类型。然而，一个标准的C头文件从不包含其他的标准头文件。标准头文件仅仅声明或定义那些在C++标准中所描述的与其有关的实体。

库中的每一个函数都在某个标准头文件中声明。与在标准C中的不同点在于，C标准头文件从不提供masking macro(即与函数同名的宏。它会屏蔽函数声明，并且达到同样的效果)。

#### 1.2.2 名字空间std

所有在标准C++头文件中出现的名字，除了operator new以及operator delete之外，都定义在一个叫做std的名字空间(namespace)内，或是定义在std所嵌套的名字空间内。包含一个标准的C++头文件一般不会将任何库名字引入当前的名字空间。

除非明确指出，否则你不可能在名字空间std内，或是std所嵌套的名字空间内定义自己的名字。该名字空间被保留给了标准C++库中的组件，现在如此，将来也是如此。

#### 1.2.3 自由度

在声明C++库中的类型及函数时，不同的实现有着其特定的自由：

* 在标准C库中，函数的名字可能有着extern “C++”或extern “C”这样的连接属性。也就是说，可以直接包含适当的标准C头文件，而不需要通过内联(inline)的方式来声明库实体。
* 库中属于某个类的成员函数的名字可能有多于文档中所列其他函数的函数签名。你可以确信此处讨论的函数调用与你所预期的意义，但不能依赖从库中所取得的成员函数的地址来进行函数调用（实际的类型可能与你所预期的不一样）。
* 有的库类可能拥有一些没有文档资料记载的非虚基类。实际上，一个有文档资料记载为从其他类所派生出来的类很有可能就是由其他的一些没有文档资料记载的基类所派生出来的。
* 一个被定义为某种integer类型同义词的类型也可能会与其他不同的一种integer类型完全一样。
* 库函数如果没有指定任何的异常(exception)规范的话，那么就意味着该函数可以抛出任意的异常，除非我们在函数的定义处明确限制了可能抛出的异常。

#### 1.2.4 保证

在另一方面，下面的这些限制调减能对你有所帮助：

* 标准C库不使用masking macro。只有特定的函数签名而不是函数名本身被保留下来另做其他用途。
* 只要不是属于某个类的成员，库函数的名字就没有其他附加的、没有文档资料记载的函数签名。也就是说，通过使用地址来进行函数调用时可以信赖的。
* 被描述成为虚拟的基类及成员函数就一定为虚拟的，而被描述为非虚拟的就一定不是虚拟的。
* 标准C++库中所定义的两种类型一般都不一样，除非它们被显式指定为一样。
* 库所提供的函数，包括那些可替换函数的默认版本，可以向外抛出尽可能多的异常（只要是在异常规范中所列出的异常种类就行）。库所提供的析构函数都不会向外抛出异常。标准C库中函数可能会传播异常，如qsort调用一个会抛出异常的比较函数，但qsort函数本身（以及其他一些标准C的函数）并不会抛出任何异常。

## 1.3 使用STL

为了获得最佳的可移植性，必须在每个翻译单元中都显式包含所需要的标准C++头文件。从一种实现移植到另一种实现上时，以前可以正常工作的一些代码有可能不能正常工作。通常我们可以这样认为：在一种很明显的依赖关系下，一个头文件必须包含另外一个头文件。但对于给定实现如何处理实际中的依赖关系我们仍可能猜错。

#### 1.3.1 名字空间

在C++的标准化进程中，名字空间被加入到C++语言中。你可以把名字空间想象成为类的包装器。它给那些在程序中定义的名字强加了一个层次，这一点就像我们使用目录结构来组织磁盘上的文件一样。

例如，可使用实际上的std::cin来指代名字cin。也可以换一种做法，使用下面的声明形式：

using namespace std;

这样做会使所有在库中定义的名字都暴露在当前的名字空间中，于是就不再需要在这些名字前添加前缀std了。如果想把在库中定义的所有名字都暴露在全局名字空间(global namespace)中，那么请在每个源文件的开始处都写上这句声明，紧跟在include指令后。同样，如果包含一个符合C标准的头文件(如<stdio.h>等)，在该头文件中所声明(或定义)的名字就会自动暴露在全局名字空间中(如果只想使用在名字空间std中声明的名字，可使用标准C++头文件<cstdio>)。

注意，宏名字并不遵循名字空间的嵌套规则。

对于这种综合声明的形式，人们有着不同的意见。一些人认为所有新的C++代码必须在库名字前面严谨地添加上std前缀。另外一些人认为这样做将会导致代码的杂乱无章，它将附带地使将现有的C++代码升级到符合标准形式的代价有所增加。我们所使用的测试代码使用了那种综合声明的形式，这样做的主要目的是为了简化将它们移植到那些不支持名字空间(或者支持的不是很好)的编译器时的转化工作。我们并没有因此就将这种容易导致众怒的编码风格作为一种标准来推行。

#### 1.3.2 方言

作为最后的警告，我们必须再一次强调，C++中的一些方言(dialect)依然存在。一些新近的语言特性（如名字空间，成员模板，模板的部分特化等）都是在C++标准逐步形成的过程中添加到该语言中的。在本书的写作过程中，并不是目前市面上可以得到的所有商业化的C++编译器都实现完全支持STL所需的所有语言特性。在我们认为恰当的地方，我们会指出如果没有实现语言中某种特定的特性的话，STL中的哪些特性将不会出现。

但是请不要让我们的保守影响你使用STL。目前编译器都在向着C++标准稳步靠拢，即便是在现在的情况下，你可以信赖的、跨步实现的STL子集也已经非常庞大了，其功能也非常强大。

## 1.4 实现STL

本书所提供的代码是作为随同C++编译器所提供的标准头文件的集合而出现的。在包含这些标准头文件的目录中，你很可能会发现一些头文件的名字与我们提供的名字十分相似，如deque或iterator等。然而，请注意：不要用本书提供的代码去替换标准头文件。STL本身在很大程度上是独立的，但它还是和标准C++库中的其他部分有着接口。如何实现这些接口在不同的实现中都存在它们自己的合法做法。我们将在以后的章节中讨论有关接口方面的内容，但我们再次处能做的就是提供一种可行的解决方案。但对于一个给定的编译器来说，该方案并不一定正确。

#### 1.4.1 编码风格

本书中代码的编码风格遵循下面的一些规则。其中许多规则也适用于其他项目，有一些则比较特殊：

* 每个秘密的(secret)宏或全局名字（即为我们的STL实现所私有的名字）都以大写字母开头，如Getint（为了教学上的可读性，我们在此省略了应该出现在库名字前面的下划线）。
* 代码的布局格式适当统一。在函数中对象的声明通常都应该出现在尽可能深的嵌套层中。嵌套的缩进形式应该忠实地反映出控制结构的嵌套形式。
* 在头文件中，所有的函数都被定义为内联函数，这样可以保证它们的独立性。没有用于库的被分拆的源代码存在。
* 为了避免代码的形式过度不规则，相对于传统的C++源文件中的函数定义来说，内联函数应该尽可能写的紧凑些（即使因此丧失了代码的部分可读性，有的头文件还是显得很大）。
* 代码中不包含有任何register声明。理由是我们很难做到聪明地放置它们，而且它们通常都会使代码变得杂乱无章。而且，现在的编译器一般都忽略程序员自己所做的register声明，自动进行寄存器的非陪。通常，编译器所做的寄存器分配工作比程序员做的要好的很多。
* 本书把每个C++源文件用程序清单的形式显式出来。每个程序清单的名称即该文件的文件名。

有时，这样做产生的代码看起来比较密集。在一个典型的项目中，程序员通常都会加上一些空格及注释，这样做会使程序增大至少30%（有的程序员可能会使程序变得更大，但我们在本书中这样做是为了从密集的文字中获得更多对代码的描述）。

这些代码同样也包含了一些头文件，不过它们被分割成几个小片段出现。在实际的实现中，这种分割的目的在于：减少在这些头文件中出现的循环引用。在今天高速计算机的运行条件下，编译时间已经只能算是一个小问题了。我们会在需要添加头文件的时候对此进行详细的解释。

#### 1.4.2 实现头文件

作为一个例子，程序清单1-1列出了文件utility，它实现了头文件<utility>。我们将在第3章中对这个小型的头文件进行更加详细的讨论。现在，我们主要关注头文件的总体结构。

**程序清单1-1**

//utility standard header

#ifndef UTILITY\_

#define UTILITY\_

#include <iosfwd>

namespace std{

    // TEMPLATE STRUCT pair

    template<class T1, class T2>

    struct pair{

        typedef T1 first\_type;

        typedef T2 second\_type;

        pair():first(T1()), second(T2())

        {

        }

        pair(const T1& V1, const T2& V2):first(V1), second(V2)

        {

        }

        template<class U1, class U2>

        pair(const pair<U1, U2>& X):first(X.first), second(X.second)

        {

        }

        T1 first;

        T2 second;

    };

    // pair TEMPLATE OPERATORS

    template<class T1, class T2>

    inline bool operator==(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return X.first == Y.first && X.second == Y.second;

    }

    template<class T1, class T2>

    inline bool operator!=(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return !(X==Y);

    }

    template<class T1, class T2>

    inline bool operator<(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return X.first<Y.first || !(Y.first<X.first) && X.second<Y.second;

    }

    template<class T1, class T2>

    inline bool operator>(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return Y<X;

    }

    template<class T1, class T2>

    inline bool operator<=(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return !(Y<X);

    }

    template<class T1, class T2>

    inline bool operator>=(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return !(X<Y);

    }

    template<class T1, class T2>

    inline pair<T1, T2> make\_pair(const T1& X, const T2& Y)

    {

        return pair<T1, T2>(X, Y);

    }

    // TEMPLATE OPERATORS

    namespace rel\_ops{

        template<class T>

        inline bool operator!=(const T& X, const T& Y)

        {

            return !(X==Y);

        }

        template<class T>

        inline bool operator>(const T& X, const T& Y)

        {

            return Y<X;

        }

        template<class T>

        inline bool operator<=(const T& X, const T& Y)

        {

            return !(Y<X);

        }

        template<class T>

        inline bool operator>=(const T& X, const T& Y)

        {

            return !(X<Y);

        }

    };

};/\* namespace std \*/

#endif /\* UTILITY \*/

首先，也是最明显的是，头文件被通常用于解释的注释及用于防范措施的保护宏(macro guard)所包围。

//utility standard header

#ifndef UTILITY\_

#define UTILITY\_

...

#endif /\* UTILITY \*/

许多C和C++程序员都把这当作一种常用的习惯用法，用以防止一个头文件被程序包含多次。typedef或者宏定义可以在一个翻译单元中出现多次，只要它们每次的定义结果都一样就可以了（这也称为良性定义，benign redefinition）。但实质上所有其他实体至多只能定义一次。在此处我们定义了宏\_UTILITY\_，把它作为一个开关来使用：在第一次使用（即被包含）时它就被设置（即被定义），然后我们就可以使用它来确保该头文件只被考虑一次，即使在一个翻译单元的include指令中多次包含它也是如此。这就可以防止那些讨厌的多次定义（这种情况经常出现在使用了大量include指令的用户代码中）。

宏名字\_UTILITY\_的选择一定要小心。C和C++标准中都保留了部分名字集给实现者使用。尤其是那些一下划线开头，后面紧接着一个大写字母的名字更是要严格控制其使用范围（只能由实现者使用）。也就是说，使用这个集合中名字的实现可能会干扰其他程序的正常翻译（同一个库中不同部分的实现更加需要注意，千万不要一不小心就彼此冲突）。

#### 1.4.3 包含

下一行就是include指令：

#include<iosfwd>

头文件<iosfwd>是标准化委员会的一个创举。它为在其他头文件中定义的部分模板及类提供了一些前向引用(forward reference，即对不完整类型的声明)。一般人很少会包含这个头文件，但对实现来说，它简直就是上天赐予的礼物。它可以帮助我们打破在标准C++库的不同组件中出现的循环依赖关系。

严格地说，在这个实现的头文件<utility>中，<iosfwd>根本就无用武之地。然而，它却能简化标准头文件中包含关系的结构。在这个特殊的实现中，其他包含<utility>的头文件都需要包含<iosfwd>。于是我们就可以在此处使用include指令来包含它，这样做将对以后产生很大的便利。附录A将概述STL头文件是如何与其他头文件（不但是和STL有关的头文件，也包括其他的标准C++库中的头文件）相互作用的。

我们本可以掩盖这种实现细节，把它作为一种不受欢迎且容易分散注意力的做法给忽略掉。然而，我们宁可强调在大量的软件中顺利地协同工作的困难程度。这个特殊的例子同样也强调了：STL并非存在于真空之中。虽说在很多方面它是一个独立的组件，但它更是一个大型的库的一部分。

#### 1.4.4 名字空间

在头文件<utility>的结构中，接下来的部分是名字空间声明：

namespace std{

    //...

}; // namespace std

这个特殊的声明确保在大括号中定义的所有名字（宏名字除外）都位于名字空间std内。在稍早的时候我们说过，名字空间的行为更象是一个类包装器，但这样说只能算是接近正确。与类包装器不同的是，名字空间的声明可以在同一个翻译单元中多次出现，并且代表同一个名字空间。名字相同的部分会有效地连接起来。

#### 1.4.6 产品代码

实际产品代码不可能象我们所展示的那样整洁有序。通常，为了在不同的实现之间得到很好的移植性，库中的代码充满了各种不同的条件编译指令。给定的实现可以打开它所能达到的条件开关，在代码中反复使用C++中的方言。对于不支持的特性来说，解决的方法有两种：一是禁用它们；二是提供一种简单的替代形式。你可以很容易地得到一个中文菜单，代价就是头文件的可读性受到影响。本书中的代码确实基于一个完整的STL产品。在此处我们以一种最通常的形式把这些代码展现给大家，目的是为了更好地利用这些代码来辅助我们的讲解，使大家能够很容易地写出具有可移植性的商业代码。但这本书的最主要的目的是讲解STL的复杂难懂之处。就算不再往其中添加进新的特性，这也已经够复杂了。于是，我们决定从该代码中抽取出部分代码，稍加修饰后当作实际的数学代码使用，这些代码在本质上能够完成同样的功能。这两者之间最主要的不同之处在于：用作教学目的的代码其可读性更高；这其实也是我们设计它们的初衷。

如我们在前面所提到的，在可读性方面我们也作了小小的让步。程序清单1-1中大量使用了传统的用于表示类型的名字(如T1和T2)以及表示对象的名字(如X和Y)。这种使用名字的做法在C++标准中经常(如果不是“总是”的话)出现。它们也确实遵循了C++中对事务(包括类型、对象等)命名的一种广泛采用的惯例(虽然实际上对象的名字不太可能以大写方式出现)。也就是说，头文件“内部”的名字的选择是为了让我们尽可能地理解它们。

唯一的问题在于，这样做并不完全遵循C++标准。并不是说使用它们存在着破坏用户代码的问题。参数的名字，以及嵌套在类及函数中的其他名字，并不会被外界的代码所认知，当然也就不会对外界环境产生所谓的名字污染(name pollution)。问题是以另一种表现形式而存在的。在各种不同的环境下，用户选择使用的名字可能会渗透到库声明和定义中，并且以一种危险的方式改变这些名字的原有意义。这种连锁反应通常很难诊断出来。有的时候，你可能会写出这样的程序：它们可以正常编译，但执行起来却与你所预期的结果有着天壤之别。

最容易导致这样的问题出现的情况是：定义一个有着普通名字(如T1或者T2)的宏。如果在第一次包含库的头文件前这样做了，就会导致重写任何使用该名字的声明。不管你是有意还是无意地试图破坏库，在库中都缺乏一种严肃的编程方法来避免收到这种攻击的破坏。

是的，我们知道，人们经常使用诸如T1和X这样的宏名字，一个有着严密思维的程序员不应该幻想并沉迷于这样的行为之中。许多程序员很快就会意识到，在把类型、宏及其他名字的命名方式格式化之后，维护它们的工作将会得到极大的减轻。但是C++标准仍然允许在程序中使用这样的名字来定义宏——它们通常是作为用户保留的name space的部分而使用的(name space，或名字集，虽说和C++中的namespace声明十分相似，但请不要把这两者混为一谈)。

实际上，使用这样普通的名字来声明类型经常会产生问题。在这里我们并不想给出任何示例，但这个问题在C以及C++中确实是一个尽人皆知的事实。在太多的环境中，外部包容环境中类型意义的改变将会对内部嵌套环境中的声明(甚至是某些表达式)的解释产生深刻的影响。

#### 1.4.5 保留名字

在产品库代码中，所有名字在被当作宏名字使用时都应该同样小心，应尽量让它保持它所代表的真正目的。这意味着我们应该这样写：

    template<class \_T1, class \_T2>

    inline bool operator==(const pair<\_T1, \_T2>& \_X, const pair<\_T1, \_T2>& \_Y)

    {

        return \_X.first == \_Y.first && \_X.second == \_Y.second;

    }

    template<class \_T1, class \_T2>

    inline bool operator<(const pair<\_T1, \_T2>& \_X, const pair<\_T1, \_T2>& \_Y)

    {

        return \_X.first < \_Y.first || !(\_Y.first < \_X.first) && \_X.second < \_Y.second;

    }

虽说与程序清单1-1中的代码相比，上面代码的可读性可能稍差了点，但它确实是节选自实际的产品代码。你可能会理解，为什么我们没有将许多这样的代码强加给那些原本可能会对学习STL更感兴趣的人们。

## 1.5 测试STL

测试本身是一个很严肃的主题(从Pla95中可以获得对此话题更广泛的讨论，而那些也不过只是表层的内容而已)。我们都知道，一个典型的库函数不可能被详尽地测试。输入值的组合通常都是一些天文数字，特别是对浮点参数更是如此。请想一下对于模板而言，可能出现的最糟糕的情况是什么(可为任何类型参数对模板进行特化)？

实际上，我们甚至不需要做任何尝试。作为替代方案，我们提供了一系列简单的测试程序。它们中的每一个都会对STL头文件所提供功能的部分或全部进行测试。你将会发现，这些测试程序主要关注于一些外部行为。这意味着，它们本质上就是一个简单验证套件。然而，它们偶尔也会闯入“测试内部结构”的领域中。某些实现中出现的错误有时是如此普遍，如此危险，以至于我们很难做到不去测试它们。另外，我们所给出的这些测试程序很少进行性能测试。

你最终将会发现，这些测试都非常粗浅和简单。然而，即使是简单的测试也能有效地解决实际中的问题。对于每个特性，我们都提供了一个基本且可行的例子，这可以使你能够切实地从错综复杂的STL之中获得真正的好处。可以仅凭少量的几行代码来证实一个模板是否达到了它的最初设计目的。这就在此保证了实现的正确性。当发生变化时(毫无疑问一定会发生)，重复这些测试就可以重新保证正确性。基于上述理由，我们相信：写出简单的测试代码以及保存它们是非常值得做的事情。

#### 1.5.1 简单的测试

我们发现：当执行成功时，那些简单且有把握的测试将打印出一条标准可靠的信息，并且在退出时返回一个表示执行正确的状态。如果有可能的话，测试代码就不再告诉我们其他的信息了。

我们通过在每个头文件的名字前面添加字母“t”来构造测试代码的文件名，如果有必要的话，还需要将它们截断，以使得文件名(不包括后缀“.c”)不超过8个字符。在以此规则得到的文件名稍微有点儿奇怪的同时，在各种操作环境下文件管理系统的麻烦会少很多。也就是说，tutility.c就是用来测试头文件<utility>的。它测试该头文件所定义的是否是它所支持的并在测试结束时显示一个可靠的测试信息：

SUCCESS testing <utility>

然后就正常地退出。如果愿意的话，可以通过一个普通的命令脚本来运行本测试或其他测试，并且简单地对它的退出状态进行检测。

注意，在每个这样的测试文件中都定义了其本身的main函数。把每个文件与标准C++库链接后，就可以得到独立的测试程序。不能把它们中的任何一个加入到你自己的标准C++对象-模块(object-module)库中。因为每个文件都是被设计成一个独立的程序来运行的。

#### 1.5 习题

习题1-1 在先前给出的那个例子模板中：

template<class T>

inline T Sum\_all(T \*first, T \*last)

{

    T sum;

    for(sum=0; first != last; ++first)

        sum += \*first;

    return sum;

}

请说出模板是否可以被下列的类型成功地特化：

* int类型
* float类型
* vodi类型
* 指针类型

习题1-2 对于上面给出的模板sum\_all，请列出对于能够成功特化该模板的类型参数T所需要支持的全部操作。

习题1-3 定义一个类X，使得当x是类X的一个对象时，调用sum\_all(x)将得到定义良好的结果。

习题1-4 sum\_all(&x, &x)的结果是什么？通过这种方式来得到x的值是否是一种很好的做法？请说出你的理由。

习题1-5 假设需要检测用类型float去特化该模板时sum\_all是否会按照所预期的方式工作》请列出你的测试代码。

习题1-6 假设我们把上题中的类型float换成任意的类型T，请列出你的测试代码。

习题1-7 如何更改sum\_all的定义，使得对于有着奇怪定义operator=(const T&)以及operator+=(const T&)的参数类型T，其模板的特化依然有着“明智”的行为。

# 第2章 迭代器

## 2.1 背景知识

迭代器在STL中起着粘合剂的作用，用来将STL的各个部分结合在一起。从本质上来说，STL提供的所有算法都是模板，我们可以通过使用自己指定的迭代器来对这些模板进行特化。我们也建议大家按照这种格式写出自己的算法模板。同样，在标准库中所提供的所有容器也都提供了这样的迭代器，用以存取它们所管理的数据序列。同样也建议大家在自己定义的容器中提供与之相配的迭代器。由于对象指针可以被当作任意种类的迭代器使用，所以这不会给你带来许多限制。

在此，我们主要以<utility>、<iterator>和<memory>这三个头文件为线索展开讨论。这三个头文件中的第一个能提供的信息远比迭代器技术要多。但是，对这三个头文件的使用一直贯穿着整个STL，它们也用来提供实现STL所必需的机制。在接下来的三章中，我们将分别对它们进行更详细的讲解。在此先提醒大家一下，对于这些细节方面的讨论有时十分单调乏味，而且在开始讨论算法和容器前，我们几乎很难察觉到这些细节的好处。从现在起，我们仅仅关心迭代器所共有的一些属性，像如何将它们分类以及不同种类的特点是什么等。

