第15章　模板和包容器类

包容器类常用于创建面向对象程序的构造模块(building block),它使得程序内部代码更容易构造。

一个包容器类可描述为容纳其他对象的对象。可把它想像成允许向它存储对象，而以后可以从中取出这些对象的高速暂存存储器或智能存储块。

包容器类非常重要，曾被认为是早期的面向对象语言的基础。例如，在 Smalltalk中，程序员把语言设想为带有类库的程序翻译器，而类库的重要部分就是包容器类。所以 C++编译器的供应商很自然地会为用户提供包容器类库。

像许多早期的别的C++库一样，早期的包容器类库仿效 Smalltalk的基于对象的层次结构，该结构非常适合Smalltalk，但是该结构在C++的使用中却带来了一些不便，因此有必要寻求另外的方法。

包容器类是解决不同类型的代码重用问题的另一种方法。继承和组合为对象代码的重用提供一种方法，而C++的模板特性为源代码的重用提供一种方法。

虽然C++模板是通用的编程工具，但当它们被引入该语言时它们似乎不支持基于对象的包容器类层次结构。新近版本的包容器类库则完全由模板构造，程序员可以很容易地使用。

本章首先介绍包容器类和采用模板实现包容器类的方法，接着给出一些包容器类和怎样使用它们的例子。

15.1　包容器和循环子

假若打算用C语言创建一个堆栈，我们需要构造一个数据结构和一些相关函数，而在 C++中，则把二者封装在一个抽象数据类型内。下面的stack类是一个栈类的例子，为简化起见，它仅处理整数：

```
//: ISTACK.CPP -- Simple integer stack
#include <assert.h>
#include <iostream.h>

class istack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  istack() : top(0) { stack[top] = 0; }
  void push(int i) {
    if(top < ssize) stack[top++] = i;
  }
  int pop() {
    return stack[top > 0 ? --top : top];
  }
```

```
  friend class istackIter;
};

// An iterator is a "super-pointer":
class istackIter {
  istack& S;
  int index;
public:
  istackIter(istack& is)
    : S(is), index(0) {}
  int operator++() { // Prefix form
    if (index < S.top - 1) index++;
    return S.stack[index];
  }
  int operator++(int) { // Postfix form
    int returnval = S.stack[index];
    if (index < S.top - 1) index++;
    return returnval;
  }
};
// For interest, generate Fibonacci numbers:
int fibonacci(int N) {
  const sz = 100;
  assert(N < sz);
  static F[sz]; // Initialized to zero
  F[0] = F[1] = 1;
  // Scan for unfilled array elements:
  int i;
  for(i = 0; i < sz; i++)
    if(F[i] == 0) break;
  while(i <= N) {
    F[i] = F[i-1] + F[i-2];
    i++;
  }
  return F[N];
}

main() {
  istack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  istackIter it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
```

```
    for(int k = 0; k < 20; k++)
      cout << is.pop() << endl;
}
```

类istack是最为常见的自顶向下式的堆栈的例子。为了简化，此处栈的尺寸是固定的，但是也可以对其修改，通过把存储器安排在堆中分配内存，来自动地扩展其长度（后面的例子会介绍）。

第二个类istackIter是循环子的例子，我们可以把其当作仅能和 istack协同工作的超指针。注意，istackIter是istack的友元，它能访问istack的所有私有成员。

像一个指针一样， istackIter的工作是扫视istack并能在其中取值。在上述的简单的例子中，istackIter可以向前移动（利用运算符++的前缀和后缀形式）和取值。然而，此处却并没有对定义循环子方法予以限制。完全可以允许循环子在相关包容器中以任何方法移动和对包容的值进行修改。可是，按照惯例，循环子是由构造函数创建的，它只与一个包容器对象相连，并且在生命周期中不重新相连。（大多数循环子较小，所以我们可以容易地创建其他循环子。）

为了使例子更有趣，这个 fibonacci函数产生传统的“ 兔子繁殖数 ”，这是一个相当有效的实现，因为它决不会多次产生这些数。

在主程序main()中，我们可以看到栈和它的相关循环子的创建和使用。一旦创建了这些类，便可以很方便的使用它们。

包容器的必要性

很明显，一个整数堆栈不是一个重要的工具。包容器类的真正的需求是在堆上使用 new创建对象和使用delete析构对象的时候体现的。一个普遍的程序设计问题是程序员在写程序时不知道将创建多少对象。例如在设计航空交通控制系统时不应限制飞机的数目，不希望由于实际飞机的数目超过设计值而导致系统终止。在 CAD系统设计中，可以安排许多造型，但是只有用户能确定到底需要多少造型。我们一旦注意到上述问题，便可在程序开发中发现许多这样的例子。

依赖虚存储器去处理“ 存储器管理 ”的 C程序员常常发现 new、delete和包容器类思想的混乱。表面上看，创建一个囊括任何可能需求的 huge型全局数组是可行的，这不必有很多考虑（并不需要弄清楚 malloc()和free()）,但是这样的程序移植性较差，而且暗藏着难以捕捉的错误。

另外，创建一个huge型全局数组对象，构造函数和析构函数的开销会使系统效率显著地下降。C++中有更好的解决方法：将所需要的对象用 new创建并将其指针放入包容器中，待实际使用时将其取出。该方法确定了只有在绝对需要时才真正创建对象。所以在启动系统时可以忽略初始化条件，在环境相关的事件发生时才真正创建对象。

在大多数情况下，我们应当创建存放感兴趣的对象的包容器，应当用 new创建对象，然后把结果指针放在包容器中（在这个过程中向上映射），具体使用时再将指针从包容器中取出。该技术有很强的灵活性且易于分类组织。

15.2 模板综述

现在出现了一个问题， istack可存放整数，但是也应该允许存放造型、航班、工厂等等数据对象，如果这种改变每次都依赖源码的更新，则不是一个明智的办法。应该有更好的重用

方法。

有三种源代码重用方法：用于契约的 C方法；影响过C++的Smalltalk方法；C++的模板方法。

15.2.1 C方法

毫无疑问，应该摒弃C方法，这是由于它表现繁琐、易发生错误、缺乏美感。如果需要拷贝stack的源码并对其手工修改，还会带入新的错误。这是非常低效的技术。

15.2.2 Smalltalk 方法

Smalltalk方法是通过继承来实现代码重用的，既简单又直观。每个包容器类包含基本通用类object的所属项目。Smalltalk的基类库十分重要，它是创建类的基础。创建一个新类必须从已有类中继承。可以从类库中选择功能和需求接近的已有类作为父类，并在对父类的继承中加以修正从而创建一个新类。很明显这种方法可以减少我们的工作而提高效率（因此花大量的时间去学习Smalltalk类库是成为熟练的Smalltalk程序员的必由之路）。

所以这意味着Smalltalk的所有类都是单个继承树的一部份。当创建新类时必须继承树的某一枝。大多数树是已经存在的（它是Smalltalk的类库），树的根称作object——每个Smalltalk包容器所包含的相同的类。

这种方法表现出的整洁明了在于Smalltalk类层次上的任何类都源于object的派生，所以任何包容器可容纳任何类，包括包容器本身。基于基本通用类的（常称为 object）的单树形层次模式称为“ 基于对象的继承 ”。我们可能听说过这个概念，并猜想这是另一个 OOP的基本概念，就像“ 多态性 ”一样。但实际上这仅仅意味着以object（或相近的名称）为根的树形类结构和包含object的包容器类。

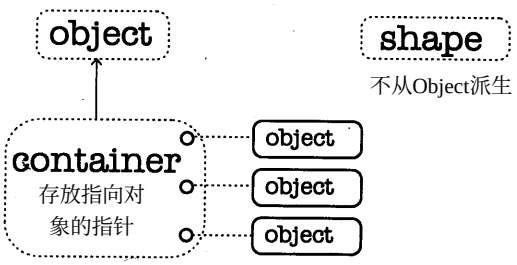

图 15-1

由于Smalltalk类库的发展史较C++更长久，且早期的 C++编译器没有包容器类库，所以C++能将Smalltalk类库的良好思想加以借鉴。这种借鉴出现在早期的C++实现中 [1]，由于它表现为一个有效的代码实体，许多人开始使用它，但把它用于包容器类的编程时则发现了一个问题。

该问题在于，在 Smalltalk中，我们可以强迫人们从单个层次结构中派生任何东西，但在C++中则不行。我们本来可能拥有完善的基于 object的层次结构以及它的包容器类，但是当我们从其他不用这种层次结构的供应商那里购买到一组类时，如造型类、航班类等等（层次结构增加开销，而C程序员可以避免这种情况），我们如何把这些类树集成进基于object的层次结构的包容器中呢？这些问题如下所示：

由于C++支持多个无关联层次结构，所以 Smalltalk的“ 基于object的层次结构 ”不能很好地工作。

解决方案似乎是明显的。如果我们有许多继承层次结构，我们就应当能从多个类继承：多重继承可以解决上述问题。所以我们应按下述的方法去实施：

[1] OOPS库，Keith Gorlen在NIH时开发的。一般以公开源代码的形式使用。

oshape具有shape的特点和行为，但它也是object的派生类，所以可将其置于包容器内。

但是原先的C++并不包含多重继承，当包容器问题出现时，C++供应商被迫去增加多重继承的特性。另外一些程序员一直认为多重继承不是一个好主意，因为它增加了不必要的复杂性。那时一句再三重复的话是“C++不是Smalltalk”，这意味着“不要把基于object的层次结构用于包容器类”。但最终[1]，由于不断的压力，还是把多重继承加入到该语言中了。编译器供应商将基于object的包容器类层次结构加入产品中并进行了调整，它们中的大多数由模板来替代。我们可以为多重继承是否可以解决大多数编程问题而进行争论，但是，在下一章中可以看到，由于其复杂性，除某些特殊情况，最好避免使用它。