## 2.2 功能描述

C++中的迭代器相对于C中的对象指针来说更加一般化。指针本身就可以作为定义好的迭代器来使用。这种一般化行为主要体现在可以在C++中声明新类，然后对于这些类中的大部分操作符进行重载(overload)，赋予它们新的意义。甚至还可以让迭代器指向一个奇异值(singular value)，该迭代器的行为与一个定义有问题的指针十分相似。

#### 2.2.1 输出迭代器

可使用迭代器来存取有序序列中的元素。在此，“存取(access)”是一个一般化的术语，我们既用它来表示将值存储到对象中，也用它来表示从一个对象中获得它所保存的值。如果需要创建一个新的值序列，并且以有序的方式来产生值，可以写一个如下的循环语句：

for(; <not done>; ++next)

    \*next = <whatever>;

在此，next表示一个迭代器类型为X的对象，<not done>是一个用来检测循环是否应该终止的谓词(predicate)，而<whatever>则是一个表达式，类型为序列中元素的类型T，最起码也是那种很容易转化为T的类型。

具有这样的类型X，可以这样使用的迭代器，称为输出迭代器(output iterator)。从上面的讨论来看，一个输出迭代器至少还需要定义下面这些操作：

* \*next = <whatever>将<whatever>的值赋给序列中将要产生的下一个元素。
* ++next改变next的值，使之指向序列中的下一个元素。

在C中经常使用一种稍有不同的方法来完成同样的操作：

while(<not done>)

    \*next++ = <whatever>;

为了支持这种写法，表达式\*next++ = <whatever>必须将上面所描述的两种操作结合起来，使之具有和C中的指针一样的意义。

实际上，一个输出迭代器所能保证的一点也不比上面列出来的属性多(甚至不能判断一个输出迭代器是否前进得太多)。围绕着输出迭代器的一系列操作包括一个复制构造函数。与这些可怜的属性相对应的是，输出迭代器允许以各种各样的方式来实现它们。甚至可以将一个合适的输出迭代器装扮成每次存储一个输出记录的形式，使之输出到一个文件中。

表2-1以接近数学中常用符合的形式形象地描述了输出迭代器的各种属性。对于其他种类的迭代器，本书也将提供同样形式的表格。我们发现，使用这样的符号记录方式对于示例代码及上面的注释来说能起很大的作用，但也没有必要使用它来完全替换掉那些代码及注释。在某些对于输出迭代器来说非常重要的限制条件中，这样的表并不能完全清楚地表达它们。如：

* 必须保证输出迭代器在存储每一个元素后得以增加。
* 必须保证输出迭代器在两次存储的间隔内增加的次数不会超过一次。

**表2-1 输出迭代器的属性**

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 结果的类型 | 含义 | 注释 |
| X(a) | X | 产生a的一个拷贝 | 析构函数是可见的  \*X(a) = t  与\*a = t 的作用相同 |
| X u(a)  X u=a | X& | u是a的拷贝 |  |
| r = a | X& | 将a赋值给r | 结果\*r = t  和\*a = t作用相同 |
| \*a = t | void | 在序列中存储新元素 |  |
| ++t | X& | 指向下一个元素 | &r == &++r |
| r++ | 可以转换为const X& | {X tmp = r;  ++r;  return tmp;} |  |
| \*r++ = t | void |  |  |
| 注释：X是迭代器类型；a的类型为X&；T是元素类型；t的类型为T | | | |

如果输出迭代器实际上就是指针的话，上面的限制就不明显了。但当你看到STL使用一些具有特殊目的的输出迭代器的技巧时，就会知道对于这些限制的需求了。你可能会更欣赏表2-1中一些相对深奥的条目，如有关r++的返回值类型等。但从现在起，只需要记住，输出迭代器只是在类似上面给出的循环语句中使用。

#### 2.2.2 输入迭代器

输入迭代器(input iterator)用来产生新的序列。为了顺序存取已有的值，或只是需要对已有的序列进行遍历，可以这样做(和前面的方法略有不同)：

for(p=first; p!=last; ++p)

    <process>(\*p);

在这里，p、first和last都是迭代器类型X的对象，<process>是一个函数，它能够接受元素类型为T的参数。对于上面出现的由元素组成的序列，我们用一个半封闭期间[first, last]来表示，其中，first和last各代表一个迭代器。

注意，last其实并不表示序列中的任何一个元素。实际上，它表示的是“end-of=sequence”标记，如在实际的序列末端紧接着的第一个元素。空序列是满足first==last的序列。C标准中也经常使用这种方法来操作C中的数组——可以将指向数组末端后的第一个元素的地址存储在指针中，但不能得到这个指针所指向的元素的值。迭代器一般化了这种概念，使其更加完美。

具有这样的类型X，可以这样使用的迭代器称为输入迭代器(input iterator)。从上面的讨论来看，输入迭代器至少还需要定义下面这些操作：

* 当两个类型为X的迭代器p和q没有同时指向一个元素时，p!=q就为真（为方便起见，p==q通常都被定义成与p!=q在逻辑上相反）。
* \*p是类型T的一个右值(rvalue)。表达式p==last没有定义。
* ++p将改变p的值，将它指向序列中目前所指向元素的下一个元素。表达式p==last没有定义。

在C中经常使用一种稍有不同的方法来完成同样的操作：

while(first!=last)

    <process>(\*first++);

为了支持这种写法，表达式\*first++必须将上面所描述的两种操作结合起来，使之具有和C中的这种一样的意义。

从这些属性我们可以得知，只有当可以从first到达last时，[first, last]才时一个有限的序列。换另一种说法就是，first的值在经过有限次的增加后必定会得到一个确定的值，它与last的值相等的。

与其兄弟输出迭代器一样，输入迭代器所保证的也不比上面的这些属性多。同样围绕着输入迭代器的一系列操作包括一个复制构造函数、一个析构函数和一个赋值操作符，它们都带有通常所需的一些属性。这其中唯一新增的就是指针指向操作符p->m，只有在类型T是一个结构化类型时，它才被定义。同样，输入迭代器也允许有各种各样的实现。

你甚至可以将一个合适的输入迭代器装扮成每次存取一个输入记录的形式，从一个文件中获得所需要的输入。为了能使用这种技巧，输入迭代器定义了一个end-of-sequence值，它在大多数情况下是一种end-of-file标记。这个end-of-sequence值存储在last中，用来指向一个真正记录的迭代器first不会等于last。然而，当first指向的记录是序列中的最后一个元素时，再增加first的值将导致其值发生改变，变为与last相等。也就是说，我们可以以同样的控制方式来完成这两件事情：通过增加指针的值来遍历一个数组与通过增加输入迭代器的值来遍历一个文件。

表2-2以接近数学中常用的符号形式形象地描述了输入迭代器的各种属性。我们在此重申一遍，这样的表并不能将所有有关输入迭代器的重要限制都讲述清楚。例如，它没有说，只有当两个迭代器中至少具有关联end-of-sequence值时，它们之间的比较才一定有意义(当使用任意类型的输入迭代器时，对于序列中的两个明显不同的地方，你甚至不能很明确地说出序列是否被越界存取)。建议在处理输入迭代器时遵循我们在输出迭代器处所提出的建议——只在类似上面给出的循环语句中使用它们。

**表2-2 输出迭代器的属性**

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 结果的类型 | 含义 | 注释 |
| X(a) | X | 产生a的一个拷贝 | 析构函数是可见的  \*X(a)与\*a 作用相同 |
| X u(a)  X u=a | X& | u是a的拷贝 | 创建完成后  U==a |
| r = a | X& | 将a赋值给r | 结果r == t |
| a == b | 可以转换为bool | 相等比较 | a和b在同一值域内 |
| a != b | 可以转换为bool | !(a == b) |  |
| \*a | T | 从序列中存取元素 | 不是end-of-sequence |
| a->m | m的类型 | (\*a).m | T有成员m |
| ++r | X& | 指向下一个元素 | &r == &++r，r不是end-of-sequence，r的拷贝无效 |
| (void)r++ | void | (void)++r |  |
| \*r++ | T | {T tmp = \*r;  ++r;  return tmp;} |  |
| 注释：X是迭代器类型；a和b的类型为X&；r的类型为X&；T是元素类型；t的类型为T | | | |

#### 2.2.3 前向迭代器

从我们先前列举的一些理由来看，输出迭代器和输入迭代器可以说“足智多谋”。它们可以用来处理任意长度的文件。然而，迭代器的有关更加普遍的用法是：存取有关完全存储在内存中的序列。在这种情况下，所需要使用的迭代器就必须具有较少让人觉得惊奇的属性。

举例来说，你仍然只是满足于从头至尾地存取一个序列中的所有元素。但这次你想要对于该序列中的元素同时具有读和写的权利，或者想在先前所存取的任意地方标注出一个位置作为“书签“以方便下次存取。在这种情况下，仍然使用我们在讲解输出迭代器和输入迭代器时所用的那个控制循环，你所需要的很简单，那就是让迭代器看起来更像是一个传统的指针。

运用前向迭代器(forward iterator)就可以达到所有的这些要求。和输入迭代器一样，可以比较两个前向迭代器是否相等，但现在它们可以都为（或都不为）end-of-sequence。当然，和先前所讨论的一样，这两个迭代器的值必须处于同一个值域中。和输入迭代器一样，前向迭代器也可以有end-of-sequence值，它的意义和数组中“off the end”元素的地址差不多，你也可以将它想象成其他任意的end-of-sequence标记。你还可以用一个前向迭代器的多个有效拷贝来指向当前序列中的任意位置。

可以把前向迭代器想象成为一个指向单向链表中元素的指针。你可以明确地指出它是否指向链表的末端(用null来标记end-of-sequence)。如果它指向的元素不是end-of-sequence，就可以通过它来存取该链表的元素，或是把它移到序列中下一个元素的位置。但是不能让它回退，也不能通过它来直接存取链表中的任意元素。

严格地讲，你“可以”通过使用前向迭代器来完成这些操作中的一部分。但这具有一定的欺骗性，不管链表的长度如何，你都不可能在恒定的时间内完成这些操作。对于所有的迭代器来说，都存在着一个隐式要求：我们对于迭代器的所有存在都不能有太大的开销。不能随着迭代器所指向序列的长度改变而改变存在所需花费的时间。

表2-3以接近数学中常用符号的形式形象地描述了前向迭代器的各种属性。

**表2-3 前向迭代器的属性**

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 结果的类型 | 含义 | 注释 |
| X() | X | 产生一个默认值 | 析构函数是可见的，值可以是end-of-sequence |
| X u  X u=a | X& | u具有默认值 |  |
| X(a) | X | 产生a的一个拷贝 | 析构函数是可见的，\*x(a)和\*a的作用相同 |
| X u(a)  X u=a | X& | u是a的一个拷贝 | 创建完成后，u==a |
| r=a | X& | 将a赋给r | 结果r==a |
| a==b | 可转换为bool | 相等比较 | a和b在同一值域中 |
| a!=b | 可转换为bool | !(a==b) |  |
| \*a | T& | 从序列中存取元素 | a不是end-of-sequence，a==b意味着\*a==\*b |
| \*a=t | T& | 在元素中存储 | a不是end-of-sequence，x是可变的 |
| a->m | m的类型 | (\*a).m | T有成员m |
| ++r | X& | 指向下一元素 | &r == &++r，r不是end-of-sequence，r==s意味着++r==++s |
| r++ | 可转换为常量x& | {x tmp = r;  ++r;  return tmp;} |  |
| \*r++ | T& | {T tmp = \*r;  ++r;  return tmp;} |  |
| 注释：X是迭代器类型；a和b的类型为X&；r和s的类型为X&；T是元素类型；t的类型为T | | | |

#### 2.2.4 双向迭代器

比前向迭代器更为广泛的一种迭代器同时支持递增及递减操作。通过使用这种迭代器的这些特性，许多算法都得以以更加高效的方式实现。STL所定义的双向迭代器(bidirectioanl iterator)就是这样的一种迭代器，与前向迭代器相比，它多了可以在序列中逆向移动这种特性。

我们可以将双向迭代器想象成指向一个双向链表中的元素的指针。我们可以明确地指出它是否指向该链表的末端(用null来标记end-of-sequence)。如果迭代器指向的不是链表的末端，我们就可以通过它来存取该链表中的元素，或是将它移到序列中下一个元素的位置。如果迭代器指向的不是链表中的第一个元素，我们就可以把它回退到序列中前一个元素的位置。但是我们不可能通过它直接存取链表中任意位置的元素，至少我们不可能通过恒定时间的操作来达到这个目的。

表2-4以类似于常用数学符号的形式描述了双向迭代器的各种附加属性。所有适用于前向迭代器的属性，也同样适用于双向迭代器。

**表2-4 双向迭代器的附加属性**

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 结果的类型 | 含义 | 注释 |
| --r | X& | 指向前一个元素 | 对于一些s，++s=r，&r==&--r，r不是end-of-sequence，r==s意味着--r==--s |
| r-- | 可转换为常量x& | {x tmp = r;  --r;  return tmp;} |  |
| \*r-- | T& | {T tmp = \*r;  --r;  return tmp;} |  |
| 注释：X是迭代器类型；r和s的类型为X&；T是元素类型；所有其他属性和前向迭代器相同 | | | |

#### 2.2.5 随机存取迭代器

我们所讨论的最后一种迭代器具有C语言中对象指针的所有强大功能。除了在双向迭代器中所提到的那些特性外，随机存取迭代器还支持与整型值的加减操作、指针之间的相减、两个迭代器在序列中的顺序比较，以及使用下标方式操作该迭代器等。某些算法只能依靠这种程度的弹性才有可能运作良好（通过二分法来对序列进行排序和快速查找就是这样的两个例子）。

然而，请记住，随机存取迭代器仍然不是C语言风格的指针。例如，它们之间存在的一个区别是某种类型Dist，它可以是也可以不是一种最基本的整数类型。我们可以将作用于整型值的算法应用于Dist对象，但这并不能阻止我们将Dist定义为一个类。

表2-5以类似于常用数学符号的形式描述了随机存取迭代器的各种附加属性（相对于双向迭代器来说）。所有适用于双向迭代器的属性同样也适用于随机存取迭代器。

**表2-5 随机存取迭代器的附加属性**

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 结果的类型 | 含义 | 注释 |
| a<b | 可转换为bool的任意类型 | 从a可以到达b | a和b处于同一个值域中 |
| a>b | 可转换为bool的任意类型 | b<a |  |
| a<=b | 可转换为bool的任意类型 | !(b<a) |  |
| a>=b | 可转换为bool的任意类型 | !(a<b) |  |
| r+=n | X& | {Dist m=n;  while(0<m)  --m, ++r;  while(m<0)  ++m, --r;  return r;} |  |
| a+n  n+a | X | {X tmp=a;  tmp += n;  return tmp;} | a和b处于同一个值域中 |
| r -= n | X& | r += -n |  |
| a-n | X | a + (-n) |  |
| b-a | Dist | {Dist m=0;  while(a<b)  ++a, ++m;  while(b<a)  ++b, --m;  return m;} |  |
| a[n] | 可转换为T的任意类型 | \*(a+n) |  |
| 注释：X是迭代器类型；a和b的类型为X；r和s的类型为X&；T是序列元素中的类型；Dist是序列X的差距类型；其他属性和双向迭代器中讨论的一样 | | | |

## 2.3 使用迭代器

STL中大量使用了迭代器，它们用于不同的算法和算法作用的序列之间，起着桥梁的作用。为了本书中其他章节的简洁期间，我们使用迭代器类型的名字来代指迭代器的种类。为了提升其功能，我们把不同的迭代器总结为几下几类：

* OutIt——假设X为一个输出迭代器，那么它只能通过存储来间接地拥有一个值V。我们在输出迭代器中存储了一个值后，必须将其递增。如：(\*X++=V)、(\*X=V, ++X)，或者是(\*X=V, X++)。
* InIt——假设X为一个输入迭代器，那么它的值也可以为end-of-sequence。如果它不等于end-of-sequence的话，我们就可以通过间接的方式来存取它所拥有的值V，如：(V=\*X)。如果想要取得序列中的下一个元素的值(或是end-of-sequence)，必须将其递增，如：++X、X++，或者是(V=\*X++)。一旦我们对一个输入迭代器进行了递增，它的所有其他拷贝就不保证一定能够完成下面的操作：比较、间接取值，或者是再对其进行递增等。
* FwdIt——假设X是一个前向迭代器，如果\*X是可变的，那么它就可以用来代替输出迭代器（因为这时它本身就是一个输出迭代器）。同样，它也可以用来替换一个输入迭代器。然而，也可以通过一个前向迭代器来读取它所拥有的值（通过使用V=\*X），而这个值就是你刚刚存储到它之中的那个值（通过使用\*X=V）。你可以同时拥有一个前向迭代器的多份拷贝，它们中的每一份都可以用来间接取值，或是各自进行递增。
* BidIt——假设X是一个双向迭代器，那么我们可以用它来替换一个前向迭代器。并且，你还可以对双向迭代器进行递减，如：--X、X--，或者是(V=\*X--)。
* RanIt——假设X和Y都是随机存取迭代器，那么我们就可以用它们来替换一个双向迭代器（因为它们本来就是一个双向迭代器）。我们同样也可以对随机存取迭代器进行许多整数运算（这一点和对象指针一样）。如果N为一个整型对象，那么我们就可以这样写：X[N]、X<Y、X-N、N+X等。

注意，对象指针可以用来替换一个随机存取迭代器（或者说它本来就是随机存取迭代器）。

#### 2.3.1 迭代器的种类

迭代器分类的层次可以总结为下面的三种情况。如果只需对序列进行只写(write-onle)操作，我们可以选择以下的任意一种迭代器：

输出迭代器

->前向迭代器

->双向迭代器

->随机存取迭代器

右向箭头(->)意味着“可以被...替换”。例如，对于那些需要使用输出迭代器的算法，如果我们传给它们一个前向迭代器，它们的执行不应该有任何异常。但反之则不成立。

如果只需对序列进行只读(read-only)操作，我们可以选择以下的任意一种迭代器：

输入迭代器

->前向迭代器

->双向迭代器

->随机存取迭代器

在这种情况下，输入迭代器是这些种类中功能最少的一种。

最后，如果需要对序列进行读写(read/write)操作，我们可以选择以下的任意一种迭代器：

前向迭代器

->双向迭代器

->随机存取迭代器

记住，对象指针总是可以当作随机存取迭代器来使用。因此，只要它支持对指定的序列进行适当的读写操作，它也就可以当作任意种类的迭代器来使用。

迭代器的这种“代数学上”的应用几乎是STL中其他部分的基础。清楚地了解每一种迭代器的适用范围及限制条件，对于我们了解迭代器在STL容器及算法中的应用极为重要。

## 2.4 习题

习题2-1 写出下面操作所需的功能最少的迭代器种类：

* 提供无限个0
* 向文件中写入一个值序列
* 实现一个栈（后进先出队列）

习题2-2 下面列出的几种迭代器中，哪一种是可以替换的：

* 输出迭代器
* 只读前向迭代器
* 随机存取迭代器

习题2-3 迭代器同样也可以基于Fortran语言格式的Do循环：

for(p=first; p<=last; ++p)

<process>(\*p);

试比较这种格式与STL中所选择的那种格式（见本章“输入迭代器”一节）

习题2-4 解释为什么在所写的算法中使用其他种类的迭代器，而不是随机存取迭代器？

习题2-5 解释为什么宁愿定义一个仅能通过迭代器来存取的数据结构，而不是让它可以被随机存取呢？

习题2-6 写出这样一个模板类bidir<FwdIt>，当我们用一个前向迭代器类型来特化FwdIt时，它的表现就和双向迭代器一样。你会采用何种方法来使它和预期中的双向迭代器行为一致？

习题2-7 写出这样的一个模板类ran\_read<InIt>，当我们用一个输入迭代器类型来特化InIt时，它的表现就和一个只读的随机存取迭代器一样。

# 第3章 <utility>

## 3.1 背景知识

<utility>是一个很小的头文件。它包括了贯穿使用在STL中的几个模板的声明。与最初指定的一样，它看上去还是比较大（虽然与其他的头文件相比还是比较小）。然而，<utility>中的两个成员后来被消除掉了。在惠普的实现中，类empty与类restrictor的设计目的是避免一些在模板定义的翻译初始阶段可能会出现的问题。现在，通过改进了翻译技术，这样的类也就不再需要了。

现在，在<utility>中剩下来的就只有模板类pair、一些与之相关联的模板函数和操作符，以及其他四个模板操作符了。该模板类用来将两个对象表示成一个对象——当你想要一个函数返回两个值，或者想用一个容器来存储具有成对值的元素时，这样做就比较方便。那四个模板操作符赋予了==操作符以及<操作符新的内涵，它要求与之相应的两个操作值具有相同的类型，通过一致性的方式来定义剩余的相关操作符。

#### 3.1.1 pair

我们可以构造一个带有明确初始值的模板类pair<T, U>的对象，也可以让默认构造函数提供默认初始值来完成这个工作。在此处我们并没有卖弄什么花哨的技巧——如果需要取得x（为一个pair对象）的第一个成员对象（其类型为T）的话，没有可以这样写：x.first；如果是第二个的话（其类型为U），则使用：x.second。

使用成员模板构造函数，也可以用另一个模板类pair<V, W>的对象来构造模板类pair<T, U>的对象。相应的成员提供初始值。

#### 3.1.2 make\_pair

模板函数make\_pair是一个十分方便的工具。通过使用它们，我们就可以在必要时实时产生出一个pair对象。然而不幸的是，并不是每一次都能成功。模板函数在检测模板参数时将忽略掉所有的const属性（至少它们可以这样做——某些现在常见的编译器就时这样处理的）。也就是说，我们不能依赖make\_pair来产生一个含有一个或多个常量成员对象的pair对象。而这些正时容器map和multimap有时所需要的（参见第14章）。不过，这个模板函数也有它自己的用途。

#### 3.1.3 operator==和operator<

有时，比较两个pair对象x和y还是有意义的。由两个模板操作符来完成这些必要的比较操作。在这两种比较操作中，比较容易证明的是“相等(equality)”。如果x和y对应的成员对象都相等的话，我们就可以说x和y是相等的。但我们怎样才能说两个pair对象谁“小于”谁呢？

我们可以这样假设：在pair的两个成员对象中，第一个成员对象的权重(weight)总是大于第二个的。在这种情况下，如果我们可以确认x.first<y.first的话，那x就小于y。还可以继续地加上这样的判断条件：x.first==x.second && x.second<y.second。然而，令我们好奇的是，该操作符的定义实际上并不完全是这样。它完全是由operator<来决定的，就如在它的成员对象中所定义的一样：

    template<class T, class U>

    inline bool operator<(const pair<T, U>& X, const pair<T, U>& Y)

    {

        return X.first < Y.first || !(Y.first < X.first) && X.second < Y.second;

    }

考虑一会儿后，我们可能相信：实际中使用的这种形式与我们上面所谈论的形式在逻辑上是等价的(equivalent)——至少对于该成员对象的类型来说，如果比较操作符的定义和我们直觉上认为的情况相同，那么上述结论就是成立的。

对于这种特有的形式，有一种理由是：简单即优雅。如果对于每个参数类型来说，需要定义的操作符是一个而不是两个，那么这个模板的定义就很不错了。而另一个理由就比较微妙了。就一个operator<来说，比之那些“一堆互相作用的比较操作符”，其由定义顺序导致的依赖关系要更健壮、更具有威力。我们将连同其他一些能够改变序列的顺序的算法来详细地说明这其中的原因（参见第5章）。我们现在所说的是，STL在类似这样的一些细节方面考虑得非常周到。这也正是它的长处之一。

#### 3.1.4 operator!=

现在我们可能会猜测：为什么在头文件<utility>中还需要存在着剩下的那四个模板呢？对于那些定义了operator==(const T&, const T&)的类型T来说，第一个模板函数为其提供了一个合理的operator!=定义，STL代码中大量地组合使用了这两个操作符，如：在循环控制中用它们来比较序列中的迭代器。在这个模板中，我们所需做的只是为类定义一个operator==，剩下的事情库会替我们完成的。

#### 3.1.5 operator>、operator<=和operator>=

其他三个模板函数以相同的方式扩展了operator<的使用，为其他三种关系操作(operator>、operator<=和operator>=)提供了合适的定义。然而，在此处它们的结果却有些争议。这些定义在类型为T的对象上事实完全排序(total ordering)。在多数情况下，程序员所需要的就是这种效果。但并不是在所有的情况下都如此。

有些类可能只定义了部分排序(partial ordering)。部分排序的一个经典例子就是由依赖关系所形成的树(dependency tree)。其他的类可能会选择和常见的排序规则没多大关系的方式来定义这些操作符。在这种情况下，这些模板合适的存在就完全是一件麻烦事了。它们将会导致分析过程中产生歧义或程序产生令人惊讶的代码、或支持那些类设计者认为是惹人嫌的标记法。

这些模板函数都被定义在名字空间rel\_ops中（嵌套定义在名字空间std中）。它们将不会参加在程序中进行的重载解析（除非是显式地要求它们这样做）。我们将在下面就这个问题继续讨论。

## 3.2 功能描述

namespace std{

    // TEMPLATE STRUCT pair

    template<class T, class U>

    struct pair;

    // TEMPLATE FUNCTIONS

    template<class T, class U>

    pair<T, U> make\_pair(const T& X, const U& Y);

    template<class T, class U>

    bool operator==(const pair<T, U>& X, const pair<T, U>& Y);

    template<class T, class U>

    bool operator!=(const pair<T, U>& X, const pair<T, U>& Y);

    template<class T, class U>

    bool operator<(const pair<T, U>& X, const pair<T, U>& Y);

    template<class T, class U>

    bool operator>(const pair<T, U>& X, const pair<T, U>& Y);

    template<class T, class U>

    bool operator<=(const pair<T, U>& X, const pair<T, U>& Y);

    template<class T, class U>

    bool operator>=(const pair<T, U>& X, const pair<T, U>& Y);

namespace rel\_pos{

template<class T>

bool operator!=(const T& X, const T& Y);

template<class T>

bool operator<=(const T& X, const T& Y);

template<class T>

bool operator>(const T& X, const T& Y);

template<class T>

bool operator>=(const T& X, const T& Y);

};

};

包含STL的标准头文件<utility>就可以得到在STL中普遍使用的几个模板的定义。

在给出了operator==和operator<的定义的情况下，四个模板操作符(operator!=、operator<=、operator>以及operator>=)定义了相同类型的成对操作数上的完全排序。

如果实现支持名字空间，那么这些模板操作符就定义在名字空间rel\_ops中（嵌套定义于名字空间std中）。要想使用这些模板操作符，就应该在程序中写处如下的声明：

using namespace std::rel\_pos;

它会把这些操作符暴露在当前的名字空间中。

template<class T, class U>

    pair<T, U> make\_pair(const T& X, const U& Y);

该函数返回pair<T, U>(x, y)。

template<class T>

bool operator!=(const T& X, const T& Y);

template<class T, class U>

    bool operator!=(const pair<T, U>& X, const pair<T, U>& Y);

该函数返回!(x==y)。

template<class T, class U>

    bool operator==(const pair<T, U>& X, const pair<T, U>& Y);

该模板函数返回x.first == y.first && x.second == y.second。

template<class T, class U>

bool operator<(const pair<T, U>& X, const pair<T, U>& Y);

该模板函数返回x.first < y.first || !(y.first < x.first) && x.second < y.second。

template<class T>

bool operator<=(const T& X, const T& Y);

template<class T, class U>

bool operator<=(const pair<T, U>& X, const pair<T, U>& Y);

该模板函数返回!(y<x)。

template<class T>

bool operator>(const T& X, const T& Y);

template<class T, class U>

bool operator>(const pair<T, U>& X, const pair<T, U>& Y);

该模板函数返回y<x。

template<class T>

bool operator>=(const T& X, const T& Y);

template<class T, class U>

bool operator>=(const pair<T, U>& X, const pair<T, U>& Y);

该模板函数返回!(x<y)。

template<class T, class U>

    struct pair{

        typedef T first\_type;

        typedef U second\_type;

T first;

U second;

        pair();

        pair(const T& X, const U& Y);

        template<class V, class W>

        pair(const pair<V, W>& pr);

    };

上述模板类中存储了一对对象，first的类型为T，second的类型为U。类型定义first\_type的含义和模板参数T一样，而second\_type则相当于模板参数U。

第一个（默认）构造函数将first初始化为T()，second初始化为U()。第二个构造函数将first初始化为x，second初始化为y。第三个（模板）构造函数将first初始化为pr.first，second初始化为pr.second。T和U只需各自提供应该默认构造函数，应该接受单参数的构造函数，以及一个析构函数。

## 3.3 使用<utility>

如果我们在翻译单元中包含其他STL头文件，那么<utility>就很可能会一同包含入该翻译单元中。只有在程序中需要使用该头文件中定义的STL组件并且不指望通过其他STL头文件将这些组件的定义带入翻译单元时，我们才有必要在程序中显式地包含它。但是在程序中需要使用该头文件中所定义的实体的可能性同样也很大。如果在翻译过程中我们得到了这样的诊断信息：pair定义，请记住：最简单的解决方案就是将这个头文件包含到程序中。

简单地来讲，对于返回一对相关值的函数调用来说，模板类pair是一个相当不错的工具。如果这两个类型都不是常量类型，我们就可以通过调用make\_pair(x, y)，来实时地产生一个pair对象用以存储x和y的值。否则，我们只能显式调用构造函数：pair<T, U>(x, y)。

如我们在情况所提及的一样，模板类pair的定义中包含一个成员模板的构造函数：

template<class U1, class U2>

pair(const pair<U1, U2>& X):first(X.first), second(X.second){ }

这个构造函数解决了使用make\_pair时存在的一些问题。只要两个pair之间相应的成员对象的类型可以赋值的话，我们就可以将make\_pair(x, y)所返回的结果赋给pair<T, U>的一个对象。这个函数并没有要求这两个pair的类型完全一致。不过虽然它很方便，但是并不是每一个现有的编译器都支持这样的构造函数。为了得到最大的可移植性，我们最好还是使用那种看起来不是那么优美的构造函数来得到同样的结果：pair<T, U>(x, y)。

如果你在自己定义的类T的两个对象间定义了相等性比较(operator==(const T&))，那么请在代码中添加我们在本章早些时候所描述过的using指令，这样就可以很自然地获得对它们之间不等性判断的定义(operator!=(const T&))。如果想为类T定义一个完全排序关系的话，我们只需要定义用于判断“小于”的比较操作(operator<(const T&))。

然而，在实际生活中，这四个模板操作符的作用并不明显，重载操作符的检索规则已经和名字空间的引入完全结合起来了。并不是所有的编译器都能对此做出完全正确的反应，有时可能会得不到你所希望的结果。我们建议在代码中最好不要依赖名字空间std::rel\_ops中的那些定义。最好在代码所使用的名字空间中将它们替换成你自己的定义。

## 3.4 实现<utility>

程序清单3-1列出了文件utility，它实现了头文件<utility>。我们已经在本章的前面部分讨论了其中存在着的一些特性。除此之外，其他的也就没什么了。

//utility standard header

#ifndef UTILITY\_

#define UTILITY\_

#include <iosfwd>

namespace std{

    // TEMPLATE STRUCT pair

    template<class T1, class T2>

    struct pair{

        typedef T1 first\_type;

        typedef T2 second\_type;

        pair():first(T1()), second(T2())

        {

        }

        pair(const T1& V1, const T2& V2):first(V1), second(V2)

        {

        }

        template<class U1, class U2>

        pair(const pair<U1, U2>& X):first(X.first), second(X.second)

        {

        }

        T1 first;

        T2 second;

    };

    // pair TEMPLATE OPERATORS

    template<class T1, class T2>

    inline bool operator==(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return X.first == Y.first && X.second == Y.second;

    }

    template<class T1, class T2>

    inline bool operator!=(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return !(X==Y);

    }

    template<class T1, class T2>

    inline bool operator<(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return X.first<Y.first || !(Y.first<X.first) && X.second<Y.second;

    }

    template<class T1, class T2>

    inline bool operator>(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return Y<X;

    }

    template<class T1, class T2>

    inline bool operator<=(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return !(Y<X);

    }

    template<class T1, class T2>

    inline bool operator>=(const pair<T1, T2>& X, const pair<T1, T2>& Y)

    {

        return !(X<Y);

    }

    template<class T1, class T2>

    inline pair<T1, T2> make\_pair(const T1& X, const T2& Y)

    {

        return pair<T1, T2>(X, Y);

    }

    // TEMPLATE OPERATORS

    namespace rel\_ops{

        template<class T>

        inline bool operator!=(const T& X, const T& Y)

        {

            return !(X==Y);

        }

        template<class T>

        inline bool operator>(const T& X, const T& Y)

        {

            return Y<X;

        }

        template<class T>

        inline bool operator<=(const T& X, const T& Y)

        {

            return !(Y<X);

        }

        template<class T>

        inline bool operator>=(const T& X, const T& Y)

        {

            return !(X<Y);

        }

    };

};// namespace std

#endif // UTILITY

## 3.5 测试<utility>

程序清单3-2列出了文件tutility.c。由于它所测试的头文件本身很小且没有什么号测试的，它也就比较简单。它主要检测那些明显存在着的定义。如果一切正常的话，它将打印出：

SUCCESS testing <utility>

然后就正常退出。

//test <utility>

#include<assert.h>

#include<iostream>

#include<utility>

using namespace std;

typedef pair<int, char> Pair\_ic;

Pair\_ic p0;

class Int{

public:

    Int(int v):val(v)

    {

    }

    bool operator==(Int x) const

    {

        return val == x.val;

    }

    bool operator<(Int x) const

    {

        return val < x.val;

    }

private:

    int val;

};

int main()

{

    Pair\_ic p1 = p0, p2(3, 'a');

    // TEST pair

    assert(p1.first == 0);

    assert(p1.second == 0);

    assert(p2.first == 3);

    assert(p2.second == 'a');

    assert(p2 == make\_pair((Pair\_ic::first\_type)3, (Pair\_ic::second\_type)'a'));

    assert(p2 < make\_pair((Pair\_ic::first\_type)4, (Pair\_ic::second\_type)'a'));

    assert(p2 < make\_pair((Pair\_ic::first\_type)3, (Pair\_ic::second\_type)'b'));

    assert(p1 != p2);

    assert(p2 > p1);

    assert(p2 <= p2);

    assert(p2 >= p2);

    // TEST rel\_ops

    using namespace std::rel\_ops;

    Int a(2), b(3);

    assert(a == a);

    assert(a < b);

    assert(a != b);

    assert(b > a);

    assert(a <= b);

    assert(b >= a);

    cout<<"SUCCESS testing <utility>"<<endl;

    return 0;

}

## 3.6 习题

习题3-1 写出模板类trio<class T, class U, class V>。要求：以模板类pair为原型，并且能够存储任意类型的三个对象。

习题3-2 在上题中，如果要求只使用模板类pair，模板类trio又应该怎么写才能达到同样的效果？

习题3-3 当T和U中有一个是const类型时，我们可能需要经常构造一个pair<T, U>对象。下面的定义是否可以达到这样的效果？如果不可以，请说出你的理由。

template<class T, class U>

inline pair<T, U> make\_pair(T& X, U& Y)

    {

        return pair<T, U>(X, Y);

    }

习题3-4 请描述出一种可能的编码情况，使得表达式x.first<y.first || !(y.first<x.first) && x.second < y.second并不等同于x.first<y.first || x.first==y.first && x.second < y.second。

习题3-5 怎样使用遍布于STL中的模板函数operator<=(const T& x, ocnst T& y)，才能保证不会影响到定义在库外面的类？

# 第4章 <iterator>

## 4.1 背景知识

正如其名字所暗示的那样，头文件<iterator>中提供了迭代器使用的许多方法。在这个头文件中没有很复杂的东西，但却有很多需要详细解释的细节。除此之外，迭代器还被划分为五种不同的风格（或种类），这其中还不包括对象指针（它们可以当作不同种类的迭代器来使用）。我们将会发现，在本章中所提供的大部分内容都将可能拥有将近半打的不同的版本。

由于这个头文件实在是太大了，所以我们决定把它分成不同的片段来展示给大家。不管怎样，大部分的片段所满足的需求都不尽相同。

#### 4.1.1 迭代器属性

首先需要解决的问题是，对所有种类的迭代器施加某种规则。我们在写算法时，常常需要知道给定的迭代器X的某些属性。被问的最多的问题有：

* X到底属于哪种迭代器（是输出的，输入的，前向的，还是其他）？
* \*X指定的元素类型（通常写做T）到底是什么？
* N到底应该为哪种差距类型（通常写做Dist），才可以在“重复地递增X”时用来计算所有元素的个数？

记住，整个STL几乎都是由模板组成的。模板所能够知道的就是被用来特化它的那些类型（再加上一些由此派生出来的类型）。如果想传送更多的信息，摆在你面前的有两种选择：一是在模板中新增模板参数来夹带所需的信息；另外一种就是让迭代器本身就包含有这些额外信息。

很明显，如果能够实现的话，第二种方法更为优雅。事实证明，大部分情况下都可以做到这一点。简单点的做法就是：让你所定义的所有迭代器类都派生于同一个基类，然后在这个基类中定义所有必要的属性。这个基类的一个原型就是模板类iterator：

template<class C, class T, class Dist=ptrdiff\_t, class Pt=T\*, class Rt=T&>

struct iterator{

    typedef C iterator\_category;

    typedef T value\_type;

    typedef Dist difference\_type;

    typedef Pt pointer;

    typedef Rt reference;

};

这个模板类中惟一有点特别的东西就是它定义的成员类型的名字。在另一个使用其他方式定义这些成员类型的迭代器中，我们就不需要它们了。在本章接下来的部分中，我们将详细讨论这五个成员类型的使用。

#### 4.1.2 迭代器标签

第一个模板参数定义了迭代器的种类，它必须是下面已经定义好的几种迭代器标签(iterator tag)之一：

struct output\_iterator\_tag{};

struct input\_iterator\_tag{};

struct forward\_iterator\_tag:public input\_iterator\_tag{};

struct bidirectional\_iterator\_tag:public forward\_iterator\_tag{};

struct random\_access\_iterator\_tag:public bidirectional\_iterator\_tag{};

上面列出的这些简单的空结构体只是用来表达唯一的类型，没有需要构造（或析构）的值。这个继承层次和第1章的“迭代器的种类”一节展示的具有只读属性的迭代器的层次完全一样。当我们在不同的迭代器种类之间重载函数时，这种方法十分方便。我们将在本章的“实现<iterator>”一节中详细讨论这个问题。

例如，如果在这个基本结构上构造一个前向迭代器：

class My\_it:public iterator<forward\_iterator\_tag, char, long>

{

......

}

那么，算法就可以从类型My\_it::iterator\_category中检测到迭代器的种类信息；从My\_it::value\_type中得到元素的类型信息；从My\_it::distance\_type中得到差距类型信息。像My\_it这样特定形式的迭代器可以用来遍历一个元素类型为char的二进制文件。由于在这个特殊的例子使用了类型long作为My\_it的差距类型，因此在My\_it迭代器上进行的运算，与以前C库函数fseek和ftell遵循的规则相同（这两个函数也是以long来作为文件的偏移量）。

附带说一声，按照一般的惯例，输出迭代器都是由基类iterator<output\_iterator, void, void, void, void>所派生出来的。它并不需要向外提供参数类型T或者Dist。对于一个输出迭代器来说，我们唯一能做的就是用它存储一个元素，而且这个值不能再读回来。我们也不能够在这样的迭代器上面进行任何运算。造成所有这一切的原因就是迭代器种类的类型。

#### 4.1.3 迭代器种类

现在，通过一个小小的技巧，我们就可以来回答上面所列出的三个问题中的第一个了。当一个算法的实现方法不止一种时，我们就有必要考虑迭代器的种类问题。下面是一个来自STL头文件<algortthm>的简单例子。模板函数reverse用来反转一个序列（即第一个元素和最后一个交换，第二个和倒数第三个交换，以此类推）。通过所有模板函数iter\_swap来进行实际的交换动作，可以使用两个双向迭代器来完成这项工作：

template<class BidIs>

inline void reverse(BidIt first, BidIt last)

{

    for(; first!=last && first!=--last; ++first)

        Iter\_swap(first, last);

}

但是如果我们知道这两个迭代器实际上是随机迭代器的话，我们就可以把上面的代码改进得更快一些：

template<class RanIt>

inline void reverse(RanIt first, RanIt last)

{

for(; first<last; ++first)

Iter\_swap(first, --last);

}

如果所有的迭代器都基于上面给出的那些类的话，我们就可以采取一个小小的技巧，使得自己在这个二选一的选择中更加聪明一点。我们所需做的只是通过使用迭代器的种类信息，在被重载的模板函数Reverse的定义中进行选择：

template<class BidIt>

inline void reverse(BidIt first, BidIt last)

{

    Reverse(first, last, typename BitIt::iterator\_category());

}

template<class BidIt>

inline void Reverse(BidIt first, BidIt last, bidirectional\_iterator\_tag)

{

    for(; first!=last && first!=--last; ++first)

        iter\_swap(first, last);

}

template<class BidIt>

inline void Reverse(BidIt first, BidIt last, random\_access\_iterator\_tag)

{

    for(; first<last; ++first)

        iter\_swap(first, --last);

}

现在，对reverse的特化已经被解释成为对于某个Reverse的调用了。决定调用哪个Reverse完全依赖于迭代器的成员类型iterator\_category的定义，它提供模板参数的类型。在上面给出的函数调用中，构造函数BidIt::iterator\_category()返回了一个迭代器标签类型的对象，这个对象并没有被其他语句使用，它存在的唯一理由就是作为一个唯一不同的类型来参与重载函数的解析。

最终的结果就是：模板特化的机制在此被当作翻译时的switch语句使用。稍微智能的翻译器甚至能够在优化过程中将额外的函数调用及哑参数的设置给优化掉。这看起来很巧妙了，不是吗？

唔，它还不够巧妙。诚然，STL以自己能够与其他代码实现无缝连接而自豪。尤其是，它还允许我们在任意可以使用迭代器的地方使用对象指针来替代迭代器。而上面给出的方案却不适合单纯由指针所构成的迭代器。也就是说，不存在一种简单的方法来扩展Reverse的定义，使之能够处理任意类型的对象指针。我们需要进一步地改进已有的方法。

实际上，我们将描述解决这个问题的两种不同方法。第一种比较理想化，它目前也被C++标准所接纳了。不幸的是，在本书的书写过程中，这种方法所要求的语言特性还没有得到广泛推广。第二种比较实际，在早期的日子里，它也被采用以使得STL能够正常工作。

正式的解决方案是引入称为iterator\_traits的又一个模板类，它仅有的模板参数就是迭代器的类型，而它的属性也正是我们所希望获得的。

template<class It>

struct iterator\_traits{

typedef typename It::iterator\_category iterator\_category;

typedef typename It::value\_type value\_type;

typedef typename It::distance\_type distance\_type;

typedef typename It::pointer pointer;

typedef typename It::reference reference;

};

从表面上看，它好像没有增加什么。这个新类所干的不过是重复了其“可能是基于模板类iterator的”模板参数所定义过的类型定义而已。但是这新的模板类却提供了一个很好的机会，引入部分特化来处理所有的指针类型。

template<class T>

struct iterator\_traits<T\*>{

    typedef random\_access\_iterator\_tag iterator\_category;

    typedef T value\_type;

    typedef ptrdiff\_t distance\_type;

    typedef T \*pointer;

    typedef T& reference;

};

（库中也包含了对于常量指针的稍有不同的部分特化。）这个定义也就注定了所有的指针都是随机存取迭代器，并且其差距类型为ptrdiff\_t。你可能会想起：在C标准中，ptrdiff\_t被确定为一个整数类型，我们用它来表示指向同一数组中的两个对象指针之间的差距，在标准C的头文件<stddef.h>中，使用typedef定义了它。

现在，我们可以将模板函数reverse重写如下：

template<class BidIt>

inline void reverse(BidIt first, BidIt lase)

{

    Reverse(first, last, typename iterator\_traits<BidIt>::iterator\_category());

}

如果BitIt是一个基于模板类iterator的类，那么模板类iterator\_traits就被特化成一般形式。成员类型iterator\_category也就被定义为BidIt::iterator\_category。但如果BitIt是一个指针类型的话，这时部分特化就起作用了。成员类型iterator\_category也就代表了random\_access\_iterator\_tag。

然而，现阶段模板的部分特化并没有普遍获得支持。我们还需要一种可靠的方案来完成这项工作。幸运的是，函数重载正好能提供我们所需的“魔法”。我们以两种方式定义了模板函数Iter\_cat：

template<class C, class T, class Dist>

inline C Iter\_cat(const iterator<C, T, Dist>&)