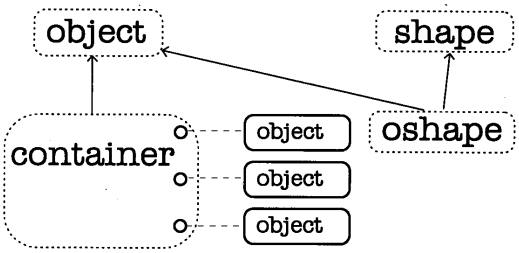

图 15-2

15.2.3 模板方法

尽管具有多重继承的基于对象的层次结构在概念上是直观的，但是在实践上较为困难。在Stroustrup的最初著作[2]中阐述了基于对象层次的一种更可取的选择。包容器类被创造作为参数化类型的大型预处理宏，而不是带自变量的模板，这些自变量能为我们所希望的类型替代。当我们打算创建一个包容器存放某个特别类型时，应当使用一对宏调用。

不幸的是，上述方法在当时的Smalltalk文献中被弄混淆了，加之难以处理，基本上没有什么人将其澄清。

在此期间，Stroustrup和贝尔实验室的C++小组对原先的宏方法进行了修正，对其进行了简化并将它从预处理范围移入了编译器。这种新的代码替换装置被称为模板[3]，而且它表现了完全不同的代码重用方法：模板对源代码进行重用，而不是通过继承和组合重用对象代码。包容器不再存放称为object的通用基类，而由一个非特化的参数来代替。当用户使用模板时，参数由编译器来替换，这非常像原来的宏方法，却更清晰、更容易使用。

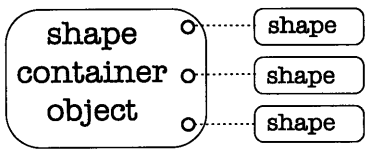

图 15-3

现在，使用包容器类时关于继承和组合的忧虑可以消除了，我们可以采用包容器的模板版本并且复制出和我们的问题相关的特定版本，像这样：

编译器会为我们做这些工作，而我们最终是以所需要的包容器去做我们的工作，而不是用那些令人头疼的继承层次。在C++中，模板实现了参数化类型的概念。模板方法的另一好处是对继承不熟悉、不适应的程序新手能正确地使用密封的包容器类。

15.3 模板的语法

“模板（template）”这一关键字会告诉编译器下面的类定义将操作一个或更多的非特定类型。当对象被定义时，这些类型必须被指定以使编译器能够替代它们。

---

[1] 我们也许决不能知道其全部，因为该语言的控制仍在AT&T中。

[2] The C++Programming Language，由Bjarne Stroustrup著（第一版，Addision-Wesley, 1986）。

[3] 模板的灵感最初出现在ADA。

下面是一个说明模板语法的小例子：

```
//: STEMP.CPP -- Simple template example
#include <iostream.h>
#include <assert.h>

template<class T>
class array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    assert(index >= 0 && index < size);
    return A[index];
  }
};
main() {
  array<int> ia;
  array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
}
```

除了这一行

```
template<class T>
```

以外，它看上去像一个通常的类。这里 T是替换参数，它表示一个类型名称。在包容器类中，它将出现在那些原本由某一特定类型出现的地方。

在array中，其元素的插入和取出都用相同的函数，即重载的 operator[]来实现。它返回一个引用，因此可被用于等号的两边。注意，当下标值越界时，标准 C库的宏assert()将输出提示信息（使用assert()而不是allege()在于我们可以在调试后彻底移去测试代码）。这里，抛出一个异常，并由类的用户处理它会更适合一些，关于这些将在第 17章中做进一步介绍。

在main()中，我们可以看到非常容易地创建包含了不同类型对象的数组。当我们说：

```
array<int> ia;
array<float> fa;
```

这时，编译器两次扩展了数组模板（这被称为实例），创建两个新产生的类，我们可以把它们当作array_int和array_float（不同的编译器对名称有不同的修饰方法）。这些类就像手工创建的一样，除非你定义对象ia和fa，编译器会为你创建它们。注意要避免类在编译和连接中被重复定义。

15.3.1 非内联函数定义

当然，有时我们使用非内联成员函数定义，这时编译器会在成员函数定义之前察看模板声

明。下面在前述例子的基础上加以修正来说明非内联成员函数的定义：

```
//: STEMP2.CPP -- Non-inline template example
#include <assert.h>

template<class T>
class array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};

template<class T>
T& array<T>::operator[](int index) {
  assert(index >= 0 && index < size);
  return A[index];
}

main() {
  array<float> fa;
  fa[0] = 1.414;
}
```

注意，在成员函数的定义中，类名称被限制为模板参数类型：array<T>。

你可以想象在一些混合型中编译器实际支持两个名字和参数类型。

• 头文件

甚至是在定义非内联函数时，模板的头文件中也会放置所有的声明和定义。这似乎违背了通常的头文件规则：“不要在分配存储空间前放置任何东西”，这条规则是为了防止在连接时的多重定义错误。但模板定义很特殊。由template<…>处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

有时，也可能为了满足特殊的需要（例如，强制模板实例仅存在于一简单的 Windows DLL 文件中）而要在一个独立的CPP文件中放置模板的定义。大多数编译器有一些机制允许这么做，那么我们就必须检查我们特定的编译器说明文档以便使用它。

15.3.2 栈模板(the stack as a template)

对于ISTACK.CPP的包容器和循环子(第15.1节)，可以使用模板，作为普通包容器类实现：

```
//: STACKT.H -- Simple stack template
#ifndef STACKT_H_
#define STACKT_H_
template<class T> class stacktIter; // declare

template<class T>
class stackt {
```

```
  enum { ssize = 100 };
  T stack[ssize];
  int top;
public:
  stackt() : top(0) { stack[top] = 0; }
  void push(const T& i) {
    if(top < ssize) stack[top++] = i;
  }
  T pop() {
    return stack[top > 0 ? --top : top];
  }
  friend class stacktIter<T>;
};

template<class T>
class stacktIter {
  stackt<T>& S;
  int index;
public:
  stacktIter(stackt<T>& is)
    : S(is), index(0) {}
  T& operator++() { // Prefix form
    if (index < S.top - 1) index++;
    return S.stack[index];
  }
  T& operator++(int) { // Postfix form
    int returnIndex = index;
    if (index < S.top - 1) index++;
    return S.stack[returnIndex];
  }
};
#endif // STACKT_H_
```

注意在引用模板名称的地方，必须伴有该模板的参数列表，如 stackt<T>& S。我们可以想象，模板参数表中的参数将被重组，以对于每一个模板实例产生唯一的类名称。

同时也注意到，模板会对它包含的对象做一定的假设。例如，在 push()函数中，stackt会认为T的内部有一种赋值运算。

这里有一个修正过的例子用于检验模板：

```
//: STACKT.CPP -- Test simple stack template
#include <assert.h>
#include <iostream.h>
#include "..\14\stackt.h"
// For interest, generate Fibonacci numbers:
int fibonacci(int N) {
  const sz = 100;
```

```
  assert(N < sz);
  static F[sz]; // Initialized to zero
  F[0] = F[1] = 1;
  // Scan for unfilled array elements:
  int i;
  for(i = 0; i < sz; i++)
    if(F[i] == 0) break;
  while(i <= N) {
    F[i] = F[i-1] + F[i-2];
    i++;
  }
  return F[N];
}

main() {
  stackt<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  stacktIter<int> it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
}
```

唯一的不同是在实例is和it的创建中：我们指明了栈和循环子应该存放在模板参数表内部对象的类型。

15.3.3 模板中的常量

模板参数并不局限于有类定义的类型，可以使用编译器内置类型。这些参数值在模板特定实例时变成编译期间常量。我们甚至可以对这些参数使用缺省值：

```
//: MBLOCK.CPP -- Built-in types in templates
#include <assert.h>
#include <iostream.h>

template<class T, int size = 100>
class mblock {
  T array[size];
public:
  T& operator[](int index) {
    assert(index >= 0 && index < size);
    return array[index];
  }
};
```

```
class number {
  float f;
public:
  number(float F = 0.0f) : f(F) {}
  number& operator=(const number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const number& x) {
      return os << x.f;
  }
};

template<class T, int sz = 20>
class holder {
  mblock<T, sz>* np;
public:
  holder() : np(0) {}
  number& operator[](int i) {
    assert(i >= 0 && i < sz);
    if(!np) np = new mblock<T, sz>;
    return np->operator[](i);
  }
};

main() {
  holder<number, 20> H;
  for(int i = 0; i < 20; i++)
    H[i] = i;
  for(int j = 0; j < 20; j++)
    cout << H[j] << endl;
}
```

类mblock是一个可选的数组对象，我们不能在其边界以外进行索引。（如果出现这种情况，将在第17章中介绍比assert()更好的方法。）

类holder和mblock极为相似，但是在 holder中有一个指向 mblock的指针，而不是含有mblock类型的嵌入式对象。该指针并不在 holder的构造函数中初始化，其初始化过程被安排在第一次访问的时候。 如果我们正在创建大量的对象，却又不立即全部访问它们，可以用这种技术，以节省存储空间。

15.4 stash & stack模板

贯穿本书且不断修正更新的stash和stack都是真正的包容器类，所以将其转化成为模板是必要的。但是，首先需要解决一个有关包容器类的重要问题：当包容器释放一个指向对象的指针时，

会析构该对象吗？例如，当一个包容器对象失去了指针控制，它会析构所有它所指向的对象吗？

15.4.1 所有权问题

所有权问题是普遍关心的问题。对对象进行完全控制的包容器通常无需担心所有权问题，因为它清晰、直接、完全地拥有其包含的对象。但是若包容器内包含指向对象的指针（这种情况在C++中相当普遍，尤其在多态情况下），而这些指针很可能用于程序的其他地方，那么删除了该指针指向的对象会导致在程序的其他地方的指针对已销毁的对象进行引用。为了避免上述情况，在设计和使用包容器时，必须考虑所有权问题。

许多程序都非常简单，一个包容器所包含的指针所指向的对象都仅仅用于包容器本身。在这种情况下，所有权问题简单而直观：该包容器拥有这些指针所指向的对象。由于通常大多数都是上述情况，因此把包容器完全拥有包容器内的指针所指向的对象的情况定义为缺省情形。

处理所有权问题的最好方法是由用户程序员来选择。这常常用构造函数的一个参数来完成，它缺省地指明所有权（对于典型理想化的简单程序）。另外还有读取和设置函数用来查看和修正包容器的所有权。假若包容器内有删除对象的函数，包容器所有权的状态会影响删除，所以我们还可以找到在删除函数中控制析构的选项。我们可以对在包容器中的每一个成员添加所有权信息，这样，每个位置都知道它是否需要被销毁，这是一个引用记数变量，在这里是包容器而不是对象知道所指对象的引用数。

15.4.2 stash模板

“stash”类是一个理想的模板构造的实例,有关它的修改贯穿于本书(最近的见第12章)。下面的例子中，带有所有权操作的循环子也加入其中。

```
//: TSTASH.H -- PSTASH using templates
#ifndef TSTASH_H_
#define TSTASH_H_
#include <stdlib.h>
#include "..\allege.h"
// More convenient than nesting in tstash:
enum owns { no = 0, yes = 1, Default };
// Declaration required:
template<class Type, int sz> class tstashIter;

template<class Type, int chunksize = 20>
class tstash {
  int quantity;
  int next;
  owns own; // Flag
  void inflate(int increase = chunksize);
protected:
  Type** storage;
public:
  tstash(owns owns = yes);
  ~tstash();
  int Owns() const { return own; }
```

```
  void Owns(owns newOwns) { own = newOwns; }
  int add(Type* element);
  int remove(int index, owns d = Default);
  Type* operator[](int index);
  int count() const { return next; }
  friend class tstashIter<Type, chunksize>;
};

template<class Type, int sz = 20>
class tstashIter {
  tstash<Type, sz>& ts;
  int index;
public:
  tstashIter(tstash<Type, sz>& TS)
    : ts(TS), index(0) {}
  tstashIter(const tstashIter& rv)
    : ts(rv.ts), index(rv.index) {}
  // Jump interator forward or backward:
  void forward(int amount) {
    index += amount;
    if(index >= ts.next) index = ts.next -1;
  }
  void backward(int amount) {
    index -= amount;
    if(index < 0) index = 0;
  }
  // Return value of ++ and -- to be
  // used inside conditionals:
  int operator++() {
    if(++index >= ts.next) return 0;
    return 1;
  }
  int operator++(int) { return operator++(); }
  int operator--() {
    if(--index < 0) return 0;
    return 1;
  }
int operator--(int) { return operator--(); }
operator int() {
  return index >= 0 && index < ts.next;
}
Type* operator->() {
  Type* t = ts.storage[index];
  if(t) return t;
  allege(0,"tstashIter::operator->return 0");
```

```
    return 0; // To allow inlining
  }
  // Remove the current element:
  int remove(owns d = Default){
    return ts.remove(index, d);
  }
};

template<class Type, int sz>
tstash<Type, sz>::tstash(owns Owns) : own(Owns) {
  quantity = 0;
  storage = 0;
  next = 0;
}

// Destruction of contained objects:
template<class Type, int sz>
tstash<Type, sz>::~tstash() {
  if(!storage) return;
  if(own == yes)
    for(int i = 0; i < count(); i++)
      delete storage[i];
  free(storage);
}

template<class Type, int sz>
int tstash<Type, sz>::add(Type* element) {
  if(next >= quantity)
    inflate();
  storage[next++] = element;
  return(next - 1); // Index number
}

template<class Type, int sz>
int tstash<Type, sz>::remove(int index,owns d){
  if(index >= next || index < 0)
    return 0;
  switch(d) {
    case Default:
      if(own != yes) break;
    case yes:
      delete storage[index];
    case no:
      storage[index] = 0; // Position is empty
  }
  return 1;
```

```
}

template<class Type, int sz> inline
Type* tstash<Type, sz>::operator[](int index) {
  // No check in shipping application:
  assert(index >= 0 && index < next);
  return storage[index];
}

template<class Type, int sz>
void tstash<Type, sz>::inflate(int increase) {
  void* v =
    realloc(storage,
(quantity+increase)*sizeof(Type*));
  allegemem(v);  // Was it successful?
  storage = (Type**)v;
  quantity += increase;
}
#endif // TSTASH_H_
```

尽管枚举enum owns常常被嵌入在类中，但这里还是将其定义成全局量。这是更方便的使用方法，假若打算观察其效果，我们可以试着把它移进去。

storage指针在类中被置为保护方式，这样通过继承得到的类可以直接访问它。这意味着继承类必然依赖于tstash的特定实现，但是正如我们将在SORTED.CPP 例子(见15.7)中看到的，这样做是值得的。

own标志指明了包容器是否以缺省方式拥有它所包容的对象。如果是这样，存在于包容器中的指针所指向的对象将在析构函数中被相应地销毁。这是一种简单的方法，包容器知道它所包含的类型。可以在构造函数中用重载函数owns()读和修改缺省的所有权。

应该认识到，如果包容器存放的指针是指向基类的，该类型应该具备一个虚析构函数来保证正确地清除派生对象，置于包容器中的派生对象的地址已经被向上映射。

在生存期中tstashIter遵循着与单个包容器相结合的循环子模式。另外拷贝构造函数允许我们创建新的循环子，指向已存在的循环子所指向的位置，这样可以非常高效地在包容器中创建书签。forward() 和 backward()成员函数允许移动循环子几步，它和包容器边界有关。增量和减量的重载运算符可以移动循环子一个位置。循环子所涉及的元素常常用灵活的指针来对其操作，通过调用包容器中的remove()函数可完成对当前对象的消除。

下面的例子是创建和检测两个不同的 tstash对象，一个属于新类Int并在它的析构函数和构造函数中给出报告，而另一个含有属于第12章的String类的对象：

```
//: TSTEST.CPP -- Test TSTASH
#include <fstream.h>
#include "..\allege.h"
#include "..\14\tstash.h"
#include "..\11\strings.h"
const bufsize = 80;
ofstream out("tstest.out");
```

```
class Int {
  int i;
public:
  Int(int I = 0) : i(I) {
    out << ">" << i << endl;
  }
  ~Int() { out << "~" << i << endl; }
  operator int() const { return i; }
  friend ostream&
    operator<<(ostream& os, const Int& x) {
      return os << x.i;
  }
};

main() {
  tstash<Int> intStash; // Instantiate for int
  for(int i = 0; i < 30; i++)
    intStash.add(new Int(i));
  tstashIter<Int> Intit(intStash);
  Intit.forward(5);
  for(int j = 0; j < 20; j++, Intit++)
    Intit.remove(); // Default removal
  for(int k = 0; k < intStash.count(); k++)
    if(intStash[k]) // Remove() causes "holes"
      out << *intStash[k] << endl;

  ifstream file("tstest.cpp");
  allegefile(file);
  char buf[bufsize];
  // Instantiate for String:
tstash<String> stringStash;
while(file.getline(buf, bufsize))
  stringStash.add(makeString(buf));
for(int u = 0; u < stringStash.count(); u++)
  if(stringStash[u])
    out << *stringStash[u] << endl;
tstashIter<String> it(stringStash);
int j = 25;
it.forward(j);
while(it) {
  out << j++ << ": " << it->str() << endl;
  it++;
}
}
```