{

    C x;

    return x;

}

template<class T>

inline random\_access\_iterator\_tag Iter\_cat(const T\*)

{

    random\_access\_iterator\_tag x;

    return x;

}

附带说一句，在STL的最初实现中，这个模板函数被命名为iteratro\_category。这个名字没有被C++标准保留下来，所以我们使用了另一个为实现者保留的名字（至少我们还保留了以下划线开头的那些名字）。同样，在理论上，这两个函数都可以简单地返回一个构造函数调用的结果（如：C()）。在此处所使用的那种稍显啰嗦的形式避免了在现有的编译器中所存在的一些问题。和其他大部分可行的C++代码一样，本书所提供的代码也偶尔因为实际中的需要而显得有点奇怪。

现在，我们可以将模板函数reverse重写如下：

template<class BidIt>

inline void reverse(BidIt first, BidIt last)

{

Reverse(first, last, Iter\_cat(first));

}

如果BidIt是一个基于模板类iterator的类，那么模板函数Iter\_cat的第一种形式就会被调用。这个函数返回一个对象，其类型与iterator的第一个模板参数类型(即BitIt::iterator\_category)相同。但如果BitIt是一个指针类型的话，这时被调用的就是模板函数的第二种形式了。此时函数明确地返回一个类型为random\_access\_iterator\_tag的对象。

本书中的代码使用了模板函数Iter\_cat，而不是模板类iterator\_traits。这使得它们可以很好地适应过渡时期的编译器。

现在，轮到我们来回答前面剩下来的那两个问题了。如果我们想知道一个类型为It的迭代器所关联的元素的类型T，原则上可以写出它的类型名称：iterator\_traits<It>::value\_type。但当It是一个对象指针时，这仍然依赖于部分特化。为了避免出现这个问题，我们在本书中的代码里面使用了另一个模板函数Val\_type。如果编译器支持部分特化，我们就可以把这个模板函数简单地写作：

template<class It>

inline typename iterator\_traits<It>::value\_type(It)

{

    return 0;

}

如果编译器不支持部分特化，我们也可以通过一对重载函数来达到同样的效果：

template<class C, class T, class D, class Pt, class Rt>

inline T \*val\_type(iterator, T, D, Pt, Rt)

{

    return 0;

}

template<class T>

inline T \*val\_type(const T\*)

{

    return 0;

}

（原来的名字value\_type现在语句不再被C++标准所允许了，商业化的实现一般都用一个其他的名字来代替它，如：\_Val\_type。）

这个函数的任何版本都没有返回一个我们所关注的类型为T的对象。如果那样的话将会在该类型描述的对象很大的情况下出现各种问题，或是在构造和析构中出现许多“有趣的”副作用，又或是仅仅为了操作和复制而不得不等上很长时间。为了避免这些问题的出现，我们使这些函数返回的都是一个指向所期望类型的指针。而这对于模板实现其“魔法”来说，已经足够了。

例如，在此我们给出在前面的例子中被reverse调用的模板函数iter\_swap的实现：

template<class FwdIt1, class FwdIt2>

inline void iter\_swap(FwdIt1 x, FwdIt2 y)

{

Iter\_swap(x, y, Val\_type(x));

}

template<class FwdIt1, class FwdIt2, class T>

inline void Iter\_swap(FwdIt1 x, FwdIt2 y, T\*)

{

T tmp = \*x;

\*x = \*y;

\*y = tmp;

}

为了交换两个值，需要使用一个临时对象来保存其中的一个值。于是iter\_swap调用了一个模板函数Iter\_swap，并增加了一个额外的参数。Val\_type的返回类型T\*也正好匹配Iter\_swap的参数类型T\*。也就是说，我们已经从作为基类的迭代器中（或是从作为一个迭代器使用的对象指针中）得到了元素的类型T，然后就可以把它传递到任意需要使用它的函数中去。

同样，如果我们想要确定计量两个类型为It的迭代器之间差距的类型时，可以简单而直接地写出类型的名称：iteratro\_traits<It>::distance\_type。实际上，我们在本书中的代码里使用了模板函数Dist\_type。使用了部分特化的版本看起来如下所示：

template<class It>

inline typename iterator\_traits::difference\_type \*Dist\_type(It)

{

    return 0;

}

而没有使用部分特化的替代版本看起来如下所示：

template<class C, class T, class D, class Pt, class Rt>

inline D \*Dist\_type(iterator<C, T, D, Pt, Rt>)

{

    return 0;

}

template<class T>

inline ptrdiff\_t \*Dist\_type(const T\*)

{

    return 0;

}

(原来的名字是distance\_type。)

有两种操作经常被各种迭代器使用，它们并不总是表达为操作符的形式：

* X += N将一个Dist对象N加到迭代器X上。只有当X是双向迭代器或随机存取迭代器时，N才可以为负。
* N = X2 - X1把两个迭代器X2和X1之间的差距赋给N。只有当迭代器的种类是随机存取时，N才可能为夫。

当然，对于任意的输出迭代器来说，上面的操作毫无意义。对于输入迭代器来说，也几乎没有意义，因为上面的任意一种操作都将使得输入迭代器无效。STL包含它们更多地是为了完整性而不是通用性。

诚然，使用这样的操作符表达形式很方便，但在最初的STL中却很少使用它们。诸如X+=N及X2-X1的形式仅为随机存取迭代器所需要。它们不可能被那些功能稍差的迭代器种类（即输入，输出，前向，双向迭代器）完全支持。对于迭代器所支持的简单操作，其时间复杂度应该为分摊付出的常量时间(amortized constant time)，例如，取得两个迭代器之间差距的时间不应该随其差距的大小而变化。我们甚至被误导去定义具有线性时间复杂度的操作。为了避免这些，STL中的算法使用了如下两个模板函数：

template<class InIt, class Dist>

inline void advance(InIt& p, Dist n);

template<class Init>

inline typename iterator\_traits<InIt>::distance\_type distance(InIt first, InIt last);

第一个模板函数实质上计算p+=n；第二个返回n=last-first，但这一切都是建立在迭代器的种类能够使其结果有意义的前提下。如果这些函数中的一个对于给定的迭代器种类所花费的时间不是一个常量，那么那些额外的时间复杂度就将会被纳入到考虑的范围内，用来检测那些调用该函数的算法的时间复杂度。

当我们成功地引入了iterator\_traits时，模板类distance同样也能够说明上述情况。模板函数Dist\_type不能根据上面给出的distance定义的需求，及时地把它的回答传递给外界，以定义模板函数的返回值。所以，在这个实现中，我们将distance定义为返回类型ptrdiff\_t。但为了求稳，我们按照一贯的方式，在应该所以它的地方所以了模板函数Distance，其声明如下：

template<class InIt, class D>

inline void Distance(InIt first, InIt last, D n0);

它计算n0 += last-first，并返回一个对象，其类型就是我们所希望的差距类型D(Distance是早先版本中模板函数distance重命名后的名字)。

也就是说，在这个实现，我们可以避免要求翻译器必须支持模板的部分特化。只要模板类能够明确地利用这个语言特性就可以了。

#### 4.1.4 专用迭代器

头文件<iterator>中剩余的其他部分为一些专门的迭代器定义了一个分类：

* 反转型迭代器(reverse iterator)
* 插入型迭代器(insertion iterator)
* 流迭代器(stream iterator)
* 流缓冲迭代器(stream buffer iterator)

#### 4.1.5 反转型迭代器

如同其名字所暗示的那样，在使用反转型迭代器向前遍历时，它实际上是在往回遍历一个序列。同样，如果用它向后遍历，它实际上是在向前遍历序列。这是一个很方便的工具，尤其在需要操作一个逆序序列时更加有用——这时不需要真正地去“反转”这个序列，然后在完成操作后再把它转回去，只需完成反向的操作就可以了。

模板类reverse\_iterator是一个随机存取迭代器，它被声明为：

template<class RanIt>

class reverse\_iterator;

在此，RanIt就是我们想要使之“回遍”的那个基本的随机存取迭代器的类型。我们这样定义：

typedef reverse\_iterator<char\*> Revptr;

它定义了一个可以在由char组成的数组中“回退”的迭代器类型Revptr。

这个模板类实际上只存储了一个对象，也就是由类RanIt所得到的那个基本的迭代器对象。任何人都可以通过调用它的成员函数base()来获得这个对象的值。从reverse\_iterator所派生出来的类也可以通过使用保护性的对象current来存取它。

附带说一句，模板类reverse\_iterator对于我们选择开发的任何随机存取迭代器来说都是一个非常好的原型。它提供了所有必需的成员函数。在头文件<iterator>中还同样提供了所有我们可能需要为它定义的模板操作符。我们将在容器模板类vector(参见第11章)和deque(参见第13章)中利用这种特殊的迭代器。

#### 4.1.6 反转型双向迭代器

从前，在STL中还存在另外一个模板类reverse\_bidirectional\_iterator。它的意图很明显，就是把一个双向迭代器变成一个反转型迭代器。然而，在后来的标准化进程中，这个模板类消失。标准化委员会认定模板类reverse\_iterator同样可以完成这项工作。

为了使得上面的替代工作得以顺利完成，我们需要对语言本身有所改进。在随机存取迭代器中，我们定义了一些不可能被双向迭代器支持的成员函数，如operator+=，它用来将一个整型偏移量加到迭代器上。

许多旧版本的编译器会检测所有的模板成员函数(甚至在该函数从未被调用的情况下也是如此)，以判断表达式是否有效。标准化委员会最终决定，只有那些被调用的成员函数才应该被检测。这个改动解决了在定义operator->的迭代器模板类中存在着的应该相似问题，因为只有在迭代器指向的类型是类、结构体或联合时，operator->才有意义。

然而，编译器还需要一些时间来赶上这些变化。在对双向迭代器使用reverse\_iterator时，有的编译器仍然会给出出错消息。为此，我们在此处提供的实现仍然保留了模板类reverse\_bidirectional\_iterator，不过名字换成了Revbidit。你将不会再看到这个模板类以这种表达方式出现，但有些编译器却会在容器模板类list(参见第12章）、set和multiset(参见第14章)以及map和multimap(参见第15章)中利用它。

在此我们保留下模板类Revbidit还有另一个原因。对于我们所选择开发的双向迭代器来说，它是一个非常好的原型。它提供了所有必需的成员函数，其中包括所有双向迭代器所要求支持的模板操作符。

#### 4.1.7 插入型迭代器

我们经常需要通过“从头到尾来一遍”来产生一个元素序列。处理这些产生出来的元素的最完美的代理就是输出迭代器。在所有允许往序列中写入的迭代器类型中，输出迭代器具有最少的语义需求。这些简单需求让我们得以以相当大的自由度来写输出迭代器。

举例来说，我们希望往类型为Cont的容器对象C中插入我们所产生的序列。在STL中所定义的容器模板类都提供了许多统一的可供我们利用的属性。尤其是：

* 每个容器都定义它的元素类型为Cont::value\_type，把用于被控序列的迭代器类型定义为Cont::iterator。
* 所有支持在常数时间内向被控序列末端插入新元素的容器都定义了成员函数：

C.push\_back(Cont::value\_type&)

* 所有支持在常数时间内向被控序列前端插入新元素的容器都定义了成员函数：

C.push\_front(Cont::value\_type)

* 所有支持在常数时间内向被控序列中由迭代器指定的任意位置处插入新元素的容器都定义了成员函数：

C.insert(Cont::value\_type&, Cont::iterator)

最后那个函数返回一个指向被插入元素的迭代器。

当然，我们也可以遵照这些规则来定义自己的容器。STL也鼓励我们按照这样的模型去做，以使我们添加的部分可以很容易地和STL的其他部分融合在一起。

头文件<iterator>中定义了几个使用了这些成员函数的模板类：

* back\_insert\_iterator<Cont>是一个输出迭代器，它用来将产生的元素添加到类型为Cont的容器对象的末端。
* front\_insert\_iterator<Cont>是一个输出迭代器，它用来将产生的元素添加到容器的前端（是的，产生出来的元素以逆序的方式结束于被控序列前端）。
* insert\_iterator<Cont, Iter>是一个输出迭代器，它用来将产生的元素插入到一个由迭代器指定的元素前面。

每个这样的模板类中都存储了一个指向容器对象的引用，其名字为保护型的container。最后的那个模板类(insert\_iterator)还在对象中存储了插入点，它拥有保护型的名字iter。

头文件<iterator>也定义了三个相关的模板函数：

* back\_inserter(Cont&)对于容器参数返回一个back\_insert\_iterator。
* front\_inserter(Cont&)对于容器参数返回一个front\_insert\_iterator。
* inserter(Cont&)对于容器参数和迭代器参数返回一个insert\_iterator

这些函数提供了一种方便的方法，用以实时产生出一个插入型迭代器。

#### 4.1.8 流迭代器

如果我们可以通过“聪明地”所有输出迭代器，将产生的序列传给一个容器的话，我们就同样可以通过某种“聪明的”形式，将序列插入到一个输出流(output stream)中。同样，我们也可以通过“聪明地”使用输入迭代器，从一个输入流(input stream)中获得一个元素序列。实际上，这就是我们将要讨论的东西。

假设我们有类型为T的元素，且对于了从流中提取值的操作：

istream& operator>>(istream&, T&);

我们所需做的只是从输入流(如cin)中读取所需的字符，将其转化为一个类型为T的值，然后存储到对象X中去，这么写：

cin>>X;

模板类istream\_iterator<T>为我们做到了这一点。从表面上看，它是通过使用一个输入迭代器来从序列中得到一个元素（该模板类还有其他的参数，它们用来处理其他的更为花哨的输入流，但默认值已足以应付广泛使用的istream的要求了）。如果我们这样写：

typedef stream\_iterator<int, ptrdiff\_t> Int\_init;

Int\_init int\_in(cin);

那么表达式\*int\_in++就会从cin中提取出一个int，并把它作为该表达式的值传递。

当然，我们首先一个判断在输入流中这个值是否有效。适当的方法是将int\_it与一个标明文件结束的end-of-sequence值进行相等性比较（见第1章的“end-of-sequence值”）。默认构造函数总是产生具有end-of-sequence值的对象。也就是说，我们可以这么写：

Int\_init first(cin), last;

然后，就可以从流中不断地提取元素，直到遇到文件结束为止。我们可以这么写（以固定格式的循环方式）：

for(p=first; p!=last; ++p)

    <process>(\*p);

或它的变体：

while(first != last)

    <process>(\*first++);

头文件<iterator>中也定义了模板类ostream\_iterator。它定义了一个用来向输出流中插入元素的输出迭代器。假设我们有类型为T的元素，且定义了向流中插入值的操作：

ostream& operator<<(ostream&, T);

我们所需做的只是将一个类型为T的值转化为一个字符序列，并把这个序列写入到输出流(如cout)中，这么写：

cout<<X;

模板类ostream\_iterator<T>为我们做到了这一点。从表面上看，它是通过使用一个输出迭代器来向序列中存储一个元素（再次重申一遍，该模板类还有其他的参数，它们用来处理其他更为花哨的输出流，但默认值已足以应付广泛使用的ostream的要求了）。如果我们这样写：

typedef ostream\_iterator<int> Int\_outit;

Int\_outit int\_out(cout);

那么表达式\*int\_out++=X就会将类型为int的值X插入到cout中。更好的一种声明方式为：

Int\_outit int\_out(cout, “ “);

它在每个被插入的值后面都添加了一个空格。这样，插入值所产生的字段就不会连成一片，不好辨别了。

#### 4.1.9 流缓冲迭代器

STL中一个后来添加的部分是一对迭代器，它们用来直接向一个流缓冲区(stream\_buffer)中插入提取类型为E(通常为char)的元素。模板类basic\_streambuf<E, T>定义了一个流缓冲区对象，它被用在实际的流与出现存储器之间，起传输媒介的作用。它是原有的IOStream类streambuf的模板化版本，类似于标准C库中的类型FILE。在此，T表示一个traits类，通常用来提供元素类型E的各种信息。

由于iostream以及被标准化了，因此我们不去研究它的纷繁难懂之处，那将会是一个非常大的主题。在此，我们只需知道模板类istreambuf\_iterator<E, T>在新增的方法里面起着重要的作用就行了。从输入流中所读取的每个字符其实都是通过这种类型的输入迭代器而间接获得的。与之相似的是，写到输出流中的每个字符，其实也是通过类型为ostreambuf\_iterator<E, T>的一个输出迭代器而间接地写到输出流上的。

在头文件<iterator>中所定义的方法被贯穿地使用于STL中。我们应该让自己足够地熟悉这个头文件，这样才可能很舒服地使用它。如果没有大量使用在本章开始处所描述的迭代器的属性，我们几乎不可能写处应该很好的算法来。在使用容器时，我们通常会惊讶于使用插入型迭代器是多么方便。在真正处理有关iostreams的工作时，我们几乎不可避免地至少要和流缓冲迭代器打交道。

让我们最后一次强调，这个头文件有着非常重要的应用。在所有的五种迭代器中，只有前向迭代器在头文件中没有任何原型的定义。其他的几种都给我们提供了极好的模型。我们只需大胆使用它们就可以了。

## 4.2 功能描述

namespace std{

    struct input\_iterator\_tag;

    struct output\_iterator\_tag;

    struct forward\_iterator\_tag;

    struct bidirectional\_iterator\_tag;

    struct random\_access\_iterator\_tag;

    // TEMPLATE CLASSES

    template<class C, class T, class Dist, class Pt, class Rt>

    struct iterator;

    template<class It>

    struct iterator\_traits;

    template<class T>

    struct iterator\_traits<T\*>;

    template<class RanIt>

    struct reverse\_iterator;

    template<class Cont>

    struct back\_insert\_iterator;

    template<class Cont>

    struct front\_insert\_iterator;

    template<class Cont>

    struct insert\_iterator;

    template<class U, class E, class T, class Dist>

    struct istream\_iterator;

    template<class U, class E, class T>

    struct ostream\_iterator;

    template<class E, class T>

    struct istreambuf\_iterator;

    template<class E, class T>

    struct ostreambuf\_iterator;

    //TEMPLATE FUNCTIONS

    template<class RanIt>

    bool operator==(const reverse\_iterator<RanIt>& lhs, const reverse\_iterator<RanIt>& rhs);

    template<class U, class E, class T, class Dist>

    bool operator--(const istream\_iterator<U, E, T, Dist>& lhs, const istream\_iterator<U, E, T, Dist>& rhs);

    template<class E, class T>

    bool operator==(const istreambuf\_iterator<E, T>& lhs, const istreambuf\_iterator<E, T>& rhs);

    template<class RanIt>

    bool operator!=(const reverse\_iterator<RanIt>& lhs, const reverse\_iterator<RanIt>& rhs);

    template<class U, class E, class T, class Dist>

    bool operator!=(const istream\_iterator<U, E, T, Dist>& lhs, const istream\_iterator<U, E, T, Dist>& rhs);

    template<class E, class T>

    bool operator!=(const istreambuf\_iterator<E, T>& lhs, const istreambuf\_iterator<E, T>& rhs);

    template<class RanIt>

    bool operator<(const reverse\_iterator<RanIt>& lhs, const reverse\_iterator<RanIt>& rhs);

    template<class RanIt>

    bool operator>(const reverse\_iterator<RanIt>& lhs, const reverse\_iterator<RanIt>& rhs);

    template<class RanIt>

    bool operator<=(const reverse\_iterator<RanIt>& lhs, const reverse\_iterator<RanIt>& rhs);

    template<class RanIt>

    bool operator>=(const reverse\_iterator<RanIt>& lhs, const reverse\_iterator<RanIt>& rhs);

    template<class RanIt>

    Dist operator-(const reverse\_iterator<RanIt>& lhs, const reverse\_iterator<RanIt>& rhs);

    template<class RanIt>

    reverse\_iterator<RanIt> operator+(Dist n, const reverse\_iterator<RanIt>& rhs);

    template<class Cont>

    back\_insert\_iterator<Cont> back\_inserter(Cont& x);

    template<class Cont>

    front\_insert\_iterator<Cont> front\_inserter(Cont& x);

    template<class Cont, class Iter>

    insert\_iterator<Cont> inserter(Cont& x, Iter it);

    template<class InIt, class Dist>

    void advance(InIt& it, Dist n);

    template<class InIt>

    iterator\_traits<InIt>::difference\_type distance(InIt first, InIt last);

};

包含STL标准头文件<iterator>就可以得到一系列类、模板类及模板函数的定义，这都对迭代器的声明及操作有着辅助作用。

**advance**

template<class InIt, class Dist>

void advance(InIt& it, Dist n);

该模板函数提供将it递增n次后，最终有效地向前移动了it。如果InIt是一个随机存取迭代器的话，函数就只计算表达式it+=n。否则，函数通过计算++it来完成每次递增。当InIt是一个输入或前向迭代器时，n不允许为负。

**back\_insert\_iterator**

template<class Cont>

class back\_insert\_iterator

:public iterator<output\_iterator\_tag, void, void, void, void>

{

public:

    typedef Cont container\_type;

    typedef typename Cont::reference reference;

    typedef typename Cont::value\_type value\_type;

    explicit back\_insert\_iterator(Cont& x);

    back\_insert\_iterator& operator=(typename Cont::reference val);

    back\_insert\_iterator& operator\*();

    back\_insert\_iterator& operator++();

    back\_insert\_iterator& operator++(int);

protected:

    Cont \*container;

};

该模板类描述了一个输出迭代器对象。它将元素插入到一个类型为Contd容器中，我们也可以通过它所存储的保护型指针对象container来存取该容器。这个容器必须定义：

* 成员类型const\_reference，即容器所控制的序列元素中元素的常量引用(constant reference)的类型
* 成员类型reference，即容器所控制的序列中元素的引用(reference)的类型
* 成员类型value\_type，即容器所控制的序列中的元素的类型
* 成员函数push\_back(value\_type c)，它可以将一个值为c的新元素添加到序列的末端

**back\_insert\_iterator::back\_insert\_iterator**

explict back\_insert\_iterator(Cont& x);

该构造函数将container初始化为x。

**back\_insert\_iterator::container\_type**

typedef Cont container\_type;

该类型为模板参数Cont的同义词。

**back\_insert\_iterator::operator\***

back\_insert\_iterator& operator\*();

该成员函数返回\*this。

**back\_insert\_iterator::operator++**

back\_insert\_iterator& operator++();

back\_insert\_iterator& operator++(int);