在两种情形中，都创建了循环子，用来在包容器中前后移动。请注意使用构造函数所产生

的优美效果：我们无需关心使用数组的实现细节。我们告诉包容器和循环子做什么，而不是怎么做，这将使得问题的解更容易形成概念，更容易建立和更容易修改。

15.4.3 stack模板

在第13章中的stack类，它既是一个包容器，也是一个带有相关循环子的模板。下面是新的头文件：

```
//: TSTACK.H -- Stack using templates
#ifndef TSTACK_H_
#define TSTACK_H_

// Declaration required:
template<class T> class tstackIterator;

template<class T>
class tstack {
  struct link {
    T* data;
    link* next;
    link(T* Data, link* Next) {
      data = Data;
      next = Next;
    }
  } * head;
  int owns;
public:
  tstack(int Owns = 1) : head(0), owns(Owns) {}
  ~tstack();
  void push(T* Data) {
    head = new link(Data,head);
  }
  T* peek() const { return head->data; }
  T* pop();
  int Owns() const { return owns; }
  void Owns(int newownership) {
    owns = newownership;
  }
  friend class tstackIterator<T>;
};

template<class T>
T* tstack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  link* oldHead = head;
```

```
  head = head->next;
  delete oldHead;
  return result;
}

template<class T>
tstack<T>::~tstack() {
  link* cursor = head;
  while(head) {
    cursor = cursor->next;
    // Conditional cleanup of data:
    if(owns) delete head->data;
    delete head;
    head = cursor;
  }
}

template<class T>
class tstackIterator {
  tstack<T>::link* p;
public:
  tstackIterator(const tstack<T>& tl)
    : p(tl.head) {}
  tstackIterator(const tstackIterator& tl)
    : p(tl.p) {}
  // operator++ returns boolean indicating end:
  int operator++() {
    if(p->next)
      p = p->next;
    else p = 0; // Indicates end of list
    return int(p);
  }
  int operator++(int) { return operator++(); }
  // Smart pointer:
  T* operator->() const {
    if(!p) return 0;
    return p->data;
  }
  T* current() const {
    if(!p) return 0;
    return p->data;
  }
  // int conversion for conditional test:
  operator int() const { return p ? 1 : 0; }
};
#endif // TSTACK_H_
```

我们可以注意到，这个类已被修改，能支持所有权处理，因为该类可以识别出确切的类型（或至少是基类型，它在运作中使用了虚析构函数）。如同tstash的情形一样，缺省方式是包容器销毁它的对象，但我们可以通过修改析构函数的参数或者通过用 Owns()对成员函数进行读写以改变这种缺省方式。

循环子是非常简单的小规模指示器。当创建一个 tstackIterator时，它从链表的头开始，在链表中只能向前推进。假若打算重新从头部启动，可以创建一个新的循环子；假若要记住链表中的某位置，可以从指向该位置的已生成的循环子处创建一个新的循环子（使用拷贝构造函数）。

为了对循环子所指的对象调用函数，我们可以使用灵巧指针（循环子的通常方法）或使用被称为current()的函数，该函数看上去和灵巧指针相同，因为它返回一个指向当前对象的指针，但它们是不同的，因为灵巧指针执行逆向引用的外部层次（见第 11章）。最后，operator int()指出，是否我们已处在表的尾部和是否允许在条件语句中使用该循环子。

完整的实现包含在这个头文件中，所以这里没有单独的 CPP文件。下面是循环子检测和练习的小例子：

```
//: TSTKTST.CPP -- Use template list & iterator
#include "..\14\tstack.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

main() {
  ifstream file("tstktst.cpp");
  allegefile(file);
  const bufsize = 100;
  char buf[bufsize];
  tstack<String> textlines;
  // Read file and store lines in the list:
  while(file.getline(buf,bufsize))
    textlines.push(String::make(buf));
  int i = 0;
  // Use iterator to print lines from the list:
  tstackIterator<String> it(textlines);
  tstackIterator<String>* it2 = 0;
  while(it) {
    cout << *it.current() << endl;
    it++;
    if(++i == 10) // Remember 10th line
      it2 = new tstackIterator<String>(it);
  }
  cout << *(it2->current()) << endl;
  delete it2;
}
```

tstack被实例化为存放String对象并且填充了来自某文件的一些行。然后循环子被创建，用

来在被链接的表中移动。第十行用拷贝构造函数由第一个循环子产生第二循环子，以后，这一行被打印，动态创建的循环子被销毁。这里，动态对象的创建被用于控制对象的生命周期。

这和先前的stack类测试例子十分相似，但是现在所包含的对象能随 tstack的销毁而适当地被销毁。

15.5 字符串和整型

为了在本章的剩余部分进一步改进这些例子，有必要引入功能强大的字符串类，它与整数对象一起来保证初始化。

15.5.1 栈上的字符串

这里是一个更加完全的字符串类，在这本书之前该类已被使用。另外，它使用了模板，添加了一个特殊的特性：对SString实例化时，我们能够决定它存在于堆上还是栈上。

```
//: SSTRING.H -- Stack-based string
#ifndef SSTRING_H_
#define SSTRING_H_
#include <string.h>
#include <iostream.h>

template<int bsz = 0>
class SString {
  char buf[bsz + 1];
  char* s;
public:
  SString(const char* S = "") : s(buf) {
    if(!bsz) { // Make on heap
      s = new char[strlen(S) + 1];
      strcpy(s, S);
    } else { // Make on stack
      buf[bsz] = 0; // Ensure 0 termination
      strncpy(s, S, bsz);
    }
  }
  SString(const SString& rv) : s(buf) {
    if(!bsz) { // Make on heap
      s = new char[strlen(rv.s) + 1];
    strcpy(s, rv.s);
  } else { // Make on stack
    buf[bsz] = 0;
    strncpy(s, rv.s, bsz);
  }
}
SString& operator=(const SString& rv) {
  // Check for self-assignment:
  if(&rv == this) return *this;
```

```
    if(!bsz) { // Manage heap:
      delete s;
      s = new char[strlen(rv.s) + 1];
    }
    // Constructor guarantees length < bsz:
    strcpy(s, rv.s);
    return *this;
  }
  ~SString() {
    if(!bsz) delete []s;
  }
  int operator==(const SString& rv) const {
    return !stricmp(s, rv.s);
  }
  int operator!=(const SString& rv) const {
    return stricmp(s, rv.s);
  }
  int operator>(const SString& rv) const {
    return stricmp(s, rv.s) > 0;
  }
  int operator<(const SString& rv) const {
    return stricmp(s, rv.s) < 0;
  }
  char* str() const { return s; }
  friend ostream&
    operator<<(ostream& os,
               const SString<bsz>& S) {
      return os << S.s;
    }
  };

  typedef SString<> Hstring; // Heap string
  #endif // SSTRING_H_
```

通过使用typedef Hstring，我们可以得到一个普通的基于堆的字符串（使用 typedef而不是使用继承是因为继承需要重新创建构造函数和重载符 =）。但是，假若关心的是生成和销毁许多字符串时的效率，我们可以冒险设定所涉及问题的解的字符最大可能长度。如给出了模板的长度参数，它可以自动地在栈上而不是在堆上创建对象，这意味着每个对象的 new和delete的开销将被忽略。我们能发现运算符=也被提高了运行速度。

字符串的比较运算符使用了称之为 stricmp()的函数，虽然它不是标准 C函数，但却能为大多数编译器的库所认可。它执行字符串比较时会忽略字母大小写。

15.5.2 整型

类integer的构造函数会赋零值，它包含一个自动将类型转换成 int型的运算符，所以可以容

易地提取数据：

```
//: INTEGER.H -- Int wrapped in a class
#ifndef INTEGER_H_
#define INTEGER_H_
#include <iostream.h>

class integer {
  int i;
public:
  // Guaranteed zeroing:
  integer(int ii = 0) : i(ii) {}
  operator int() const { return i; }
  const integer& operator++() {
    i++;
    return *this;
  }
  const integer operator++(int) {
    integer returnval(i);
    i++;
    return returnval;
  }
  integer& operator+=(const integer& x) {
    i += x.i;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const integer& x) {
      return os << x.i;
  }
};
#endif // INTEGER_H_
```

虽然这个类相当小（它仅仅满足本章的需要），但我们可以方便地遵循第 11章中的例子而添加很多我们所需要的运算。

15.6 向量

虽然tstash的表现有一点和向量类似，但是创建一个和向量一样的类是很方便的，也就是，它的仅有的行为是索引。（因为它仅有的接口是operator[].）

15.6.1 “无穷”向量

下面的向量类仅仅拥有指针。它从不需要调整大小：我们可以简单地对任何位置进行索引，这些位置可魔术般地变化，而无需提前通知向量类。 operator[]能返回一个指针的引用，所以可以出现在=的左边（它可以是一个左值，也可以是一个右值）。向量类仅仅与指针打交道，所以它工作的对象没有类型限制，对于类型的行为没有事先假定的必要。

下面是它的头文件：

```
//: VECTOR.H -- "Infinite" vector
#ifndef VECTOR_H_
#define VECTOR_H_
#include <stdlib.h>
#include "..\allege.h"

template<class T>
class vector {
  T** pos;
  int pos_sz;
  T** neg;
  int neg_sz;
  int owns;
  enum {
    chunk = 20, // Min allocation increase
    esz = sizeof(T*), // Element size
  };
  void expand(T**& array,int& size,int index);
public:
  vector(int Owns = 1);
  ~vector();
  T*& operator[](int index);
  int Owns() const { return owns; }
  void Owns(int newOwns) { owns = newOwns; }
};
template<class T>
vector<T>::vector(int Owns)
  : pos(0), pos_sz(0),
    neg(0), neg_sz(0),
    owns(Owns) {}

template<class T>
vector<T>::~vector() {
  if(owns)
    for(int i = 0; i < pos_sz; i++)
      delete pos[i];
  free(pos);
  if(owns)
    for(int j = 0; j < neg_sz; j++)
      delete neg[j];
  free(neg);
}

template<class T>
T*& vector<T>::operator[](int index) {
```

```
    if(index < 0) {
      index *= -1;
      if(index >= neg_sz)
        expand(neg, neg_sz, index);
      return neg[index];
    }
    else { // Index >= 0
      if(index >= pos_sz)
        expand(pos, pos_sz, index);
      return pos[index];
    }
  }

  template<class T> void
  vector<T>::expand(T**& array, int& size,
                    int index) {
    const newsize = index + chunk;
    const increment = newsize - size;
    void* v = realloc(array, newsize * esz);
    allegemem(v);
    array = (T**)v;
    memset(&array[size], 0, increment * esz);
    size = index + chunk;
  }
  #endif // VECTOR_H_
```

为了易于实现，向量被分成正和负两个部分，我们可以出于某些原因改变下面的实现使相邻的存储空间得以利用。

当增加存储单元时，将使用私有函数expand()。expand()采用的参数是引用T**&而不是一个指针。这是因为在realloc()之后，它必须改变外部参数以指向一个新的物理地址。另外，还需要参数size和index，以便于对新的单元赋零。（这一点是重要的，因为如果向量拥有对象，析构函数会对所有的指针调用delete。）size以int&进行传递，因为它也必须改变以反映新的存储长度。

在operator[]中，不管是向量内的正或负的部分，若索取一个在当前使用的存储空间之外的存储位置，那么更多的存储空间会被分配。这比内置的数组更加方便，并且创建也无需为存储长度的大小而担心。

```
  //: VECTOR.CPP -- Test "infinite" vector
  #include <fstream.h>
  #include "..\allege.h"
  #include "..\14\vector.h"
  #include "..\14\sstring.h"
  typedef SString<40> String;

  main() {
  ifstream source("vector.cpp");
  allegefile(source);
  const bsz = 255;
```

```
  char buf[bsz];
  vector<String> words;
  int i = 0;
  while(source.getline(buf, bsz)) {
    char* s = strtok(buf, " \t");
    while(s) {
      words[i++] = new String(s);
      s = strtok(0, " \t");
    }
    words[i++] = new String("\n");
  }
  for(int j = 0; words[j]; j++)
    cout << *words[j] << ' ';
}
```

本程序使用了标准C函数strtok()，它取字符缓冲区的起始地址（第一个参数）和寻找定界符（第二个参数）。它用零来代换定界符并返回以标志为起始的地址。假若在随后的时间以第一参数为零的形式来调用它，它将从剩余的字符串中继续抽取直至最后。在上面的例子中，是以空格和制表符为定界符来抽取字词的。每个字词被返回进 String中，而每个指针被存放进words向量中，它最终能以拆分成字词的方式而包含整个文件。

15.6.2 集合

一个集合的约束条件是它的元素不重复。我们可以向一个集合添加元素，也可以测试一下某个元素是否是集合中的成员。下面的集合类使用了一个包含其元素的向量：

```
//: SET.H -- Each entry in a set is unique
#ifndef SET_H_
#define SET_H_
#include "..\14\vector.h"
#include <assert.h>

template<class Type>
class set {
  vector<Type> elem;
  int max;
  int lastindex; // Efficiency tool
  int within(const Type& e) {
    // Requires Type::operator== :
    for(lastindex = 0; lastindex < max;
        lastindex++)
      if(elem[lastindex]->operator==(e))
        return lastindex;
    return -1;
  }
  // Prevent assignment & copy-construction:
```

```
  void operator=(set&);
  set(set&);
public:
  set() : max(0), lastindex(0) {}
  void add(const Type&);
  int contains(const Type&);
  // Where is it in the set?:
  int index(const Type& e);
  Type& operator[] (int index) {
    // No check for shipping application:
    assert(index >= 0 && index < max);
    return *elem[index];
  }
  int length() const { return max; }
};

template<class Type> void
set<Type>::add(const Type& e) {
  if(!contains(e)) {
    elem[max] = new Type(e);//Copy-constructor
    max++;
  }
}

template<class Type> int
set<Type>::contains(const Type& e) {
  return within(e) != -1;
}

template<class Type> int
set<Type>::index(const Type& e) {
  // Prevent a new search if possible:
  if(elem[lastindex]->operator!=(e)) {
    int ind = within(e);
    assert(ind != -1); // Must know it's inside
  }
  return lastindex;
}
#endif // SET_H_
```

add()在检测确定一个新元素不在集合后，将其追加入集合中。 contains()告诉我们对象是否已存在于集合之中， index()告诉我们对象在集合之中的位置。可以使用 operator[]来检索它。length()告诉我们集合中有多少个元素。

上面的例子中始终记着数组元素的数量。作为提高的手段，最后被检索的索引元素会被保存下来，所以index()直接跟随在contains()之后将不会有两次对集合的遍历运算。内联的私有函数within()使实施更为容易。

下面的验证例子使用了集合类而生成一个字词索引，它由存于某文件中的一批字词组成。

```
//: SETTEST.CPP -- Test the "set" class
// Creates a concordance of text words
#include <fstream.h>
#include "..\14\set.h"
#include "..\14\sstring.h"
#include "..\allege.h"
const char* delimiters =
  " \t;()\"<>:{}[]+-=&*#.,/\\"
  "0123456789";

typedef SString<40> String;

main(int argc, char* argv[]) {
  allege(argc == 2, "need file argument");
  ifstream in(argv[1]);
  allegefile(in);
  ofstream out("settest.out");
  set<String> concordance;
  const sz = 255;
  char buf[sz];
  while(in.getline(buf, sz)) {
    // Capture individual words:
    char* s = strtok(buf, delimiters);
    while(s) {
      // Contains 1 entry per unique word:
      concordance.add(s); // Auto type conv.
      s = strtok(0, delimiters);
    }
  }
  for(int i = 0; i < concordance.length(); i++) {
    out << concordance[i] << endl;
  }
}
```

这个程序再次使用了strtok()，但这次的定界符则更多，此外，亦删除了尾部字符及其编号。

注意到由于 add()所希望接受的对象是字符串型的，而对 char*型的类型转换是由使用SString<40>构造函数来自动完成的，该构造函数会取得一个 char*型参数。这样会产生一个临时对象，该对象的地址可传递给 add()，假若add()在表中未发现该临时对象，就会复制它将其加入表中。add()的参数传递使用了对象引用而非指针的方式，这是重要的一点，因为在这里指针没有必要置于其中，假若使用new及运算指针将会最终失去指针。

15.6.3 关联数组

一个普通的数组使用一个整型数值作为某类型序列元素的下标。在一般情况下，若想使用

任意类型为数组下标和其他任意类型为元素，这就是模板的一个理想情形。关联数组可以使用任何类型作为元素的下标。

这里展示的关联数组使用了集合类和向量类创建了关于指针的两个数组：一个为输入类型，另一个为输出类型。假若采用了一个以前未遇到的下标，它会创建一个该下标的副本（假定输入类型具有拷贝构造函数和operator=）作为新的输入对象并且生成一个新的使用缺省构造函数的输出对象。operator[]仅仅返回一个引用给输出对象，所以我们可以使用它来完成运算。也可以用一整型参数调用in_value() 和out_value()函数去产生输入和输出的数组的所有元素：

```
//: ASSOC.H -- Associative array
#ifndef ASSOC_H_
#define ASSOC_H_
#include "..\14\set.h"
#include <assert.h>

template<class In, class Out>
class assoc_array {
  set<In> inVal;
  vector<Out> outVal;
  int max;
  // Prevent assignment & copy-construction:
  void operator=(assoc_array&);
  assoc_array(assoc_array&);
public:
  assoc_array() : max(0) {}
  Out& operator[](const In&);
  int length() const { return max; }
  In& in_value(int i) {
    // No check for shipping application:
    assert(i >= 0 && i < max);
    return inVal[i];
  }
  Out& out_value(int i) {
    assert(i >= 0 && i < max);
    return *outVal[i];
  }
};

template<class In, class Out> Out&
assoc_array<In,Out>::operator[](const In& in){
  if(!inVal.contains(in)) {
    inVal.add(in); // Copy-constructor
    outVal[max] = new Out; // Default constr.
    max++;
  }
  int x = inVal.index(in);
  return *outVal[inVal.index(in)];
```

```
}
#endif // ASSOC_H_
```