这两个成员函数都返回\*this。

**back\_insert\_iterator::operator=**

back\_insert\_iterator& operator=(typename Cont::const\_reference val);

该成员函数先完成container.push\_back(val)的动作，然后返回\*this。

**back\_insert\_iterator::reference**

typedef typename Cont::reference reference;

该类型描述了由关联容器所控制的序列中的元素的引用。

**back\_insert\_iterator::value\_type**

typedef typename Cont::value\_type value\_type;

该类型描述了由关联容器所控制的序列中的元素

**back\_inserter**

template<class Cont>

back\_insert\_iterator<Cont> back\_inserter(Cont& x);

该模板函数返回back\_insert\_iterator<Cont>(x)。

**bidirectional\_iterator\_tag**

struct bidirectional\_iterator\_tag: public forward\_iterator\_tag{};

当It描述的对象可以当作双向迭代器来使用时，该类型等价于

iterator<It>::iterator\_category。

**distance**

template<class InIt>

typename iterator\_traits<InIt>::difference\_type distance(InIt first, InIt last);

该模板函数先将一个计数器n设为0，然后不断地向前移动first，并在移动过程中对n进行递增，直到first==last。如果InIt是一个随机存取迭代器的话，那该函数就只计算表达式n+=last-first。否则，它通过计算++first完成每次迭代器递增。

**forward\_iterator\_tag**

struct forward\_iterator\_tag: public input\_iterator\_tag{};

当It描述的对象可以当作前向迭代器来使用时，该类型等价于

iterator<It>::iterator\_category。

**front\_insert\_iterator**

template<class Cont>

class front\_insert\_iterator: public iterator<output\_iterator\_tag, void, void, void, void>{

public:

    typedef Cont container\_type;

    typedef tpename Cont::reference reference;

    typedef tpename Cont::value\_type value\_type;

    explict front\_insert\_iterator(Cont& x);

    front\_insert\_iterator& operator=(tpename Cont::const\_reference val);

    front\_insert\_iterator& operator\*();

    front\_insert\_iterator& operator++();

    front\_insert\_iterator& operator++(int);

protected:

    Cont\* container;

};

该模板类描述了一个输出迭代器对象。它将元素插入到一个类型为Cont容器中，可以通过它所存储的保护型指针对象container来存取该容器。这个容器必须定义：

* 成员类型const\_reference，即容器所控制的序列中元素的常量引用的类型。
* 成员类型reference，即容器所控制的序列中元素的引用的类型。
* 成员类型value\_type，即容器所控制的序列中元素的类型。
* 成员函数push\_front(value\_type c)，它可以将一个值为c的新元素添加到序列的前端。

**front\_insert\_iterator::container\_type**

typedef Cont container\_type;

该类型为模板参数Cont的同义词。

**front\_insert\_iterator::front\_insert\_iterator**

explict front\_insert\_iterator(Cont& x);

该构造函数将container初始化为&x。

**front\_insert\_iterator::operator\***

front\_insert\_iterator& operator\*();

该成员函数返回\*this。

**front\_insert\_iterator::operator++**

front\_insert\_iterator& operator++();

front\_insert\_iterator& operator++(int);

这两个成员函数都返回\*this。

**front\_insert\_iterator::operator=**

front\_insert\_iterator& operator=(typename Cont::const\_reference val);

该成员函数先完成container.push\_front(val)的动作，然后返回\*this。

**front\_insert\_iterator::reference**

typedef typename Cont::reference reference;

该类型描述了由关联容器所控制的序列中的元素的引用。

**front\_insert\_iterator::value\_type**

typedef typename Cont::value\_type value\_type;

该类型描述了由关联容器所控制的序列中的元素。

**front\_inserter**

template<class Cont>

front\_insert\_iterator<Cont> front\_inserter(Cont& x);

该模板函数返回front\_insert\_iterator<Cont>(x)

**input\_iterator\_tag**

struct input\_iterator\_tag{};

当It描述的对象可以当作输入迭代器来使用时，该类型等价于

iterator<It>::iterator\_category

**insert\_iterator**

template<Cont>

class insert\_iterator: public iterator<output\_iterator\_tag, void, void, void, void>

{

public:

    typedef Cont container\_type;

    typedef typename Cont::reference referencd;

    typedef typename Cont::value\_type value\_type;

    insert\_iterator(Cont& x, typename Cont::iterator it);

    insert\_iterator& operator=(typename Cont::const\_reference val);

    insert\_iterator& operator\*();

    insert\_iterator& operator++();

    insert\_iterator& operator++(int);

protected:

    Cont\* container;

    typename Cont::iterator iter;

};

该模板类描述了一个输出迭代器对象。它将元素插入到一个类型为Cont的容器中，可以通过它所存储的保护型指针对象container来存取该容器。它还存储了类Cont::iterator的保护型迭代器对象iter。这个容器必须定义：

* 成员类型const\_reference，即容器所控制的序列中元素的常量引用类型
* 成员类型iterator，即容器的迭代器类型
* 成员类型reference，即容器所控制的序列中元素的引用类型
* 成员类型value\_type，即容器所控制的序列中元素的类型
* 成员函数insert(iterator it, value\_type c)，它可以将一个值为c的新元素插入到被控序列中，位置紧接着由it指定的元素的前面，并返回指向被插入元素的那个迭代器。

**insert\_iterator::container\_type**

typedef Cont container\_type;

该类型为模板参数Cont同义词

**insert\_iterator::insert\_iterator**

insert\_iterator::(Cont& x, typename Cont::iterator it);

该构造函数将container初始化为&x，将iter转化为it。

**insert\_iterator::operator\***

insert\_iterator::operator\*();

该成员函数返回\*this。

**insert\_iterator::operator++**

insert\_iterator::operator++();

insert\_iterator::operator++(int);

这两个成员函数都返回\*this。

**insert\_iterator::operator=**

insert\_iterator& operator=(typename Cont::const\_reference val);

该成员函数先完成iter=container.insert(iter, val)的动作，然后返回\*this。

**insert\_iterator::reference**

typedef typename Cont::reference reference;

该类型描述了由关联容器所控制的序列中的元素的引用。

**insert\_iterator::value\_type**

typedef typename Cont::value\_type value\_type;

该类型描述了由关联容器所控制的序列中的元素。

**inserter**

template<class Cont, class Iter>

insert\_iterator<Cont> inserter(Cont& x, Iter it);

该模板函数返回insert\_iterator<Cont>(x, it);

**istream\_iterator**

template<class U, class E=char, class T=chr\_traits, class Dist=ptrdiff\_t>

class istream\_iterator: public iterator<input\_iterator\_tag, U, Dist, U\*, U&>

{

public:

    typedef E char\_type;

    typede T traits\_type;

    typedef basic\_istream<E, T> istream\_type;

    istream\_iterator();

    istream\_iterator(istream\_type& is);

    const U& operator\*() const;

    const &\* operator->() const;

    istream\_iterator<U, E, T, Dist>& operator++();

    istream\_iterator<U, E, T, Dist> operator++(int);

};

该模板类描述了一个输入迭代器对象。它可以从一个输入流中提取类U的对象，可通过它所存储的类型为指向basic\_istream<E, T>的指针的对象存取输入流。在通过使用一个非空指针来构造或递增一个类istream\_iterator的对象后，该对象就会试图从与之相关的输入流中提取并存储一个类型为U的对象。如果提取操作失败的话，该对象就会用一个空指针有效地替换掉它以前所存储的指针（也就是说，产生一个end-of-sequence指示符）。

**istream\_iterator::char\_type**

typedef E char\_type;

该类型为模板参数E的同义词

**istream\_iterator::istream\_iterator**

istream\_iterator();

istream\_iterator(istream\_type& is);

第一个构造函数指向输入流的指针初始化为一个空指针。第二个构造函数将指向输入流的指针初始化为&is，然后试图用它来提取并存储一个类型U的对象。

**istream\_iterator::istream\_type**

typedef basic\_istream<E, T> istream\_type;

该类型为basic\_istream<E, T>的同义词

**istream\_iterator::operator\***

const U& operator\*() const;

该操作符返回所存储的类型为U的对象

**istream\_iterator::operator->**

const U\* operator->() const;

该操作符返回&\*\*this

**istream\_iterator::operator++**

istream\_iterator<U, E, T, Dist>& operator++();

istream\_iterator<U, E, T, Dist>& operator++(int);

第一个操作符试图从关联输入流中提取并存储一个类型为U的对象。第二个操作符将该对象复制了一份，然后将其递增，最终返回它的那份拷贝。

**istream\_iterator::traits\_type**

typedef T traits\_type;

该类型为模板参数T的同义词。

**istreambuf\_iterator**

template<class E, class T=char\_traits<E>>

class istreambuf\_iterator

: public iterator<input\_iterator\_tag, E, typename T::off\_type, E\*, E&>

{

public:

    typedef E char\_type;

    typedef T traits\_type;

    typede typename T::int\_type int\_type;

    typedef basic\_streambuf<E, T> streambuf\_type;

    typedef basic\_istream<E, T> istream\_type;

    istreambuf\_iterator(streambuf\_type \*sb=0) throw();

    istreambuf\_iterator(istream\_type& is) throw();

    const E& operator\*() const;

    const E\* operator->() const;

    istreambuf\_iterator& operator++();

    istreambuf\_iterator& operator++(int);

    bool equal(const istreambuf\_iterator& rhs) const;

};

该模板类描述了一个输入迭代器对象。它可以从一个输入流缓冲区中提取类E的元素，可通过它存储的类型为指向basic\_streambuf<E, T>的指针的对象存取流缓冲区。在通过使用一个非空指针来构造或递增一个类istreambuf\_iterator的对象后，该对象就会试图从与之相关的输入流中提取并存储一个类型为E的对象（然而，提取操作也可以延迟到该对象被用于间接取值或拷贝时）。如果提取操作失败的话，该对象就会用一个空指针来替换掉它以前所存储的指针（也就是说，产生一个end-of-sequence指示符）。

**istreambuf\_iterator::char\_type**

typedef E char\_type;

该类型是模板参数E的同义词

**istreambuf\_iterator::equal**

bool equal(const istreambuf\_iterato& rhs) const;

只有当存储在该istreambuf\_iterator对象中的输入流缓冲区指针于存储在rhs中的同时为空或同时不为空时，该成员函数才返回true

**istreambuf\_iterator::int\_type**

typedef typename T::int\_type int\_type;

该类型是T::int\_type的同义词

**istreambuf\_iterator::istream\_type**

typedef basic\_istream<E, T> istream\_type;

该类型是basic\_istream<E, T)的同义词

**istreambuf\_iterator::istreambuf\_iterator**

istreambuf\_iterator(streambuf\_type\* sb=0) throw();

istreambuf\_iterator(istreambuf\_type& is) throw();

第一个构造函数将输入流缓冲区指针初始化为sb。第二个构造函数将输入流缓冲区指针初始化为is.rdbuf()，然后试图提取并存储一个类型为E的对象。

**isttreambuf\_iterator::operator\***

const E& operator\*() const;

该操作符返回所存储的类型为E的对象

**istreambuf\_iterator::operator++**

istreambuf\_iterator& operator++();

istreambuf\_iterator& operator++(int);

第一个操作符试图从相关输入流中提取并存储一个类型为E的对象。第二个操作符则将该对象复制了一份，然后将其递增，最终返回那份拷贝。

**istreambuf\_iterator::operator->**

const E\* operator->() const;

该操作符返回&\*\*this。

**istreambuf\_iterator::streambuf\_type**

typedef basic\_streambuf<E, T> streambuf\_type;

该类型为basic\_streambuf<E, T>的同义词。

**istreambuf\_iterator::traits\_type**

typedef T traits\_type;

该类型为模板参数T的同义词

**iterator**

template<class C, class T, class Dist=ptrdiff\_t, class Pt=T\*, class Rt=T&>

struct iterator{

    typedef C iterator\_category;

    typedef T value\_type;

    typedef Dist difference\_type;

    typedef Pt pointer;

    typedef Rt reference;

};

该模板可以很方便地用来当作我们所定义的迭代器的基类。它定义了成员类型iterator\_category（模板参数C的同义词）、value\_type（模板参数T的同义词）、difference\_type（模板参数Dist的同义词）、pointer（模板参数Pt的同义词）以及reference（模板参数Rt的同义词）。

注意，即便当pointer指向的对象与reference代指的对象都具有常量属性时，value\_type也不必是一个常量类型。

**iterator\_traits**

template<class It>

struct iterator\_traits{

    typedef typename It::iterator\_category iterator\_category;

    typedef typename It::value\_type value\_type;

    typedef typename It::difference\_type difference\_type;

    typedef typename It::pointer pointer;

    typedef typename It::reference reference;

};

template<class T>

struct iterator\_traits<T\*>{

    typedef random\_access\_iterator\_tag iterator\_category;

    typedef T value\_type;

    typedef prtdiff\_t difference\_type;

    typedef T\* pointer;

    typedef T& refernce;

};

template<class T>

struct iterator\_traits<const T\*>{

    typedef random\_access\_iterator\_tag iterator\_category;

    typedef T value\_type;

    typedef ptrdiff\_t difference\_type;

    typedef const T\* pointer;

    typedef const T& reference;

};

该模板类定义了几个与迭代器类型It密切相关的关键性类型。它定义了成员类型iterator\_category（It::iterator\_category的同义词）、value\_type（It::value\_type的同义词）、difference\_type(It::difference\_type的同义词)、pointer(It::pointer的同义词)以及reference(It::reference的同义词)。

部分特化的版本中定义了几个与之对象指针类型T\*密切相关的关键性类型。在这个实现中，我们还可以使用一些没有用到部分特化的模板函数：

template<class C, class T, class Dist>

C Iter\_cat(const iterator<C, T, Dist>&);

template<class T>

random\_access\_iterator\_tag Iter\_cat(const T\*);

template<class C, class T, class Dist>

T\* Val\_type(const iterator<C, T, Dist>&);

template<class T>

T\* Val\_type(const T\*);

template<class C, class T, class Dist>

Dist\* Dist\_type(const iterator<C, T, Dist>&);

template<class T>

ptrdiff\_t\* Dist\_type(const T\*);

这些函数也能够为我们提供所需的同样类型（只不过使用的方法有点儿间接）。我们可以在函数调用时将这些函数作为参数。它们存在的唯一目的就是：为我们所调用的函数提供有用的模板类参数。

**operator!=**

template<class RanIt>

bool operator!=(

    const reverse\_iterator<RanIt>& lhs,

    const reverse\_iterator<RanIt>& rhs);

template<class U, class E, class T, class Dist>

bool operator!=(

    const istream\_iterator<U, E, T, Dist>& lhs,

    const istream\_iterator<U, E, T, Dist>& rhs);

template<class E, class T>

bool operator!=(

    const istreambuf\_iterator<E, T>& lhs,

    const istreambuf\_iterator<E, T>& rhs);

该模板操作符返回!(lhs==rhs)。

**operator==**

template<class RanIt>

bool operator==(

    const reverse\_iterator<RanIt>& lhs,

    const reverse\_iterator<RanIt>& rhs);

template<class U, class E, class T, class Dist>

bool operator==(

    const istream\_iterator<U, E, T, Dist>& lhs,

    const istream\_iterator<U, E, T, Dist>& rhs);

template<class E, class T>

bool operator==(

    const istreambuf\_iterator<E, T>& lhs,

    const istreambuf\_iterator<E, T>& rhs);

只有当lhs.current==rhs.current时，第一个模板操作符才会返回true。只有当lhs和rhs存储相同的流指针时，第二个模板操作符才会返回true。第三个模板操作符返回lhs.equal(rhs)。

**operator<**

template<class RanIt>

bool operator<(

    const reverse\_iterator<RanIt>& lhs,

    const reverse\_iterator<RanIt>& rhs

);

该模板操作符返回rhs.current<lhs.current[sic]。

**operator<=**

template<class RanIt>

bool operator<=(

    const reverse\_iterator<RanIt>& lhs,

    const reverse\_iterator<RanIt>& rhs

);

该模板操作符返回!(rhs<lhs)。

**operator>**

template<class RanIt>

bool operator>(

    const reverse\_iterator<RanIt>& lhs,

    const reverse\_iterator<RanIt>& rhs

);

该模板操作符返回rhs<lhs。

**operator>=**

template<class RanIt>

bool operator>=(

    const reverse\_iterator<RanIt>& lhs,

    const reverse\_iterator<RanIT>& rhs

);

该模板操作符返回!(lhs<rhs)。

**operator+**

template<class RanIt>

reverse\_iterator<RanIt> operator+(Dist n, const reverse\_iterator<RanIt>& rhs);

该模板操作符返回rhs+n。

**operator-**

template<class RanIt>

Dist operator-(

    const reverse\_iterator<RanIt>& lhs,

    const reverse\_iterator<RanIT>& rhs

);

该模板操作符返回rhs.current-lhs.current[sic]。

**ostream\_iterator**

template<class U, class E=char, class T=char\_traits<E>>

class ostream\_iterator

: public iterator<output\_iterator\_tag, void, void, void, void>

{

public:

    typedef U value\_type;

    typedef E char\_type;

    typedef T traits\_type;

    typedef basic\_ostream<E, T> ostream\_type;

    ostream\_iterator(ostream\_type& os);

    ostream\_iterator(ostream\_type& os, cost E\* delim);

    ostream\_iterator<U, E, T>& operator=(const U& val);

    ostream\_iterator<U, E, T>& operator\*();

    ostream\_iterator<U, E, T>& operator++();

    ostream\_iterator<U, E, T>& operator++(int);

};

该模板类似描述了一个输出迭代器对象。它可以将一个类型为U的对象插入到一个输出流中去。可通过它所存储的类型为指向basic\_ostream<E, T>的指针的对象存取该输出流。它同时还存储了一个指向分界字符串的指针，该字符串是一个元素类型为E且以空字符为结束的字符串，在每个插入动作后，它都会被添加到流的末端（注意，该字符串本身并非是由构造函数中复制的）。

**ostream\_iterator::char\_type**

typedef E char\_type;

该类型为模板参数E的同义词。

**ostream\_iterator::operator\***

ostream\_iterator<U, E, T>& operator\*();

该操作符返回\*this。

**ostream\_iterator::operator++**

ostream\_iterator<U, E, T>& operator++();

ostream\_iterator<U, E, T>& operator++(int);

这两个操作符都返回\*this。

**ostream\_iterator::operator=**

ostream\_iterator<U, E, T>& operator=(const U& val);

该操作符先将val插入到关联输出流中，然后返回\*this。

**ostream\_iterator::ostream\_iterator**

ostream\_iterator(ostream\_type& os);

ostream\_iterator(ostream\_type& os, const E\* delim);

第一个构造函数将输出流指针初始化为&os，指向分界符字符串的指针初始化为空字符串。第二个构造函数将输出流指针初始化为&os， 指向分界符字符串的指针初始化为delim。

**ostream\_iterator::ostream\_type**

typedef basic\_ostream<E, T> ostream\_type;

该类型为basic\_ostream<E, T>的同义词。

**ostream\_iterator::traits\_type**

typedef T traits\_type;

该类型为模板参数T的同义词。

**ostream\_iterator::value\_type**

typedef U value\_type;

该类型为模板参数U的同义词。

**ostreambuf\_iterator**

template<class E, class T=char\_traits<E>>

class ostreambuf\_iterator

: public iterator<output\_iterator\_tag, void, void, void, void>

{

public:

    typedef E char\_type;

    typedef T traits\_type;

    typedef basic\_streambuf<E, T> streambuf\_type;

    typedef basic\_ostream<E, T> ostream\_type;

    ostreambuf\_iterator(streambuf\_type\* sb) throw();

    ostreambuf\_iterator(ostream\_type& os) throw();

    ostreambuf\_iterator& operator=(E x);

    ostreambuf\_iterator& operator\*();

    ostreambuf\_iterator& operator++();

    T1 operator++(int);

    bool failed() const throw();

};

该模板类描述了一个输出迭代器对象。它可以将类E的元素插入到输出流缓冲区中。可通过它所描述的类型为指向basic\_streambuf<E, T>的指针的对象存取该输出流缓冲区。

**ostreambuf\_iterator::char\_type**

typedef E char\_type;

该类型是模板参数E的同义词。

**ostreambuf\_iterator::failed**

bool failed() const throw();

只有当前面对输入流缓冲区进行的插入动作失败时，该成员函数才返回true。

**ostreambuf\_iterator::operator\***

ostreambuf\_iterator& operator\*();

该操作符返回\*this。

**ostreambuf\_iterator::operator++**

ostreambuf\_iterator& operator++();

T1 operator++(int);

第一个操作符返回\*this。第二个操作符返回一个类型为T1的对象，其中T1可以转化为ostreambuf\_iterator<E, T>。

**ostreambuf\_iterator::operator=**

ostreambuf\_iterator& operator=(E x);

该操作符先将插入到关联流缓冲区中，然后返回\*this。

**ostreambuf\_iterator::ostream\_type**

typedef basic\_ostream<E, T> ostream\_type;

该类型是basic\_ostream<E, T>的同义词。

**ostreambuf\_iterator::ostreambuf\_iterator**

ostreambuf\_iterator(streambuf\_type\* sb) throw();

ostreambuf\_iterator(ostream\_type& os) throw();

第一个构造函数将输入流缓冲区指针初始化为sb。第二个构造函数将输出流缓冲区指针初始化为os.rdbuf()（存储的指针不能为空指针）。

**ostreambuf\_iterator::streambuf\_type**

typedef basic\_streambuf<E, T> streambuf\_type;

该类型为basic\_streambuf<E, T>的同义词。

**ostreambuf\_iterator::traits\_type**

typedef T traits\_type;

该类型为模板参数T的同义词。

**output\_iterator\_tag**

struct output\_iterator\_tag{};

当It描述的对象可以当作输出迭代器来使用时，该类型等价于

iterator<It>::iterator\_category。

**reverse\_iterator**

template<class RanIt>

class reverse\_iterator: public iterator<

    typename iterator\_traits<RanIt>::iterator\_category,

    typename iterator\_traits<RanIt>::value\_type,

    typename iterator\_traits<RanIt>::difference\_type,

    typename iterator\_traits<RanIt>::pointer,

    typename iterator\_traits<RanIt>::reference>

{

    typedef typename iterator\_traits<RanIt>::difference\_type Dist;

    typedef typename iterator\_traits<RanIt>::pointer Ptr;

    typedef typename iterator\_traits<RanIt>::reference Ref;

public:

    typedef RanIt iterator\_type;

    reverse\_iterator();

    explicit reverse\_iterator(RanIt x);

    template<class U>

    reverse\_iterator(const reverse\_iterator<U>& x);