关联数组的一个经典的程序实例是对一文件进行字数统计，这是一个相当简单但很实用的工具。关联数组的输出值不能是内部数据类型（built-in），这是关联数组的限制之一。因为内部数据类型没有缺省构造函数，在创建时不能初始化。为了在字数统计程序中解决该问题，整数类和SString类一同用于关联数组的检测：

```
//: ASSOC.CPP -- Test of associative array
#include "..\14\assoc.h"
#include "..\14\sstring.h"
#include "..\14\integer.h"
#include "..\allege.h"
#include <fstream.h>
#include <ctype.h>

main() {
  const char* delimiters =
    " \t;()\"<>:{}[]+-=&*#.,/\\";
  assoc_array<SString<80>, integer> strcount;
  ifstream source("assoc.cpp");
  allegefile(source);
  ofstream out("assoc.out");
  allegefile(out);
  const bsz = 255;
  char buf[bsz];
  while(source.getline(buf, bsz)) {
    char* s = strtok(buf, delimiters);
    while(s) {
      strcount[s]++; // Count word
      s = strtok(0, delimiters);
    }
  }
  for(int i = 0; i < strcount.length(); i++) {
    out << strcount.in_value(i) << " : "
        << strcount.out_value(i) << endl;
  }

  // The "shopping list" problem:
  assoc_array<SString<>, integer> shoplist;
  ifstream list("shoplist.txt");
  allegefile(list);
  ofstream olist("shoplist.out");
  allegefile(olist);
  while(list.getline(buf, bsz)) {
    int i = strlen(buf) - 1; // Last char
```

```
    while(isspace(buf[i]))
      i--; // Find nonzero char at end
    while(!isspace(buf[i]))
      i--; // Back up to space or tab
    int count = atoi(&buf[i+1]); // Use value
    while(isspace(buf[i]))
      i--; // Back up to non-whitespace
    buf[i+1] = 0; // Mark end of description
    i = 0;
    while(isspace(buf[i]))
      i++; // Find start of first word
    shoplist[&buf[i]] += count;
  }
  for(int j = 0; j < shoplist.length(); j++) {
    olist << shoplist.in_value(j) << " : "
          << shoplist.out_value(j) << endl;
  }
}
```

该程序中关键的一行是：

```
strcount[s]++;//count word
```

由于s是char*型参数，而operator[]所希望的是SString<80>型，编译器可生成一个临时对象传递给使用构造函数SString<80>(char*)的operator[]。该临时对象被创建于栈上，所以它可以快速生成和销毁。

operator[]返回一个integer&给assoc_array中相应的对象，这样我们可以赋于对象任何行为。这里，简单地通过累加运算来说明发现了另一个单词。主程序的后面部分用于解决传统“货单”问题。货单文件的每一行都包含项目（名称可能用空格分开）和数量。这些项目可能在表中不止一次出现，关联数组可对其进行统计。在这些代码中 C的标准宏isspace()的使用作为对环境空间的额外补偿。下面代码行用于统计的实现：

```
shoplist[&buf[i]] +=count;
```

和以前一样， char*的内部检索会产生一个SString对象，因为这是索引运算符预期搜寻的对象，类型间的转化由构造函数完成。如果我们跟踪该程序就会发现由这个临时对象引起的构造函数和析构函数的调用。

15.7 模板和继承

没有东西能妨碍我们采用在普通类中的方法来使用类模板。例如我们能很容易地从一个模板中获得继承，可以从已存在的模板中继承和加以实例化从而创建一个新的模板。尽管 tstash类在前述中已经可满足我们的要求，现在当我们希望它追加自排序功能时，可以方便地对其进行代码重用和数值添加：

```
//: SORTED.H -- Template inheritance
#ifndef SORTED_H_
#define SORTED_H_
#include <stdlib.h>
```

```
#include <string.h>
#include <time.h>
#include "..\14\tstash.h"
#include "..\14\set.h"
template<class T>
class sorted :  public tstash<T> {
  void bubblesort();
public:
  int add(T* element) {
    tstash<T>::add(element);
    bubblesort();
    return 0; // Sort moves the element
  }
};
template<class T>
void sorted<T>::bubblesort() {
  for(int i = count(); i > 0; i--)
    for(int j = 1; j < i; j++)
      if(*storage[j-1] > *storage[j]) {
        // Swap the two elements:
        T* t = storage[j-1];
        storage[j-1] = storage[j];
        storage[j] = t;
      }
}

// Quick & dirty sorted set:
template<class T>
class sortedSet : public set<T> {
  sorted<T> Sorted;
public:
  void add(T& e) {
    if(contains(e)) return;
    set<T>::add(e);
    Sorted.add(new T(e));
  }
  T& operator[] (int index) {
    assert(index >= 0 && index < length());
    assert(Sorted[index]);
    return *Sorted[index];
  }
  int length() {
    return Sorted.count();
  }
};
```

```
// Unique random number generator:
template<int upper_bound>
class urand {
  int map[upper_bound];
  int recycle;
public:
  urand(int Recycle = 0);
  int operator()();
};

template<int upper_bound>
urand<upper_bound>::urand(int Recycle = 0)
  : recycle(Recycle) {
  memset(map, 0, upper_bound * sizeof(int));
  // Seed the random number generator:
  time_t t;
  srand((unsigned)time(&t));
}

template<int upper_bound>
int urand<upper_bound>::operator()() {
  if(!memchr(map, 0, upper_bound)) {
    if(recycle)
      memset(map, 0,
             sizeof(map) * sizeof(int));
    else
      return -1; // No more spaces left
  }
  int newval;
  while(map[newval = rand() % upper_bound])
    ; // Until unique value is found
  map[newval]++; // Set flag
  return newval;
}
#endif // SORTED_H_
```

本例子包含了一个随机数发生器类，它可产生唯一数和重载 operator()以便使用一般的函数调用语句。urand的唯一性是由保存随机数空间的所有可能的数的影（map）象而产生的（随机数空间的上界由模板参数设置），并标记每一个已使用的为关闭。构造函数的第二个可选参数的作用是允许我们在界内随机数用完的情况下可以重用这些数。注意，为了优化运行速度我们将影象定义成固定完整数组，而不论我们需要多少数。假若我们打算优化数组长度，可以这样改动下面的实现：把map安排成动态申请存储方式；把随机数本身送入 map而不是置标志，这样的改变不会影响任何客户代码。

模板sorted为所有由它实例化而产生的类施加一个约束：它们必须包含一个 >运算符。在SString中，这种施加是明显的，但是在 integer中，自动类型转换运算符 int()提供了一个内置>

运算符的途径。当模板提供更多的功能时，通常要对类赋予更多的需求，。有时不得不继承被包含的类以增加必要的功能。注意使用重载运算符的价值： integer类所提供的功能依赖于它的底层实现。

在例中，可以看到在tstash中的storage以保护方式而非私有方式定义的好处。这对于让类sorted知道很重要，这是真正的依赖性。假若改变 tstash的下层实现的一些东西而非一个数组，如链表，冒泡排序中的元素交换就会和原来完全不同，于是从属类 sorted也需要改变。然而，一个更好的选择是（假若有可能的话），对让tstash实现采用保护方法的一种更好的替代是提供足够的保护接口函数，这样一来，访问和交换可在派生类中完成而无需涉及底层实现。这种方法仍然可以改变下层实现，但不会传播这些修改。

注意，附加类sortedSet说明怎样可以快速地由已存在的类中粘贴所需的功能。 sortedSet可从set类中获取接口，而且当添加一个新元素到set类时，也同时为其添加这个元素到sorted对象，所返回的值全是排序过的。我们可能会考虑设计一个更为简化、有效的类版本，在这里就不再详述了（标准C++模板库包含一个可自排序的集合类）。下面是对SORTED.H的测试：

```
//: SORTED.CPP -- Testing template inheritance
#include "..\14\sorted.h"
#include "..\14\sstring.h"
#include "..\14\integer.h"
typedef SString<40> String;

char* words[] = {
  "is", "running", "big", "dog", "a",
};
const wordsz = sizeof words / sizeof *words;

main() {
  sorted<String> ss;
  for(int i = 0; i < wordsz; i++)
    ss.add(new String(words[i]));
  for(int j = 0; j < ss.count(); j++)
    cout << ss[j]->str() << endl;
  sorted<integer> is;
  urand<47> rand1;
  for(int k = 0; k < 15; k++)
    is.add(new integer(rand1()));
  for(int l = 0; l < is.count(); l++)
    cout << *is[l] << endl;
}
```

该例通过创建一个排序数组来验证SString 和interger类。

15.7.1 设计和效率

在sorted中，每次调用add()时，新元素都将被插入，数组也重新排序。这里使用的冒泡排序方法效率低下，不提倡使用（但它易于理解和编码）。由于冒泡排序方法是私有实现的一部分，在这里是相当合适的。在我们开发程序时，一般的步骤是：

1) 使类接口正确。
2) 尽量迅速、准确地实现原型。
3) 验证我们的设计。

常常，仅当我们集成工作系统的初期“草稿”时才会发现类接口问题，这种情况非常普遍。在系统集成和初次实现期间，我们还可能发现需要“帮助者”类，如同对包容器和循环子的需要一样。有时在系统分析期间很难发现上述问题（在分析中我们的目标是得到能快速实现和检测的全貌设计）。只有在设计被验证后，我们才有必要花时间对其进行完全刷新和考虑性能要求。假如设计失败或者性能要求不需考虑，则冒泡排序方法就不错了，没有必要进一步浪费时间。（当然，一个理想的解决方案是利用别人的已被证实的排序包容器；首先应留意标准C++的模板库）。

15.7.2 防止模板膨胀

每次模板实例化，其中的代码都会重新生成（其中的内联函数除外）。假若模板内的一些功能并不依赖定义类型，我们可以把它们放入一个通用基类以避免不必要的代码重新生成。例如在第13章的INHSTAK.CPP(见13.5.2)中的继承被用于定义stack所能接受和产生的类型。下面是一个模板化的版本代码：

```
//: NOBLOAT.H -- Templatized INHSTAK.CPP
#ifndef NOBLOAT_H_
#define NOBLOAT_H_
#include "..\11\stack11.h"

template<class T>
class nbstack : public stack {
public:
  void push(T* str) {
    stack::push(str);
  }
  T* peek() const {
    return (T*)stack::peek();
  }
  T* pop() {
    return (T*)stack::pop();
  }
  ~nbstack();
};

// Defaults to heap objects & ownership:
template<class T>
nbstack<T>::~nbstack() {
  T* top = pop();
  while(top) {
    delete top;
    top = pop();
  }
}
#endif // NOBLOAT_H_
```

在以前，内联函数不产生代码，而是通过仅一次性地创建一个基类代码提供其功能。但是所有权问题则可以通过加入一个析构函数来解决 (它依赖类型，必须由模板来构造)，在这里的所有权是缺省的。注意，当基类析构函数被调用时，栈将被清空，所以不会出现重复释放问题。

15.8 多态性和包容器

多态性、动态对象生成和包容器在一个真正的面向对象程序中和谐地被利用，这是很普遍的。动态对象生成和包容器所解决的问题在于设计初期我们可能不知道需要多少对象，需要什么类型的对象，这是因为包容器可持有指向基类对象的指针，每逢我们把派生类指针放入包容器，会发生向上映射（具有相应的代码组织和可扩展性的好处）。下面的例子有点像垃圾回收的工作过程，首先所有的垃圾被放入一个垃圾箱中，然后分类放入不同的箱中，它有一个函数用于遍历垃圾箱并估算出其中什么有价值。这里的垃圾回收模拟实现并不完美，在第 18章说明“运行时类型识别(RTTI) ”时，再对该例进一步介绍。

```
//: RECYCLE.CPP -- Containers & polymorphism
#include <fstream.h>
#include <stdlib.h>
#include <time.h>
#include "..\14\tstack.h"
ofstream out("recycle.out");

enum type { Aluminum, Paper, Glass };

class trash {
  float Weight;
public:
  trash(float Wt) : Weight(Wt) {}
  virtual type trashType() const = 0;
  virtual const char* name() const = 0;
  virtual float value() const = 0;
  float weight() const { return Weight; }
  virtual ~trash() {}
};

class aluminum : public trash {
  static float val;
public:
  aluminum(float Wt) : trash(Wt) {}
  type trashType() const { return Aluminum; }
  virtual const char* name() const {
    return "aluminum";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
```

```
  }
};

float aluminum::val = 1.67;

class paper : public trash {
  static float val;
public:
  paper(float Wt) : trash(Wt) {}
  type trashType() const { return Paper; }
  virtual const char* name() const {
    return "paper";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float paper::val = 0.10;

class glass : public trash {
  static float val;
public:
  glass(float Wt) : trash(Wt) {}
  type trashType() const { return Glass; }
  virtual const char* name() const {
    return "glass";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float glass::val = 0.23;

// Sums up the value of the trash in a bin:
void SumValue(const tstack<trash>& bin,ostream& os){
  tstackIterator<trash> tally(bin);
  float val = 0;
  while(tally) {
    val += tally->weight() * tally->value();
    os << "weight of " << tally->name()
       << " = " << tally->weight() << endl;
```

```
    tally++;
  }
  os << "Total value = " << val << endl;
}

main() {
  // Seed the random number generator
  time_t t;
srand((unsigned)time(&t));

tstack<trash> bin; // Default to ownership
// Fill up the trash bin:
for(int i = 0; i < 30; i++)
  switch(rand() % 3) {
    case 0 :
      bin.push(new aluminum(rand() % 100));
      break;
    case 1 :
      bin.push(new paper(rand() % 100));
      break;
    case 2 :
      bin.push(new glass(rand() % 100));
      break;
  }
// Bins to sort into:
tstack<trash> glassbin(0); // No ownership
tstack<trash> paperbin(0);
tstack<trash> ALbin(0);
tstackIterator<trash> sorter(bin);
// Sort the trash:
// (RTTI offers a nicer solution)
while(sorter) {
  // Smart pointer call:
  switch(sorter->trashType()) {
    case Aluminum:
      ALbin.push(sorter.current());
      break;
    case Paper:
      paperbin.push(sorter.current());
      break;
    case Glass:
      glassbin.push(sorter.current());
      break;
  }
  sorter++;
```

```
    }
    SumValue(ALbin, out);
    SumValue(paperbin, out);
    SumValue(glassbin, out);
    SumValue(bin, out);
}
```

这里使用了基类中的虚函数结构，这些函数将在派生类中得到再定义。由于包容器 tstack是trash的实例化，所以它含有trash指针，这些指针是指向基类的。然而， tstack也会含有指向trash的派生类对象的指针，正如调用push()所看到的那样。随着这些指针的加入，它们会失去本来的特定身份而变成trash的指针。然而由于多态性，当通过tally 和 sorter循环调用虚函数时，与要求相适应的行为仍会发生。（注意，使用循环子的灵巧指针会导致虚函数的调用。）

trash类还包含一个虚析构函数，向任何类中自动增加内容都应当利用虚函数。当bin包容器超出了相应范围时，包容器的析构函数会为它所包容的所有对象调用虚析构函数，进行有效的清除。

由于包容器类模板一般很少有所见到的“普通”类的向下继承和向上映射，所以我们几乎不会看到在这些类中存在虚函数，它们的重用是以模板方式而非继承方式。

15.9 包容器类型

这里所采用的一系列包容器类大约和“数据结构”类工具相当（当然包容器由于具备一些相关联的功能因而比数据结构更丰富）。包容器将其功能和数据结构打包集成在一起，它能更为自然地表达“数据结构”的概念。

虽然包容器可能需要特定的行为（如在栈尾压入和抛出元素），但是通过使用较通用的循环子便可获得更大的自由度。下列的大多数类型都很容易支持关联循环子。

所有的包容器都允许放入和取出一些东西。它们的不同在于其功能用途，而有一些包容器的相异之处则仅仅在于存放内容的类型不同。它们的不同在于访问速度：一些易于线性访问，但在序列中间插入元素时，时间开销却较高。其他的一些在中间插入元素时时间开销较低，但线性访问的开销却较高。如果要在这两种情形下做出取舍，应首先着眼于最可通融的方法。如果在程序运行后，发现速度的通融性较差则需要进一步优化，优化成更加高效的方法。

下面是存在于C++标准模板库STL（standard template library）中的包容器子集， STL将在附录A中讨论：

袋子：项目的（可能有重复）集合。它的元素没有特定的次序要求， STL不包含它，这是因为其功能可由表和向量来实现。

集合：不允许有重复元素的袋子就是集合。在 STL中，集合是以一种联合包容器(associative container)来描述的，联合包容器可以根据关键字向包容器提供和从包容器中取回数据元素。在集合中数据元素的存取检索关键字就是元素本身。一个 STL的多重集合允许将同一关键值的许多副本放入不同的对象中。

向量：可索引的项目序列。由于它有一致的访问时间，所以可作为缺省选择。

队列：从尾部追加元素，从头部删除元素的项目序列。它是双端队列的子集，有时不实现它们，但我们可以用双端队列来代替。

双端队列：具有两个端部的队列。序列中项目元素的追加和删除可从任意一端进行。在大部分的插入和删除运算发生在序列头部或尾部时，可用其代替列表以提高效率。在头部或尾部做插入和删除运算时，双端队列的时间复杂度为常量级，但在序列中部实施运算时呈线性时间

复杂度。

栈：在相同一端实施追加和删除的项目序列。尽管在本书中被用作一个例子，其实它是双端队列功能的一个子集。

环形队列：环形结构的项目序列，元素的追加和删除位于环形结构的顶端，它是头部和尾部关联的队列。通常让其支持系统底层活动，非常有效而且其时间复杂度为常量级。例如在一个通讯中断服务例程中，插入一些字符而随后将其删除，将不用担心存储的耗尽及必须的时间分配。但是，如不加以细致地编程，环形队列会超出正常的控制范围。 STL不包含环形队列。

列表：允许以相等的时间在任意点实施插入和删除的有根的项目序列。它不提供快速随机访问，但能在序列中间进行快速插入和删除运算。列表一般以单链表和双链表的形式来实现。单链表的遍历是单方向的，而双链表可在任意节点上向前向后移动。由于单链表仅包含一个指向下一个节点的指针而双链表则包含前趋和后继两个指针，所以单链表较双链表的存储开销低。但是在插入和删除工作效能上，双链表则优于单链表。

字典：关键字和相应值的映射，在STL中被称为“ 映像 ”（它是联合包容器的另一种形式）。关键字和相应值的映射对有时被称为“ 联系 ”。字典值的存取和关键字是相关联的。 STL也提供多重映射，允许多重映射相同关键字到相应值的多个副本上，这和哈希表相当。

树：存在一个根节点的节点和弧（节点间的连接）的集合。树不包含环（没有封闭的路径）和交叉路径。STL中不提供树，树具有的特性功能由STL的其他类型来提供。

二叉树：一个普通树从每个节点上射出的弧的数目是没有限制的，而二叉树从每个节点上最多只射出两条弧，即“ 左 ”“ 右 ”两条弧。由于节点的插入位置随其值而定，当搜寻所期望值时则不必浏览许多节点（而线性表则不同），所以二叉树是最快的信息检索方法之一。平衡二叉树在每次插入节点时，它重新组合以保证树每一部分的深度都不超过基准值。然而每次插入时的平衡处理的开销则较高。