    RanIt base() const;

    Ref operator\*() const;

    Ptr operator->() const;

    reverse\_iterator& operator++();

    reverse\_iterator operator++(int);

    reverse\_iterator& operator--();

    reverse\_iterator operator--();

    reverse\_iterator& operator+=(Dist n);

    reverse\_iterator operator+(Dist n) const;

    reverse\_iterator& operator-=(Dist n);

    reverse\_iterator operator-(Dist n) const;

    Ref operator[](Dist n) const;

protected:

    RanIt current;

};

该模板类描述了一个各方面动作都像（只是顺序相反）随机存取迭代器的对象。它在保护型对象current中存储了一个类型为RanIt的随机存取迭代器。对类型为reverse\_iterator的对象x进行递增其实是在对x.current进行递减，反之亦然。此外，表达式\*x其实代表的是\*(current-1)，且其结果的类型为Ref。Ref通常都是类型T&。

也就是说，我们可以使用一个类reverse\_iterator的对象来对一个随机存取迭代器在其中来回移动的序列进行逆向存取。

不少STL中的容器特别为RanIt是一个双向迭代器这种情况对reverse\_iterator进行了特化。在这种情况下，我们就不能再调用operator+=、operator=、operator-=、operator-以及operator[]这些成员函数了。

**reverse\_iterator::base**

RanIt base() const;

该成员函数返回current。

**reverse\_iterator::iterator\_type**

typedef RanIt iterator\_type;

该类型为模板参数RanIt的同义词。

**reverse\_iterator::operator\***

Ref operator\*() const;

该操作符返回\*(current-1)。

**reverse\_iterator::operator+**

reverse\_iterator operator+(Dist n) const;

该操作符返回reverse\_iterator(\*this)+=n。

**reverse\_iterator::operator++**

reverse\_iterator& operator++();

reverse\_iterator& operator++(int);

第一个（前递增）操作符先计算--current，然后返回\*this。

第二个（后递增）操作符先对\*this进行复制，然后计算--current，最后返回那份对\*this的拷贝。

**reverse\_iterator::operator+=**

reverse\_iterator& operator+=(Dist n);

该操作符先计算current-n，然后返回\*this。

**reverse\_iterator::operator-**

reverse\_iterator operator-(Dist n) const;

该操作符返回reverse\_iterator(\*this)-=n。

**reverse\_iterator::operator--**

reverse\_iterator& operator--();

reverse\_iterator operator--();

第一个（前递减）操作符先计算++current，然后返回\*this。

第二个（后递减）操作符先对\*this进行复制，然后计算++current，最后返回那份对\*this的拷贝。

**reverse\_iterator::operator->**

Ptr operator->() const;

该操作符返回\*this。

**reverse\_iterator::operator[]**

Ref operator[](Dist n) const;

该操作符返回\*(\*this+n)。

**reverse\_iterator::pointer**

typedef Ptr Pointer;

该类型为模板参数Ptr的同义词。

**reverse\_iterator::reference**

typedef Ref reference;

该类型为模板参数Ref的同义词。

**reverse\_iterator::reverse\_iterator**

reverse\_iterator();

explicit reverse\_iterator(RanIt x);

template<class U>

reverse\_iterator(const reverse\_iterator<U>& x);

第一个构造函数使用默认构造函数对current进行初始化。第二个构造函数将current初始化为x.current。

模板构造函数将current初始化为x.base()。

## 4.3 使用<iterator>

如果我们在翻译单元中包含任何STL头文件，那么<iterator>就很可能会被一同包含入该翻译单元中。只有需要在程序中使用该头文件中的单元并且不指望通过其他STL头文件将这些单元带入翻译单元时，才由必要在程序中显式地包含它。但是我们也很有可能需要在程序中使用该头文件中所定义的实体。如果在翻译过程中我们得到了这样的诊断信息：reverse\_iterator未定义，请记住：最简单的解决方案就是将这个头文件包含到程序中。

下面是对于<iterator>中的各种对于的简单介绍：

**迭代器**

迭代器可以是对象这种或我们所定义的类。如果我们定义了自己的迭代器类，就必须同时为其定义一些成员类型。最简单的方法就是：在可能的情况下，从一个已有的迭代器类公共性地派生处我们自己的迭代器类。否则的话，请按照下面的几种模式中的一种进行：

class OutIt

: public iterator<output\_iterator\_tag, void, void, void, void>

{

    //......

}; // for an output iterator

class InIt

: public iterator<input\_iterator\_tag, T, Dist>

{

    //......

}; // for an input iterator

class PwdIt

: public iterator<forward\_iterator\_tag, T, Dist>

{

    //......

}; // for a forward iterator

class BidIt

: public iterator<bidirectional\_iterator\_tag, T, Dist>

{

    //......

}; // for a bidirectional iterator

class RanIt

: public iterator<random\_access\_iterator\_tag, T, Dist>

{

    //......

}; // for a random\_access iterator

在此处，T是迭代器的元素类型，Dist是其差距类型。对于类型为T\*的对象指针来说，种类为random\_access\_iterator\_tag，元素类型当然为T， 差距类型为ptrdiff\_t。

我们可以通过两种方法中的一种来操作类型为Iter的迭代器first和last。在支持部分特化的实现中，我们可以这么写：

**iterator\_traits**

iterator\_traits<Iter>::iterator\_category

// the iterator category type

iterator\_traits<Iter>::value\_type

// the element type

iterator\_traits<Iter>::distance\_type

// the distance type

**advance**

advance(first, N)

// adds N to X

**distance**

N = distance(first, last)

// returns the distance from first to last

如果实现不支持部分特化，或是不能确信是否支持它，那么本书中提供的STL版本可以让我们这么写：

**Iter\_cat**

Iter\_cat(first)

// returns an object of the iterator category type

**val\_type**

val\_type(first)

//returns a null pointer to the element type

**Dist\_type**

Dist\_type(first)

// returns a null pointer to the distance type

**advance**

advance(first, N)

// adds N to X

**Distance**

Distance(first, last , N)

// adds to N the distance from first to last

在上面提到的所有情况中，迭代器种类始终为一个空结构类型。

**迭代器标签**

output\_iterator\_tag

input\_iterator\_tag

forward\_iterator\_tag

bidirenctional\_iterator\_tag

random\_access\_iterator\_tag

其中，后三个标签都是从其上面的某个标签公共地派生出来的。

**reverse\_iterator**

对于给定的一个类型为RanIt的随机存取迭代器，我们可以定义一个与之相关的反转型迭代器，并用其来逆向存取序列。

typedef reverse\_iterator<RanIt> RevIt;

为了在指定范围[first, last)中逆向存取由迭代器RanIt指定的序列，我们可以这么写：

RevIt rfirst(last), rlast(first);

for(; rfisrt != rlast; ++rfirst)

<process>(rfirst);

对于类型为BitIt的双向迭代器，我们也有可能像上面一样使用reverse\_iterator。只是需要确保把得到的反转型迭代器看成是一个双向迭代器，那些仅为随机存取迭代器所定义的操作可能会导致编译器产生一些难以理解的诊断信息。但是，如果给定的实现不支持这种用法（或是我们不能确信是否支持它）的话，那么本书中提供的STL版本可以让我们这么写：

**Revbidit**

typedef Revbidit<BidIt> RevBidIt;

RevBidIt rfirst(last), rlast(first);

for(; rfirst != rlast; ++rfirst)

    <process>(rfirst);

**插入型迭代器**

我们使用插入型迭代器来将元素插入到容器中，并使之看起来好像在用一个输出迭代器来此处产生的序列。

**back\_insert\_iterator**

给定一个类型为Cont的容器，我们可以定义一个输出迭代器OutIt，用其来向被控序列末端插入元素：

typedef back\_insert\_iterator<Cont> OutIt;

这样，我们就可以使用模板函数copy（参见第6章）将由区间(first, last)中的迭代器指定的序列添加到容器c的末端：

OutIt it(c);

copy(first, last, it);

**back\_inserter**

或者我们也可以实时产生所需的迭代器，用它来把序列添加到容器的末端：

copy(first, last, back\_inserter(c));

该迭代器提供调用Cont::push\_back(const T&)来完成插入动作。只有在能够有效地完成这个操作的前提下，STL容器才会定义这个成员函数。

**front\_insert\_iterator**

我们同样可以定义一个输出迭代器OutIt，用其来向被控序列前端以逆序的方式插入元素：

typedef front\_insert\_iterator<Cont> Outit;

为了以逆序方式将一个序列插入到容器的前端，我们可以这么写：

OutIt it(c);

copy(first, last, it);

**front\_inserter**

或者，我们可以实时产生所需的输出迭代器，并用它来把反转的序列添加到容器的前端：

copy(first, last, front\_inserter(c));

该迭代器提供调用Cont::push\_front(const T&)来完成插入动作。只有在能够有效地完成这个操作的前提下，STL容易才会定义这个成员函数。

insert\_iterator

最后，我们定义了一个输出迭代器OutIt，并用它来向被控序列中指定的位置前面插入元素：

typedef insert\_iterator<Cont> OutIt;

为了插入一个序列，我们以一个迭代器来指定插入点。例如，为了将一个序列插入到容器的前端且不反转该序列，我们可以这么写：

OutIt it(c, c.begin());

copy(first, last, it);

对所有STL容器来说，成员函数begin()要么返回指向被控序列中的第一个元素的迭代器，要么就是指向紧接着一个空序列的末端的下一个位置的迭代器。在这两种情况下，它都定义了我们想要将被复制的序列插入到其前面的插入点。

inserter

为了实时产生处我们所需的输出迭代器，并用它来把序列插入到容器的开始处，我们可以这么写：

copy(first, last, inserter(c, c.begin()));

该迭代器提供调用Cont::insert(Cont::iterator, const T&)来完成插入动作。该成员函数返回一个指向刚刚插入的那元素的迭代器。不管其开销如何，每个STL容器都定义了这个成员函数。

**流迭代器**

有时，在输入流中会包含一系列的字段(field)，而对于每个字段，我们都希望将其提取出来并作为某种类型T的对象。操作这类流的一种常见方法就是向其写入，例如：

T x;

while(cin>>x)

    <process>(x);

当从标准输入中进行提取的动作失败时（通常是在文件结束处），上述循环就会终止。

**istream\_iterator**

通过引入一个类istream\_iterator的对象，我们就可以以一种看上去像是用输入迭代器来遍历一个序列的方式来完成与上面相同的逻辑：

typedef istream\_iterator<T, char, char\_traits<char>> InIt;

这种特殊的类型对于使得InIt可以与一个类istream的对象(如cin)配合使用。模板类char\_traits是一种更为一般化的机制的一部分，用来指定输入和输出流（在C++标准中此有介绍）。通过使用默认的模板参数，可以或略掉模板参数列表中除去第一个参数意外的其他所有参数，把它写成istream\_iterator<T>。

给定上面提供的类型对于，我们就可以这么写：

InIt first(cin), last;

while(first != last)

    <process>(\*first++);

在上面提供的代码里面，迭代器完成了我们所需的提取动作。然而，请不要试图通过first以外的其他方法来存取该输入流。不要直接从流中读取，也不要创建另一个迭代器以从中读取。每个迭代器都必须不时地向前读取所需的对象，这使得输入流很少会保持一种很容易描述的状态（即便first被销毁后也是如此）。

有时我们可能希望向一个输出流中插入一个由某种类型T的对象构成的序列。产生这样的流的一种常见的方法就是向其写入，例如：

while(<not done>)

{

    T x=<next value>;

    cout<<x;

}

ostream\_iterator

通过引入一个类ostream\_iterator的对象，我们就可以以一种看上去像是用一个输出迭代器来此处生成序列的方式来完成与上面相同的逻辑：

typedef ostream\_iterator<T, char, char\_traits<char>> OutIt;

这种特殊的类型对于使得OutIt可以与一个类ostream的对象(如cout)配合使用。通过使用默认的模板参数，可以忽略掉模板参数列表中除去第一个参数以外的其他所有参数，把它写成ostream\_iterator<T>。

于是，我们可以这么写：

OutIt next(cout);

while(<not done>){

    T x=<next value>;

    \*next++ = x;

}

如果想在每次插入动作后添加一个常见的文本串，可以把上面的代码稍做改动，这么写：

OutIt next(cout, “\n”);

while(<not done>){

    T x=<next value>;

    \*next++ = x;

}

在这种情况下，每个被插入的条目后面都紧接着一个换行符。注意，我们在构造函数中所指定的那个以空字符结束的字符串并没有被复制。我们必须确保该字符串的生命周期要长于迭代器的生命周期。

附带说一句，通过next和其他机制来向输出流中写入是安全的。迭代器对其产生的输出字段没有做任何缓冲。

**流缓冲迭代器**

我们可以以一种看上去像是用输入迭代器来存取序列中的于是的方式来从输入流中提t取单个于是。为了这样操作标准输入流，我们可以这么写：

typedef istreambuf\_iterator<char, char\_traits<char>> InIt;

这种特殊的类型定义使得InIt可以与一个类istream的对象(如cin)配合使用。通过使用默认的模板参数，可以忽略掉模板参数列表中除去第一个参数以外的其他所有参数，把它写成istreambuf\_iterator<char>。

于是，我们就可以这样声明：

InIt first(cin), last;

如果first!=last，表达式\*first就会产生可以从标准输入流中得到的下一个字符（于是类型为char）。迭代器实际上存储了一个指针p，p指向的是底层的流缓冲区（在这种情况下，它的类型为streambuf）。我们也可以将first声明为：

InIt first(cin>>rdbuf());

在上面提供的代码里面，迭代器完成了我们所需的提取动作。它实质上是通过调用p->sgetc()或者p->sbumpc()类完成提取动作的。不要试图通过first以外的其他方法来存取该输入流。迭代器必须向前读取，这使得输入流很少会保持一种容易描述的状态。不要直接从流中读取，也不要创建两一个迭代器以从中读取。然而，一旦first被销毁，我们就可以确信：第一个没有被读取的元素就是流缓冲区中的下一个可以被读取的元素。简单地说，在任何时间内，输入流都应该拥有不超过一个的与之相关的类istreambuf\_iterator的对象。当存在这样的对象时，它就应该是从输入流中提取元素的唯一来源。

**ostreambuf\_iterator**

我们可以以一种看上去像是用输出迭代器来存储生成序列的方式来向输出流插入单个元素。为了这样操作标准输出流，我们可以这样写：

typedef ostreambuf\_iterator<char, char\_traits<char>> OutIt;

这种特殊的类型定义使得OutIt可以与一个类ostream的对象(如cout)配合使用。通过使用默认的模板参数，可以忽略掉模板参数列表中除去第一个参数以外的其他所有参数，把它写成ostreambuf\_iterator<char>。于是，我们就可以这样声明：

OutIt next(cout);

表达式\*next++=c将字符c(类型为char)插入到标准输出流中。实际上，迭代器存储了一个指针p，p指向的是底层的流缓冲区（在此例中是类streambuf）。我们也可以将next声明为：

OutIt next(cout->rdbuf());

迭代器通过调用p->sputc(c)来完成我们所需的插入动作。

附带说一句，通过next和其他机制来向输出流中写入是安全的。迭代器对其产生的输出字段没有做任何缓冲。

## 4.4 实现<iterator>

本书所提供的实现将一部分名义上是定义在头文件<iterator>中的定义移动到其他头文件中。新增的头文件<xutility>（它不是C++标准中所指定的头文件）包含一系列定义，它们贯穿使用于一个典型的标准C++库中，甚至在没有显式包含<iterator>的程序中也可以使用。只要程序中包含<iterator>就一定会包含<xutility>，我们的实现就可以自由地进行这样的定义重组。实际上，我们也正是这么做的。

实现通常会因为一些不同的原因而导入一些“内部的”头文件，如<xutility>等。标准C++库中包含不少循环依赖关系——如类basic\_string的典型与类exception的典型之间就存在这样的循环依赖关系（类basic\_string的典型可以抛出异常），而类exception的典型可以由类basic\_string的典型构造。库同样也包含一些比较大的头文件（如<algorithm>和<iterator>），单可能在其他的头文件中，我们只需要它们中的部分定义。在这两种情况下，将部分定义移到另一个新增的内部头文件中将会使实现者的任务更容易处理。

**<xutility>**

程序清单4-1列出了文件xutility。它包含了那些广泛使用的<iterator>和<algorithm>中的定义。我们将会在第6章中讨论它的作用。xutility首先定义了迭代器标签，它们只是一些简单的空类。迭代器标签本身没有存储任何值，它们只是作为模板参数，提供它们的类型和类型层次来传递信息。

**程序清单4-1：xutility**

// xutility internal header

#ifndef XUTILITY\_

#define XUTILITY\_

#include<utility>

//#include<streambuf>

namespace std{

    // ITERATOR STUFF (from <iterator>)

    //ITERATOR TAGS

    struct input\_iterator\_tag{};

    struct output\_iterator\_tag{};

    struct forward\_iterator\_tag: public input\_iterator\_tag{};

    struct bidirectional\_iterator\_tag

        : public forward\_iterator\_tag{};

    struct random\_access\_iterator\_tag

        : public bidirectional\_iterator\_tag{};

    struct Int\_iterator\_tag{};

    // TEMPLATE CLASS iterator

    template<

        class C, class T, class D=ptrdiff\_t, class Pt=T\*, class Rt=T&>

    struct iterator{

        typedef C iterator\_category;

        typedef T value\_type;

        typedef D difference\_type;

        typedef Pt pointer;

        typedef Rt reference;

    };

    template<class T, class D, class Pt, class Rt>

    struct Bidit: public iterator<bidirectional\_iterator\_tag, T, D, Pt, Rt>

    {

    };

    template<class T, class D, class Pt, class Rt>

    struct Ranit: public iterator<random\_access\_iterator\_tag, T, D, Pt, Rt>

    {

    };

    struct Outit: public iterator<output\_iterator\_tag, void, void, void, void>

    {

    };

    // TEMPLATE CLASS iterator\_traits

    template<class It>

    struct iterator\_traits{

        typedef typename It::iterator\_category iterator\_category;

        typedef typename It::value\_type value\_type;

        typedef typename It::difference\_type difference\_type;

        typedef typename It::pointer pointer;

        typedef typename It::reference reference;

    };

    template<class T>

    struct iterator\_traits<T\*>{

        typedef random\_access\_iterator\_tag iterator\_category;

        typedef T value\_type;

        typedef ptrdiff\_t difference\_type;

        typedef T\* pointer;

        typedef T& reference;

    };

    template<class T>

    struct iterator\_traits<const T\*>{

        typedef random\_access\_iterator\_tag iterator\_category;

        typedef T value\_type;

        typedef ptrdiff\_t difference\_type;

        typedef const T \*pointer;

        typedef const T& reference;

    };

    // TEMPLATE FUNCTION Iter\_cat

    template<class C, class T, class D, class Pt, class Rt>

    inline C Iter\_cat(const iterator<C, T, D, Pt, Rt>&)

    {

        C X;

        return X;

    }

    template<class T>

    inline random\_access\_iterator\_tag Iter\_cat(const T\*)

    {

        random\_access\_iterator\_tag X;

        return X;

    }

    // INTEGER FUNCTION Iter\_cat

    inline Int\_iterator\_tag Iter\_cat(bool)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(char)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(signed char)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(unsigned char)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(wchar\_t)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(short)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(unsigned short)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(int)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(unsigned int)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(long)

    {

        Int\_iterator\_tag X;

        return X;

    }

    inline Int\_iterator\_tag Iter\_cat(unsigned long)

    {

        Int\_iterator\_tag X;

        return X;

    }

    // TEMPLATE FUNCTION Distance

    template<class InIt>

    inline typename iterator\_traits<InIt>::difference\_type

        Distance(InIt F, InIt L)

    {

        typename iterator\_traits<InIt>::difference\_type N=0;

        Distance2(F, L, N, Iter\_cat(F));

        return N;

    }

    template<class InIt, class D>

    inline void Distance(InIt F, InIt L, D& N)

    {

        Distance2(F, L, N, Iter\_cat(F));

    }

    template<class InIt, class D>

    inline void Distance2(InIt F, InIt L, D& N, input\_iterator\_tag)

    {

        for(; F != L; ++F)

            ++N;

    }

    template<class InIt, class D>

    inline void Distance2(InIt F, InIt L, D& N, forward\_iterator\_tag)

    {

        for(; F != L; ++F)

            ++N;

    }

    template<class InIt, class D>

    inline void Distance2(InIt F, InIt L, D& N, bidirectional\_iterator\_tag)

    {

        for(; F != L; ++F)

            ++N;

    }

    template<class RanIt, class D>

    inline void Distance2(RanIt F, RanIt L, D& N, random\_access\_iterator\_tag)

    {

        N += L - F;

    }

    // TEMPLATE CLASS Ptrit

    template<class T, class D, class Pt, class Rt, class Pt2, class Rt2>

    class Ptrit: public iterator<random\_access\_iterator\_tag, T, D, Pt, Rt>

    {

    public:

        typedef Ptrit<T, D, Pt, Rt, Pt2, Rt2> Myt;

        Ptrit(){}

        explicit Ptrit(Pt P): current(P){}

        Ptrit(const Ptrit<T, D, Pt2, Rt2, Pt2, Rt2>& X): current(X.base()){}

        Pt base() const

        {

            return current;

        }

        Rt operator\*() const

        {

            return \*current;

        }

        Pt operator->() const

        {

            return &\*\*this;

        }

        Myt& operator++()

        {

            ++current;

            return \*this;

        }

        Myt& operator++(int)

        {

            Myt tmp = \*this;

            ++current;

            return tmp;

        }

        Myt& operator--()

        {

            --current;

            return \*this;

        }

        Myt& operator--(int)

        {

            Myt tmp = \*this;

            --current;

            return tmp;

        }

        bool operator==(int Y) const

        {

            return current==(Pt)Y;

        }

        bool operator==(const Myt& Y) const

        {

            return current==Y.current;

        }

        bool operator!=(const Myt& Y) const

        {

            return !(\*this==Y);

        }

        Myt& operator+=(D N)

        {

            current += N;

            return \*this;

        }

        Myt& operator+(D N) const

        {

            return Myt(current+N);

        }

        Myt& operator-=(D N)

        {

            current -= N;

            return \*this;

        }

        Myt& operator-(D N) const

        {

            return Myt(current-N);

        }

        Rt operator[](D N) const

        {

            return \*(\*this+N);

        }

        bool operator<(const Myt& Y) const

        {

            return current<Y.current;

        }

        bool operator>(const Myt& Y) const

        {

            return Y<\*this;

        }

        bool operator<=(const Myt& Y) const

        {

            return !(Y<\*this);

        }

        bool operator>=(const Myt& Y) const

        {

            return !(\*this<Y);

        }

        D operator-(const Myt& Y) const

        {

            return current-Y.current;

        }

    protected:

        Pt current;

    };

    template<class T, class D, class Pt, class Rt, class Pt2, class Rt2>

    inline Ptrit<T, D, Pt, Rt, Pt2, Rt2> operator+(

        D N, const Ptrit<T, D, Pt, Rt, Pt2, Rt2>& Y)

    {

        return Y+N;

    }

    // TEMPLATE CLASS reverse\_iterator

    template<class RanIt>

    class reverse\_iterator: public iterator<

        typename iterator\_traits<RanIt>::iterator\_category,

        typename iterator\_traits<RanIt>::value\_type,

        typename iterator\_traits<RanIt>::difference\_type,

        typename iterator\_traits<RanIt>::pointer,

        typename iterator\_traits<RanIt>::reference>

    {

    public:

        typedef reverse\_iterator<RanIt> Myt;

        typedef typename iterator\_traits<RanIt>::difference\_type D;

        typedef typename iterator\_traits<RanIt>::pointer Pt;

        typedef typename iterator\_traits<RanIt>::reference Rt;

        typedef RanIt iterator\_type;

        reverse\_iterator(){}

        explicit reverse\_iterator(RanIt X): current(X){}

        template<class U>

        reverse\_iterator(const reverse\_iterator<U>& X): current(X.base()){}

        RanIt base() const

        {

            return current;

        }

        Rt operator\*() const

        {

            RanIt tmp=current;

            return \*--tmp;

        }

        Pt operator->() const

        {

            return &\*\*this;

        }

        Myt& operator++()

        {

            --current;

            return \*this;

        }

        Myt& operator++(int)

        {

            Myt tmp=\*this;

            --current;

            return tmp;

        }

        Myt& operator--()

        {

            ++current;

            return \*this;

        }

        Myt& operator--(int)

        {

            Myt tmp=\*this;

            ++current;

            return tmp;

        }

        bool Eq(const Myt& Y) const

        {

            return current==Y.current;

        }

        // random-access only beyond this point

        Myt& operator+=(D N)

        {

            current -= N;

            return \*this;

        }

        Myt& operator+(D N) const

        {

            current += N;

            return \*this;

        }

        Myt& operator-=(D N)

        {

            current -= N;

            return \*this;

        }

        Myt& operator-(D N) const

        {

            return Myt(current+N);

        }

        Rt operator[](D N) const

        {

            return \*(\*this+N);

        }

        bool Lt(const Myt& Y) const

        {

            return Y.current<current;

        }

        D Mi(const Myt& Y) const

        {

            return Y.current-current;

        }

    protected:

        RanIt current;

    };

    // reverse\_iterator TEMPLATE OPERATORS

    template<class RanIt, class D>

    inline reverse\_iterator<RanIt> operator+(

        D N, const reverse\_iterator<RanIt>& Y)

    {

        return Y+N;

    }

    template<class RanIt>

    inline typename reverse\_iterator<RanIt>::D operator-(

        const reverse\_iterator<RanIt>& X,

        const reverse\_iterator<RanIt>& Y)

    {

        return X.Mi(Y);

    }

    template<class RanIt>

    inline bool operator==(

        const reverse\_iterator<RanIt>&X,

        const reverse\_iterator<RanIt>& Y)

    {

        return X.Eq(Y);

    }

    template<class RanIt>

    inline bool operator!=(

        const reverse\_iterator<RanIt>& X,

        const reverse\_iterator<RanIt>& Y)

    {

        return !(X==Y);

    }

    template<class RanIt>

    inline bool operator<(

        const reverse\_iterator<RanIt>& X,

        const reverse\_iterator<RanIt>& Y)

    {

        return X.Lt(Y);

    }

    template<class RanIt>

    inline bool operator>(

        const reverse\_iterator<RanIt>& X,

        const reverse\_iterator<RanIt>& Y)

    {

        return Y<X;

    }

    template<class RanIt>

    inline bool operator<=(

        const reverse\_iterator<RanIt>& X,

        const reverse\_iterator<RanIt>& Y)

    {

        return !(Y<X);

    }

    template<class RanIt>

    inline bool operator>=(

        const reverse\_iterator<RanIt>& X,

        const reverse\_iterator<RanIt>& Y)

    {

        return !(X<Y);

    }

    template<class BidIt>

    class Revbidit: public iterator<

        typename iterator\_traits<BidIt>::iterator\_category,

        typename iterator\_traits<BidIt>::value\_type,

        typename iterator\_traits<BidIt>::difference\_type,

        typename iterator\_traits<BidIt>::pointer,

        typename iterator\_traits<BidIt>::reference>

    {

    public:

        typedef Revbidit<BidIt> Myt;

        typedef typename iterator\_traits<BidIt>::difference\_type D;

        typedef typename iterator\_traits<BidIt>::pointer Pt;

        typedef typename iterator\_traits<BidIt>::reference Rt;

        typedef BidIt iterator\_type;

        Revbidit(){ }

        explicit Revbidit(BidIt X): current(X) { }

        BidIt base() const{

            return current;

        }

        Rt operator\*() const

        {

            BidIt tmp=current;

            return \*--tmp;

        }

        Pt operator->() const

        {

            Rt tmp = \*\*this;

            return &tmp;

        }

        Myt& operator++()

        {

            --current;

            return \*this;

        }

        Myt& operator++(int)

        {

            Myt tmp = \*this;

            --current;

            return tmp;

        }

        Myt& operator--()

        {

            ++current;

            return \*this;

        }

        Myt& operator--(int)

        {

            Myt tmp = \*this;

            ++current;

        return tmp;

        }

        bool operator==(const Myt& Y) const

        {

            return current == Y.current;

        }

        bool operator!=(const Myt& Y) const

        {

            return !(&this==Y);

        }

    protected:

        BidIt current;

    };

    // TEMPLATE CLASS istreambuf\_iterator

    template<class E, class Tr>

    class istreambuf\_iterator

        : public iterator<input\_iterator\_tag, E, typename Tr::off\_type, E\*, E&>

    {

    public:

        typedef istreambuf\_iterator<E, Tr> Myt;

        typedef E char\_type;

        typedef Tr traits\_type;

        typedef basic\_streambuf<E, Tr> streambuf\_type;

        typedef basic\_istream<E, Tr> istream\_type;

        typedef typename traits\_type::int\_type int\_type;

        istreambuf\_iterator(streambuf\_type \*Sb=0) throw()

            : Sbuf(Sb), Got(Sb == 0) { }

        istreambuf\_iterator(istream\_type& I) throw()

            : Sbuf(I.rdbuf()), Got(I.rdbuf() == 0) { }

        const E& operator\*() const

        {

            if(!Got)

                ((Myt\*)this)->Peek();

            return Val;

        }

        const E \*operator->() const

        {

            return &\*\*this;

        }

        Myt& operator++()

        {

            Inc();

            return \*this;

        }

        Myt operator++(int)

        {

            if(!Got)

                Peek();

            Myt tmp = \*this;

            Inc();

            return tmp;

        }

        bool equal(const Myt& X) const

        {

            if(!Got)

                ((Myt\*)this)->Peek();

            if(!X.Got)

                ((Myt\*)&X)->Peek();

            return Sbuf == 0 && X.Sbuf == 0 || Sbuf != 0 && X.Sbuf != 0;

        }

    private:

        void Inc()

        {

            if(Sbuf == 0 || traits\_type::eq\_int\_type(traits\_type::eof(), Sbuf->sbumpc()))

                Sbuf = 0, Got = true;

            else

                Got = false;

        }

        E Peek()

        {

            int\_type C;

            if(Sbuf == 0 || traits\_type::eq\_int\_type(traits\_type::eof(), C=Sbuf->sgetc()))

                Sbuf = 0;

            else

                Val = traits\_type::to\_char\_type(C);

            Got = true;

            return Val;

        }

        streambuf\_type \*Sbuf;

        bool Got;

        E Val;

    };

    // istreambuf\_iterator TEMPLATE OPERATORS

    template<class E, class Tr>

    inline bool operator==(

        const istreambuf\_iterator<E, Tr>& X,

        const istreambuf\_iterator<E, Tr>& Y)

    {

        return X.equal(Y);

    }

    template<class E, class Tr>

    inline bool operator!=(

        const istreambuf\_iterator<E, Tr>& X,

        const istreambuf\_iterator<E, Tr>& Y)

    {

        return !(X==Y);

    }

    // TEMPLATE CLASS ostreambuf\_iterator

    template<class E, class Tr>

    class ostreambuf\_iterator: public Outit

    {

    private:

        bool Failed;

        streambuf\_type \*Sbuf;

    public:

        typedef ostreambuf\_iterator<E, Tr> Myt;

        typedef E char\_type;

        typedef Tr traits\_type;

        typedef basic\_streambuf<E, Tr> streambuf\_type;

        typedef basic\_streambuf<E, Tr> ostream\_type;

        ostreambuf\_iterator(streambuf\_type \*sb):Failed(false), Sbuf(sb)

        {

        }

        ostreambuf\_iterator(ostream\_type& O):Failed(false), Sbuf(O.rdbuf())

        {

        }

        Myt& operator=(E X)

        {

            if(Sbuf==0 || traits\_type::eq\_int\_type(Tr::eof(), Sbuf->sputc(X)))

                Failed = true;

            return \*this;

        }

        Myt& operator\*()

        {

            return \*this;

        }

        Myt& operator++()

        {

            return \*this;

        }

        Myt& operator++(int)

        {

            return \*this;

        }

        bool failed() const throw()

        {

            return Failed;

        }

    };

    // AIGORITHM STUFF (from <algorithm>)

    // TEMPLATE FUNCTION copy

    template<class InIt, class OutIt>

    inline OutIt copy(InIt F, InIt L, OutIt X)

    {

        for(; F != L; ++X, ++F)

            \*X = \*F;

        return X;

    }

    // TEMPLATE FUNCTION copy\_backward

    template<class BidIt1, class BidIt2>

    inline BidIt2 copy\_backward(BidIt1 F, BidIt1 L, BidIt2 X)

    {

        while(F!=L)

            \*--X = \*--L;

        return X;

    }

    // TEMPLATE FUNCTION equal

    template<class InIt1, class InIt2>

    inline bool equal(InIt1 F, InIt1 L, InIt2 X)

    {

        return mismatch(F, L, X).first == L;

    }

    // TEMPLATE FUNCTION equal WITH PRED

    template<class InIt1, class InIt2, class Pr>

    inline bool equal(InIt1 F, InIt1 L, InIt2 X, Pr P)

    {

        return mismatech(F, L, X, P).first == L;

    }

    // TEMPLATE FUNCTION fill

    template<class FwdIt, class T>

    inline void fill(FwdIt F, FwdIt L, const T& X)

    {

        for(; F != L; ++F)

            \*F = X;

    }

    // TEMPLATE FUNCTION fill\_n

    template<class OutIt, class Sz, class T>

    inline void fill\_n(OutIt F, Sz N, const T& X)

    {

        for(; 0<N; --N, ++F)

            \*F = X;

    }

    // TEMPLATE FUNCTION lexicographical\_compare

    template<class InIt1, class InIt2>

    inline bool lexicographical\_compare(

        InIt1 F1, InIt1 L1, InIt2 F2, InIt2 L2)

    {

        for(; F1 != L1 && F2 != L2; ++F1, ++F2)

            if(\*F1 < \*F2)

                return true;

            else if(\*F2 < \*F1)

                return false;

        return F1 == L1 && F2 != L2;

    }

    // TEMPLATE FUNCTOIN lexicographical\_compare WITH PRED

    template<class InIt1, class InIt2, class Pr>

    inline bool lexicographical\_compare(

        InIt1 F1, InIt1 L1, InIt2 F2, InIt2 L2, Pr P)

    {

        for(; F1 != L1 && F2 != L2; ++F1, ++F2)

            if(P(\*F1, \*F2))

                return true;

            else if(P(\*F2, \*F1))

                return false;

            return F1 == L1 && F2 != L2;

    }

    // TEMPLATE FUNCTION max

    template<class T>

    inline const T& max(const T& X, const T& Y)

    {

        return X<Y?Y:X;

    }

    // TEMPLATE FUNCTION min WITH PRED

    template<class T, class Pr>

    inline const T& min(const T& X, const T& Y, Pr P)

    {

        return P(X, Y)?Y:X;

    }

    // TEMPLATE FUNCTION mismatch

    template<class InIt1, class InIt2>

    inline pair<InIt1, InIt2> mismatch(InIt1 F, InIt1 L, InIt2 X)

    {

        for(; F!=L && \*F==\*X; ++F, ++X)

            ;

        return pair<InIt1, InIt2>(F, X);

    }

    // TEMPLATE FUNCTION mismatch WITH PRED

    template<class InIt1, class InIt2, class Pr>

    inline pair<InIt1, InIt2>mismatch(InIt1 F, InIt1 L, InIt2 X, Pr P)

    {

        for(; F!=L && P(\*F, \*X); ++F, ++X)

            ;

        return pair<InIt1, InIt2>(F, X);

    }

    // TEMPLATE FUNCTION swap

    template<class T>

    inline void swap(T& X, T& Y)

    {

        T tmp = X;

        X = Y;

        Y = tmp;

    }

}; // namespace std

#endif // XUTILITY

**Int\_iterator\_tag**

注意，到目前为止，我们对新增的标签Int\_iterator\_tag还没有进行任何讨论。它用来标注一个具有integer类型的模板参数，这一点就如同input\_iterator\_tag用来标注一个可以当作输入迭代器来使用的模板参数一样。Int\_iterator\_tag和其他迭代器标签一样，被各种Iter\_cat重载版本所使用，这一点将在下面讨论。它在实现某些模板容器类时很有用，解决了某些成员模板很少的重载版本之间存在的歧义。

**iterator**

我们在这个头文件中定义了模板类iterator，同时，我们还定义了另外两个派生自iterator的模板类：Bidit用于描述双向迭代器，Ranit用于描述随机存取迭代器。这些模板有助于避免在一些容器模板类中经常会出现的命名问题，这时(如果没有定义这两个派生类的话)，由于它们都是代指iterator，就会出现该名字重复定义的问题。与之对应的是，类Outit仅仅是一个缩写以定义贯穿使用于STL头文件中的输出迭代器。

模板类iterator\_traits是最近才引入到STL中的一个机制，它用来确定迭代器的属性。对于这个机制来说，对指针和常量指针的那两个部分特化是它不可或缺的一部分。对于那些不支持部分特化的实现来说，模板函数Iter\_cat、Val\_type以及Dist\_type也能够提供与上面近似一致的信息。这三个模板函数中的任意一个都拥有两个版本：一个用来接受某种形式的迭代器作为参数，另一个则把指针作为参数使用。Iter\_cat甚至还对每个基本的integer类型都有一个重载的版本，所有这些重载的Iter\_cat函数都返回一个类Int\_iterator\_tag的对象，用以标识传入的迭代器参数实际上是一个整数。通过使用这三个模板函数，我们就可以用重载函数的解析机制来弥补缺乏部分特化的遗憾了。

**distance**

模板函数distance是一个标准的机制，用来确定同一范围中两个迭代器之间的差距。再次重申一次，我们同样还保留了一个旧的、不依赖于iterator\_traits(以及部分特化)的机制来和对象指针配合工作。模板函数Distance通过计算两个迭代器之间的差距来增加其第三个参数的值，而不是将这个差距的值计算出来并且直接作为该函数的返回值。这两个模板函数最终都是通过调用模板函数Distance2完成实际差距的计算。这样做虽然简单，但对于不同种类的迭代器，它仍然可以有独到的行为。在此处我们给出了一个非常经典的例子，它使用了标签重载以在一个算法的不同实现版本之间选择最适合的。

这个例子同时也展现了当前某些C++翻译器存在的实际限制。对于forward\_iterator\_tag以及bidirectional\_iterator\_tag的重载其实并非是必要的。我们之所以从input\_iterator \_tag中派生出这两个标签，主要是为了避免在后续的工作中再回到前面做这件事情，我们给出的例子就属于这种情况。但是我们还是发现，有些翻译器面对这种“艰巨”的情况，不能正确地解析重载。因此为了一些纯粹的实际需求，我们在此引入了冗余的定义以使得代码更加易于移植。

**Ptrit**

模板类Ptrit是我们对编译器限制条件的又一个让步。它将对象指针用一个派生自iterator的类封装起来，该iterator拥有随机存取迭代器所需的全部属性。你将会在第11章中看到它的使用方法。在一个完整的标准C++库的实现中，它也可以被模板类basic\_string所使用。类vector<T>可以将它的迭代器定义为对象指针T\*的同义词。实际上，许多实现都给了这种做法一定的自由度。在本书中给出的实现将这样的迭代器定义为Ptrit<T, ptrdiff\_t, T\*, T&, T\*, T&>，这样做有两个原因：

* 在使用模板类reverse\_iterator来声明vector<T>的反转型迭代器时，避免了对模板部分特化的依赖。
* 当程序员不经意地混淆了对象指针和迭代器时，可以防范许多不安全的编程实践。

除此之外，Ptrit还有另一个优点，那就是它可以当作随机存取迭代器类的极好的原型。

一开始，我们可能会认为最后的两个模板参数是冗余的。然而，更进一步的检查显示，它们有助于指定某些种类的转换构造函数。对于一个常量迭代器来说，这样的构造函数将会把它的非常量兄弟转换成为常量迭代器。而对于非常量迭代器来说，这样的构造函数实际上就相当于一个显式指定的默认复制构造函数(我们不希望出现隐式地将一个常量迭代器转换成为一个非常量迭代器的情况，但我们却希望与之相反的转换过程能够是隐式的)。

**reverse\_iterator**

模板类reverse\_iterator从一个随机存取迭代器中创建出一个反转型迭代器。当你了解了设计反转型迭代器实质原理后，就会发现，其实在reverse\_iterator的代码中并没有用到什么特殊的技巧。下面是我们从它的代码中提取出来的成员还是operator\*()：

Ref operator\*() const

{

    return \*(current-1);

}

简而言之，反转型迭代器返回的是current指向的前一位置元素。当我们仔细想过之后，会发现理由其实很简单。我们一般都是用迭代器的范围[first, last)来指定一个序列。这种半开半闭的表达方法意味着：对于一个非空的序列来说，first是它的一部分，而last只能是越过其结尾。

于是我们为current指定了反转型迭代器，范围为[last, first)。为了使last能够作为存储在current中的值使用，在一个非空的序列中，必须指定current-1处的元素。一旦做了这样的转变，其他所有迭代器的属性也就自然起来。

**Revbidit**

模板类Revbidit从一个双向迭代器中构建一个反转型迭代器。大部分在模板类reverse\_iterator中实现了的方法都在它之中实现出来了。如前面所描述的一样，成员还是operator\*()的定义体现处最基本的技巧。

**istreambuf\_iterator**

模板类istreambuf\_iterator创建了一个输入迭代器，并用它来从一个流缓冲区中提取出类型为E(通常为char)的元素。虽然输入迭代器可能没有输出迭代器那么奇怪(如我们将在下面紧接着讨论的模板类ostreambuf\_iterator)，但它还是可以被风格化的。设计一个输入迭代器所用的技巧包括：

* 向前读取，必要时拥有一个输入值。
* 检测及记录序列结束(或文件结束)。
* 创建一个end-of-sequence值。
* 比较两个迭代器，以检测它们是否到达序列的末端。

模板类istreambuf\_iterator为输入迭代器提供了一个很好的原型，但你还是必须将它与本书所提供的实现中的其他输入迭代器仔细比较，以弄懂它们之间出现不同的原因。例如，该模板类与早先的模板类istrambuf\_iterator的主要不同在于向前读取的逻辑。后者必须读入一堆元素以检测它所要提取的值(类型为U)，这使得它在将向前读取的代价变得最小化方面没有做太多的努力。

从另一个方面来说，标准C++库可以在从输入流中提取值的过程中创建及销毁任意数量的模板类istreambuf\_iterator的对象。销毁每个这样的对象使得输入流处于一个可以获知的状态中，并且不会漏掉任何未被读取的输入。通过使用它的成员还是sgetc()，流缓冲区可以让我们“取数(peek)”的输入元素，也就是说，获得输入的值但不在流中移动定位用的游标。但即使是这样的操作也会导致出现无限期地中止。举例来说，如果该流缓冲区是和一个交互式的输入流（如键盘）相关联的话，就会导致这种情况出现。也就是说，在它必须将所指向的值传给成员函数之前，该迭代器将会推迟所有“需要知道下一个元素的值”的动作。注意，取数操作必须在比较两个迭代器之前进行，以检测该迭代器是否到达了序列结束处。

**ostreambuf\_iterator**

模板类ostreambuf\_iterator创建了一个输出迭代器，用于将类型为E(通常为char)的元素插入到流缓冲区中。它与早先的模板类ostream\_iterator的主要的不同在于对输出错误的处理。一般来说，输出迭代器就像是一个能够无限地产生出元素的源头。没有任何方法可以报告它是否还能存储。但是，标准C++库需要有一种方法，用来拒绝从底层的流缓冲区中向上传递的错误。这也就是我们所新增的成员函数failed及其相关逻辑的由来。

从表面判断，这个模板类中的成员操作符的定义让让难以容忍。它们并没有提供通常与这些操作符相关的语义：

* operator\*()返回的是迭代器的引用，而不是它指向的值。
* operator++()并没有将迭代器向前移动。
* operator++(int)也没有将迭代器向前移动。
* operator=(E)是对赋值操作符的一个非常规的重载。它完成了实际上的输出操作，并按照其语义，将迭代器有效地向前移动了。

然而，请记住：对于输出迭代器next的使用，只能存在于下面的两种格式中：

for(; <not done>; ++next)

    \*next = <whateven>;

或

while(<not done>)

    \*next++ = <whatever>;

粗略看一下在这两种格式的循环中定义的这些成员操作符。我们将会看到，这些操作符提供了最少的功能，以满足输出迭代器的语义需求。

也就是说，模板类ostreambuf\_iterator为我们可能实现的输出迭代器提供了一个很好的原型。STL定义了几种不同的输出迭代器。请仔细学习它们之间的不同处和相同处。输出迭代器是一个强有力的工具，但它们也很特殊，并且隐藏了一些缺陷。

**iterator**

文件xutility的其他部分包括一些与头文件<algorithm>相关的定义。第7章将对这些定义进行讨论。程序清单4-2列出了文件iterator。它定义了剩下的那些与头文件<iterator>相关但又未在文件xutility中定义的实体。

**程序清单4-2: iterator**

// iterator standard header

#ifndef ITERATOR\_

#define TIERATOR\_

#include<xutility>

namespace std{

    // TEMPLATE CLASS back\_insert\_iterator

    template<class C>

    class back\_insert\_iterator: public Outit

    {

    public:

        typedef C container\_type;

        typedef typename C::reference reference;

        typedef typename C::value\_type value\_type;

        explicit back\_insert\_iterator(C& X): container(&X){ }

        back\_insert\_iterator<C>& operator=(

            typename C::const\_reference V)

        {

            container->push\_back(V);

            return \*this;

        }

        back\_insert\_iterator<C>& operator\*()

        {

            return \*this;

        }

        back\_insert\_iterator<C>& operator++()

        {

            return \*this;

        }

        back\_insert\_iterator<C>& operator\*(int)

        {

            return \*this;

        }

    protected:

        C \*container;

    };

    template<class C>

    inline back\_insert\_iterator<C> back\_inserter(C& X)

    {

        return back\_insert\_iterator<C>(X);

    }

    // TEMPLATE CLASS front\_insert\_iterator

    template<class C>

    class front\_insert\_iterator: public Outit

    {

    public:

        typedef C container\_type;

        typedef typename C::reference reference;

        typedef typename C::value\_type value\_type;

        explicit front\_insert\_iterator(C& X): container(&X) { }

        front\_insert\_iterator<C)& operator=(

            typename C::const\_reference V)

        {

            container->push\_front(V);

            return \*this;

        }

        front\_insert\_iterator<C)& operator\*()

        {

            return \*this;

        }

        front\_insert\_iterator<C)& operator++()

        {

            return \*this;

        }

        front\_insert\_iterator<C)& operator++(int)

        {

            return \*this;

        }

    protected:

        C \*container;

    };

    template<class C> inline

    front\_insert\_iterator<C> front\_inserter(C& X)

    {

        return front\_insert\_iterator<C>(X);

    }

    // TEMPLATE CLASS insert\_itreator

    template<class C>

    class insert\_iterator: public Outit

    {

    public:

        typedef C container\_type;

        typedef typename C::reference reference;

        typedef typename C::value\_type value\_type;

        insert\_iterator(C& X, typename C::iterator I): container(&X), iter(I) { }

        insert\_iterator<C>& operator=(typename C::const\_reference V)

        {

            iter = container->insert(iter, V);

            ++iter;

            return \*this;

        }

        insert\_iterator<C>& operator\*()

        {

            return \*this;

        }

        insert\_iterator<C>& operator++()

        {

            return \*this;

        }

        insert\_iterator<C>& operator++(int)

        {

            return \*this;

        }

    protected:

        C \*container;

    typename C::iterator iter;

    };

    template<class C, class XI> inline

    insert\_iterator<C> inserter(C& X, XI I)

    {

    return insert\_iterator<C>(X, C::iterator(I));

    }

    // TEMPLATE CLASS istream\_iterator

    template<class T, class E=char, class Tr=char\_traits<E>, class Dist=ptrdiff\_t>

    class istream\_iterator

        : public iterator<input\_iterator\_tag, T, Dist, T\*, T&>

    {

    public:

        typedef istream\_iterator<T, E, Tr, Dist> Myt;

        typedef E char\_type;

        typedef Tr traits\_type;

        typedef basic\_istream<E, Tr> istream\_type;

        istream\_iterator(): Istr(0) {}

        istream\_iterator(istream\_type& I): Istr(&I) { Getval();}

        const T \*operator\*() const

        {

            return Val;

        }

        const T \*operator->() const

        {

            return &\*\*this;

        }

        Myt& operator++()

        {

            Getval();

            return \*this;

        }

        Myt& operator++(int)

        {

            Myt tem=\*this;

            Getval();

            return tmp;

        }

        bool Equal(const Myt& X) const

        {

            return Istr == X.Istr;

        }

    protected:

        void Getval()

        {

            if(Istr != 0 && !(\*Istr>>Val))

                Istr = 0;

        }

        istream\_type \*Istr;

        T Val;

    };

    // istream\_iterator TEMPLATE OPERATORS

    template<class T, class E, class Tr, class Dist>

    inline bool operator==(

        const istream\_iterator<T, E, Tr, Dist>& X,

        const istream\_iterator<T, E, Tr, Dist>& Y)

    {

        return X.Equal(Y);

    }

    template<class T, class E, class Tr, class Dist>

    inline bool operator!=(

        const istream\_iterator<T, E, Tr, Dist>& X,

        const istream\_iterator<T, E, Tr, Dist>& Y)

    {

        return !(X == Y);

    }

    // TEMPLATE CLASS cstream\_iterator

    template<class T, class E=char, class Tr=char\_traits<E>>

    class ostream\_iterator: public Outit

    {

    public:

        typedef T value\_type;

        typedef E char\_type;

        typedef Tr traits\_type;

        typedef basic\_ostream<E, Tr> ostream\_type;

        ostream\_iterator(ostream\_type& O, const E \*D = 0): Ostr(&O), Delim(D) { }

        ostream\_iterator<T, E, Tr>& operator=(const T& X)

        {

            \*Ostr << X;

            if(Delim != 0)

                \*Ostr << Delim;

            return \*this;

        }

        ostream\_iterator<T, E, Tr>& operator\*()

        {

            return \*this;

        }

        ostream\_iterator<T, E, Tr>& operator++()

        {

            return \*this;

        }

        ostream\_iterator<T, E, Tr>& operator++(int)

        {

            return \*this;

        }

    protected:

        const E \* Delim;

        ostream\_type \*Ostr;

    };

    // TEMPLATE FUNCTION val\_type

    template<class It>

    inline typename iterator\_traits<It>::value\_type \*Val\_type(It)

    {

        return 0;

    }

    // TEMPLATE FUNCTION advance

    template<class InIt, class D>

    inline void advane(InIt& I, D N)

    {

        Advance(I, N, Iter\_cat(I));

    }

    template<class InIt, class D>

    inline void advane(InIt& I, D N, input\_iterator\_tag)

    {

        for(; 0<N; --N)

            ++I;

    }

    template<class FwdIt, class D>

    inline void Advance(FwdIt& I, D N, forward\_iterator\_tag)

    {

        for(; 0<N; --N)

            ++I;

    }

    template<class BidIt, class D>

    inline void Advance(BidIt& I, D N, bidirectional\_iterator\_tag)

    {

        for(; 0<N; --N)

            ++I;

        for(; N<0; ++N)

            --I;

    }

    template<class RanIt, class D>

    inline void Advance(RanIt& I, D N, random\_access\_iterator\_tag)

    {

        I += N;

    }

    template<class It>

    inline typename iterator\_traits<It>::difference\_type \*Dist\_type(It)

    {

        return 0;

    }

}; // namespace std

#endif // iterator

**back\_insert\_iterator**

模板类back\_insert\_iterator创建了一个输出迭代器，用来往容器末端添加新的元素。与其他输出迭代器一起，它使用了一个不常见的赋值操作符operator=(Cont::const\_reference V)来实现这样的添加动作。实际上的添加动作是由它调用指定容器的container->push\_back(V)来完成的。

**back\_inserter**

模板函数back\_inserter创建了类back\_insert\_iterator<Cont>的一个对象，并使之可以与其容器参数协同工作。

**front\_insert\_iterator**

模板类front\_insert\_iterator创建了一个输出迭代器，它通过调用container->push\_front(V)，在容器的前端添加元素（逆序）。在其他方面，它都类似于模板类back\_insert\_iterator。

**front\_inserter**

模板函数front\_inserter创建了类front\_insert\_iterator<Cont>的一个对象，并使之可以与其容器参数协同工作。

insert\_iterator

模板类insert\_iterator创建了一个输出迭代器，用它来向容器中的指定位置插入元素。与模板类back\_insert\_iterator不同的是，这样的迭代器必须还保存一个迭代器iter，用来指定迭代器中的插入点。请注意在这两个模板类之中成员函数operator++(int)返回值的不同。保存有状态改变信息的迭代器必须返回一个该迭代器的最新版本的引用。而传统的返回一个新对象的方法在此时就不再适合了。

**inserter**

模板函数inserter创建了类insert\_iterator<Cont, Iter>的一个对象，并使之可以与其容器参数协同工作。

**istream\_iterator**

模板类istream\_iterator创建了一个输入迭代器，并通过使用一个合适的operator>>的重载，从输入流中提取类型未U的值。请将它与我们在本章早些时候描述的模板类istreambuf\_iterator仔细比较。

**ostream\_iterator**

模板类ostream\_iterator创建了一个输出迭代器，并通过使用一个合适的operator<<的重载，向输出流中插入类型为U的值。它遵循了输出迭代器的通常形式。请将它与我们在本章早些时候描述的模板类istreambuf\_iterator仔细比较。

**Val\_type、Dist\_type**

模板参数Val\_type返回一个空指针，它的类型揭示了该函数的迭代器参数的“值类型”（即成员类型value\_type）。同样地，模板函数Dist\_type返回的是“差距类型”（即成员类型difference\_type，有段实际我们也把它叫做distance\_type）。在本章的早些时候，我们就已经讨论过这些看起来没什么意义的函数存在的历史原因。

**advance**

最后，模板函数advance实现了一件显然是很简单的事情：把迭代器向前移动指定次数。但它却更像前面讨论的模板函数distance。在不同种类的迭代器中，对迭代器向前移动的最佳方法都不相同。我们在此又使用了一次标签的重载，以在算法的不同实现中选择最佳的。

## 4.5 测试<iterator>

titerato.c

程序清单4-3列出了文件titerto.c。由于头文件<iterator>实在太大了，我们的测试程序就将它分做不同部分，分别进行了测试：

* 用以测试迭代器的不同属性及完成有关迭代器的简单操作的模板
* 模板类reverse\_iterator
* 插入型迭代器
* 模板类istream\_iterator
* 模板类ostream\_iterator
* 模板类istreambuf\_iterator
* 模板类ostreambuf\_iterator

**程序清单4-3: titerato.c**

// test <iterator>

#include<assert.h>

#include<iostream>

#include<string.h>

#include<strstream>

#include<deque>

#include<iterator>

using namespace std;

typedef char \*PtrIt;

// TEST GENERAL PROPERTY TEMPLATES

void takes\_ran\_tag(random\_access\_iterator\_tag)

{

}

void test\_prop()

{

    random\_access\_iterator\_tag \*ran\_tag = (random\_access\_iterator\_tag\*)0;

    bidirectional\_iterator\_tag \*bid\_tag = (random\_access\_iterator\_tag\*)0;

    forward\_iterator\_tag \*fwd\_tag = (bidirectional\_iterator\_tag\*)0;

    input\_iterator\_tag \*in\_tag = (forward\_iterator\_tag\*)0;

    output\_iterator\_tag \*p\_out\_tag = 0;

    typedef iterator<input\_iterator\_tag, float, short, float\*, float&> Iter;

    float f1;

    Iter::iterator\_category \*it\_tag = (input\_iterator\_tag\*)0;

    Iter::value\_type \*it\_val = (float\*)0;

    Iter::difference\_type \*it\_dist = (short\*)0;

    Iter::pointer it\_ptr = (float\*)0;

    Iter::reference it\_ref = f1;

    typedef iterator\_traits<Iter> Traits;

    Traits::iterator\_category \*tr\_tag = (input\_iterator\_tag\*)0;

    Traits::value\_type \*tr\_val = (float\*)0;

    Traits::difference\_type \*tr\_dist = (short\*)0;

    Traits::pointer tr\_ptr = (float\*)0;

    Traits::reference tr\_ref = f1;

    typedef iterator\_traits<PtrIt> Ptraits;

    char ch;

    takes\_ran\_tag(Ptraits::iterator\_category());

    Ptraits::value\_type \*ptr\_val = (char\*)0;

    Ptraits::difference\_type \*ptr\_dist = (ptrdiff\_t\*)0;

    Ptraits::pointer ptr\_ptr = (char\*)0;

    Ptraits::reference ptr\_ref = ch;

    const char\* pc = "abcdefg";

    advance(pc, 4);

    assert(\*pc == 'e');

    advance(pc, -1);

    assert(\*pc == 'd');

    assert(distance(pc, pc+3) == 3);

}

// TEST reverse\_iterator

typedef reverse\_iterator<PtrIt> RevIt;

class MyrevIt: public RevIt

{

public:

    MyrevIt(RevIt::iterator\_type p): RevIt(p) { }

    RevIt::iterator\_type get\_current() const { return current; }

};

void test\_revit()

{

    char\* pc = (char\*)"abcdefg" + 3;

    PtrIt pcit(pc);

    RevIt::iterator\_type\* p\_iter = (PtrIt\*)0;

    RevIt rit0, rit(pcit);

    assert(rit.base() == pcit);

    assert(\*rit == 'c');

    assert(\*++rit == 'b');

    assert(\*rit++ == 'b' && \*rit == 'a');

    assert(\*--rit == 'b');

    assert(\*rit-- == 'b' && \*rit == 'c');

    assert(\*(rit += 2) == 'a');

    assert(\*(rit -= 2) == 'c');

    assert(\*(rit + 2) == 'a' && \*rit == 'c');

    assert(\*(rit - 2) == 'e' && \*rit == 'c');

    assert(rit[2] == 'a');

    assert(rit == rit);

    assert(!(rit < rit) && rit < rit+1);

    assert((rit+2) - rit == 2);

    MyrevIt myrit(pc);

    assert(myrit.get\_current() == pcit);

}

// TEST INSERTTION ITERATORS

typedef deque<char, allocator<char>> Cont;

typedef back\_insert\_iterator<Cont> BackIt;

class MybackIt: public BackIt

{

public:

    MybackIt(BackIt::container\_type& c) : BackIt(c) { }

    BackIt::container\_type\* get\_container() const

    {

        return container;

    }

};

typedef front\_insert\_iterator<Cont> FrontIt;

class MyfrontIt: public FrontIt

{

public:

    MyfrontIt(FrontIt::container\_type& c): FrontIt(c) { }

    FrontIt::container\_type\* get\_container() const

    {

        return container;

    }

};

typedef insert\_iterator<Cont> InsIt;

class MyinsIt: public InsIt

{

public:

    MyinsIt(InsIt::container\_type& c, Cont::iterator it): InsIt(c, it) { }

    InsIt::container\_type \*get\_container() const

    {

        return container;

    }

    Cont::iterator get\_iterator() const

    {

        return iter;

    }

};

void test\_inserts()

{

    Cont c0;

    char ch;

    BackIt::container\_type \*pbi\_cont = (Cont\*)0;

    BackIt::reference \*pbi\_ref = &ch;

    BackIt::value\_type \*pbi\_val = (char\*)0;

    BackIt bit(c0);

    \*bit = 'a', ++bit;

    \*bit++ = 'b';

    assert(c0[0] == 'a' && c0[1] == 'b');

    MybackIt mybkit(c0);

    assert(mybkit.get\_container() == &c0);

    \*back\_inserter(c0)++ = 'x';

    assert(c0[2] == 'x');

    FrontIt::container\_type \*pfi\_cont = (Cont\*)0;

    FrontIt::reference \*pfi\_ref = &ch;

    FrontIt::value\_type \*pfi\_val = (char\*)0;

    FrontIt fit(c0);

    \*fit = 'c', ++fit;

    \*fit++ = 'd';

    assert(c0[0] == 'd' && c0[1] == 'c');

    \*front\_inserter(c0)++ = 'y';

    assert(c0[0] == 'y');

    InsIt::container\_type \*pii\_cont = (Cont\*)0;

    InsIt::reference \*pii\_ref = &ch;

    InsIt::value\_type \*pii\_val = (char\*)0;

    InsIt iit(c0, c0.begin());

    \*iit = 'e', ++iit;

    \*iit++ = 'f';

    assert(c0[0] == 'e' && c0[1] == 'f');

    MyinsIt myinsit(c0, c0.begin());

    assert(myinsit.get\_container() == &c0);

    assert(myinsit.get\_iterator() == c0.begin());

    \*inserter(c0, c0.begin())++ = 'z';

    assert(c0[0] == 'z');

}

//TEST istream\_iterator

void test\_istreamit()

{

    istrstream istr(" 0 1 2 3");

    typedef istream\_iterator<int, char, char\_traits<char>, ptrdiff\_t> IstrIt;

    IstrIt::char\_type \*p\_char = (char\*)0;

    IstrIt::traits\_type \*p\_traits = (char\_traits<char>\*)0;

    IstrIt::istream\_type \*p\_istream = (istream\*)0;

    IstrIt iit0, iit(istr);

    int n;

    for(n=0; n<5 && iit != iit0; ++n)

    assert(\*iit++ == n);

    assert(n==4);

}

// TEST ostream\_iterator

void test\_ostreamit()

{

    ostrstream ostr0, ostr;

    typedef ostream\_iterator<int, char, char\_traits<char>> OstrIt;

    OstrIt::value\_type \*p\_val = (int\*)0;

    OstrIt::char\_type \*p\_char = (char\*)0;

    OstrIt::traits\_type \*p\_traits = (char\_traits<char>\*)0;

    OstrIt::ostream\_type \*p\_ostream = (basic\_ostream<char, char\_traits<char>>\*)0;

    OstrIt oit0(ostr0), oit(ostr, "||");

    oit0=1, ++oit0;

    \*oit0++ = 2;

    ostr0 << ends;

    assert(strcmp(ostr0.str(), "12") == 0);

    ostr.freeze(false);

    \*oit =1, ++oit;

    \*oit++ = 2;

    ostr << ends;

    assert(strcmp(ostr.str(), "1||2||") == 0);

    ostr.freeze(false);

}

// TEST istreambuf\_iterator

void test\_istrbufit()

{

    istrstream istr("0123"), istr1("");

    typedef istreambuf\_iterator<char, char\_traits<char>> IsbIt;

    IsbIt::char\_type \*p\_char = (char\*)0;

    IsbIt::traits\_type \*p\_traits = (char\_traits<char>\*)0;

    IsbIt::int\_type \*p\_int = (int\*)0;

    IsbIt::streambuf\_type \*p\_streambuf = (basic\_streambuf<char, char\_traits<char>>\*)0;

    IsbIt::istream\_type \*p\_istream = (basic\_istream<char, char\_traits<char>>\*)0;

    IsbIt iit0, iit(istr), iit1(istr1.rdbuf());

    int n;

    for(n=0; n<5 && iit != iit0; ++n)

        assert(\*iit++ == (n + '0'));

    assert(n == 4);

    assert(iit0.equal(iit1));

}

// TEST ostreambuf\_iteraotr

void test\_ostrbufit()

{

    ostrstream ostr;

    typedef ostreambuf\_iterator<char, char\_traits<char>> OsbIt;

    OsbIt::char\_type \*p\_char = (char\*)0;

    OsbIt::traits\_type \*p\_traits = (char\_traits<char>\*)0;

    OsbIt::streambuf\_type \*p\_streambuf = (basic\_streambuf<char, char\_traits<char>>\*)0;

    OsbIt::ostream\_type \*p\_ostream = (basic\_ostream<char, char\_traits<char>>\*)0;

    OsbIt oit0((OsbIt::streambuf\_type\*)0), oit(ostr);

    \*oit0++ = 'x';

    assert(oit0.failed());

    \*oit = '1', ++oit;

    \*oit++ = '2';

    ostr << ends;

    assert(strcmp(ostr.str(), "12") == 0);

    assert(!oit.failed());

    ostr.freeze(false);

}

// TEST <iterator>

int main()

{

    test\_prop();

    test\_revit();

    test\_inserts();

    test\_istreamit();

    test\_ostreamit();

    test\_istrbufit();

    test\_ostrbufit();

    cout<<"SUCCESS testing <iterator>" << endl;

    return 0;

}

如果所有的测试都正常运行的话，测试程序将打印出：

SUCCESS testing <iterator>

并正常退出。

## 4.6 习题

习题4-1 为什么input\_iterator\_tag不派生自output\_iterator\_tag？

习题4-2 STL为什么要定义模板类reverse\_forward\_iterator？

习题4-3 定义有关迭代器，它可以将元素插入到容器的开始处，但与模板类front\_insert\_iterator不同的是，它的插入动作并不是逆序的。

习题4-4 定义有关迭代器，它可以将元素插入到容器的末端，但是插入是以逆序进行的（就好像模板类front\_insert\_iterator的对象一样）。

习题4-5 对迭代器来说，成员函数operator++(int)通常都返回该迭代器的一份拷贝。然而模板类insert\_iterator及模板类ostreambuf\_iterator返回的却是对象本身的有关引用。请解释它们这样做的原因。

习题4-6 定义一个模板类forward\_iterator<FwdIt>，它简单地“包装”了一个类FwdIt的前向迭代器。请确保它包括所有前向迭代器要求的操作。

习题4-7 定义一系列“严格的”迭代器，以包装所有的五种迭代器中的每一种。这些“严格的”迭代器应该能够报告出程序中对它们的错误使用。当迭代器被创建后，它们甚至还可以对试图将迭代器向前移动到指定的范围外这种情况给出警告信息。

习题4-8 给定一个迭代器以及它所属的种类，如何验证该迭代器的行为不超过其种类所要求的行为集？

# 第5章 Memory

## 5.1 背景知识

在STL的13个头文件中，<memory>是其中最难描述的一个。它以不同寻常的方式为容器中的元素分配存储空间。它同样也为某些算法执行期间产生的临时对象提供机制。早期的STL代码中只存在一些与之大体相似的方法，而且也不值得炫耀。例如，在流行的PC体系结构上，通过采用一种没有被C标准所概括d指针表达形式，容器可以在“近堆(near heap)”或“远堆(far heap)”上分配元素。，然而，在C++标准中，这种特殊d分配方法已经被分配器(allocator，一种在分配及释放元素存储空间时使用的中间对象)所代替。