图：无根节点的节点和弧的集合。图可以包含环和交叉路径。它通常更具有理论上的意义并不常用于实际实施。它不是STL的一部分。

• 不要做重复工作

在工作中我们首先应在编译器或其他方便之处寻求 STL中的公共组件，或从第三方供应商处购得，以减少重复工作量。从头创建这些组件目的仅仅是作为练习或没有办法的办法。

15.10 函数模板

类模板描述了类的无限集合，出现模板的大部分地方都是出现类的地方。 C++同样可以支持函数的无限集合的概念，有时它非常有用，其语法部分除用函数来替代类外和类模板没有什么两样。

假若打算创建一些函数，这些函数除了处理各自的不同类型外，函数体看上去都相同，这就有必要创建一个函数模板来描述这些函数。函数模板的典型例子是一个排序函数 [1]，然而它可适用于各种场合，下面的第一个例子作为示范。第二个例子则揭示函数模板连同循环子和包容器中的使用。

15.10.1 存储分配系统

例程malloc()、calloc()和 realloc()都可以较安全地对未开辟的存储空间进行分配。下面的

[1] 参看作者的C++Inside & out(Osborne/McGraw-Hill, 1993)。

函数模板可产生既能分配一部分新的存储空间又能为已开辟的区域重设大小（如同 realloc())的函数getmem() 。另外，它仅对新的存储进行清零处理，并且检查被其分配的存储空间。而且，提交给getmem()的参数仅是所期望的某类型元素的数目而非字节数目，所以可以降低程序出错的概率。这是它的头文件：

```
//: GETMEM.H -- Function template for memory
#ifndef GETMEM_H_
#define GETMEM_H_
#include <stdlib.h>
#include <string.h>
#include "..\allege.h"

template<class T>
void getmem(T*& oldmem, int elems) {
  typedef int cntr; // Type of element counter
  const int csz = sizeof(cntr); // And size
  const int Tsz = sizeof(T);
  if(elems == 0) {
    free(&(((cntr*)oldmem)[-1]));
    return;
  }
  T* p = oldmem;
  cntr oldcount = 0;
  if(p) { // Previously allocated memory
    ((cntr*)p)--; // Back up by one cntr
    oldcount = *(cntr*)p; // Previous # elems
  }
  T* m = (T*)realloc(p, elems * Tsz + csz);
  allegemem(m);
  *((cntr*)m) = elems; // Keep track of count
  const cntr increment = elems - oldcount;
  if(increment > 0) {
    // Starting address of data:
    long startadr = (long)&(m[oldcount]);
    startadr += csz;
    // Zero the additional new memory:
    memset((void*)startadr, 0, increment * Tsz);
  }
  // Return the address beyond the count:
  oldmem = (T*)&(((cntr*)m)[1]);
}

template<class T>
inline void freemem(T * m) { getmem(m, 0); }

#endif // GETMEM_H_
```

为了能够仅在新的存储区进行清零处理，将有一个用来指示所分配元素数目的计数器被置于每一存储区的首部。计数器的类型为 typedef cntr ，当处理较大的存储区时可将其由整型改为长整型。（当使用长整型时其他的一些问题会出现，然而不管怎样，编译器都会在警示中提示这些问题。）

之所以使用指针引用 oldmem作为参数是由于外部变量（一个指针）必须改为指向新存储区，oldmem必须指向零（以分配新存储区）或指向由 getmem()创建的存储区。该函数设想我们能正确地使用它，但如果我们打算对其调试，可在计数器附近加一个辅助标识，通过检查该标识来帮助发现getmem()中的错误调用。

如果所要求的元素数为零，则该存储被释放。有另外一个函数模板 freemem（），是这个行为的别名。

我们将注意到， getmem()的处理层次很低，存在许多底层调用和字节处理。例如，oldmem指针并不指向存储区的真实的起始位置，而恰好在起始位置计数器之后。所以在用free()释放存储区时，getmem()必须将指针向后退由cntr占用的存储空间的数目。由于oldmem的类型是T*，必须首先将其映射为cntr*，然后它被向后索引一个位置。最后，为 free()产生指定位置地址的语句表达为：

```
free(&(((cntr*) oldmem )[-1]));
```

同样地，假若这是一个已经分配的存储空间， getmem()必须后退一个cntr长度，以获取真实的存储空间的起始地址并取回先前的元素数目。在 realloc()内部需要真实的起始地址。若要开辟的存储空间是向上增加的，在 memset()中的起始地址和需要清零的元素数目可由新的元素数目减去旧的元素数目而求得。最后，产生计数器后面的地址，并将其赋给 oldmem，赋值语句为：

```
oldmem=(T*)&(((cntr*)m)[1]);
```

再者，由于oldmem是对一个指针的引用，这导致传给getmem()的外部参数的变化。

下面的程序用于测试getmem()。它分配和填入值，然后再进一步开辟更多的存储空间：

```
//: GETMEM.CPP -- Test memory function template
#include "..\14\getmem.h"
#include <iostream.h>

main() {
  int* p = 0;
  getmem(p, 10);
  for(int i = 0; i < 10; i++) {
    cout << p[i] << ' ';
    p[i] = i;
  }
  cout << '\n';
  getmem(p, 20);
  for(int j = 0; j < 20; j++) {
    cout << p[j] << ' ';
    p[j] = j;
  }
  cout << '\n';
  getmem(p, 25);
```

```
  for(int k = 0; k < 25; k++)
    cout << p[k] << ' ';
  freemem(p);
  cout << '\n';

  float* f = 0;
  getmem(f, 3);
  for(int u = 0; u < 3; u++) {
      cout << f[u] << ' ';
      f[u] = u + 3.14159;
    }
    cout << '\n';
    getmem(f, 6);
    for(int v = 0; v < 6; v++)
      cout << f[v] << ' ';
    freemem(f);
  }
```

在每次调用getmem()后，存储区中的值都被打印出来，可以看到新的存储区都被清零了。

注意由整型指针和浮点型指针而实例化getmem()的不同版本。由于上述功能涉及到非常底层的处理，我们可能认为应当使用一个非模板函数并传递 void* &作为oldmem的方式来实现。这种想法是不能实现的，因为编译器必须将我们的类型转化成 void*。为了获取引用，编译器会安排一个临时域，由于修改的是临时指针而非我们真正想要修正的指针，所以会出现错误。故使用函数模板为参数产生相应的确切类型是必需的。

15.10.2 为tstack提供函数

假设我们打算拥有一个 tstack并使一函数适用它所包含的所有对象。由于 tstack可以包含任意类型的对象，所以该函数应该可以在 tstack的任意类型和其包含的任意类型对象下工作：

```
//: APPLIST.CPP -- Apply a function to a tstack
#include "..\14\tstack.h"
#include <iostream.h>

// 0 arguments, any type of return value:
template<class T, class R>
void applist(tstack<T>& tl, R(T::*f)()) {
  tstackIterator<T> it(tl);
  while(it) {
    (it.current()->*f)();
    it++;
  }
}

// 1 argument, any type of return value:
template<class T, class R, class A>
void applist(tstack<T>& tl, R(T::*f)(A), A a) {
```

```
  tstackIterator<T> it(tl);
  while(it) {
    (it.current()->*f)(a);
    it++;
  }
}

// 2 arguments, any type of return value:
template<class T, class R, class A1, class A2>
void applist(tstack<T>& tl, R(T::*f)(A1, A2),
    A1 a1, A2 a2) {
  tstackIterator<T> it(tl);
  while(it) {
    (it.current()->*f)(a1, a2);
    it++;
  }
}

// Etc., to handle maximum probable arguments

class gromit { // The techno-dog
  int arf;
public:
  gromit(int Arf = 1) : arf(Arf + 1) {}
  void speak(int) {
    for(int i = 0; i < arf; i++)
      cout << "arf! ";
    cout << endl;
  }
  char eat(float) {
    cout << "chomp!" << endl;
    return 'z';
  }
  int sleep(char, double) {
    cout << "zzz..." << endl;
    return 0;
  }
  void sit(void) {}
};

main() {
  tstack<gromit> dogs;
  for(int i = 0; i < 5; i++)
    dogs.push(new gromit(i));
  applist(dogs, &gromit::speak, 1);
  applist(dogs, &gromit::eat, 2.0f);
```

```
  applist(dogs, &gromit::sleep, 'z', 3.0);
  applist(dogs, &gromit::sit);
}
```

applist()函数模板可获取包容器类的引用及类中成员函数的指针。为了在栈中移动applist(),这里使用了一个循环子并且把函数应用于每一个对象。假若我们已经忘了成员指针的语法，可复习第10章的后面部分。

我们可以看到有不只一个applist()版本，所以重载函数模板是可行的。虽然它们都可接受任意类型的返回值（这被忽略了，但对于匹配成员指针来说，类型信息则是所要求的），applist()的每一个版本都有一些不同的参数，由于它是一个模板，所以这些参数的类型是任意的。（在类gromit中，可以看到一批不同的函数[1]）。由于不存在“超模板”为我们生成模板，所以我们必须决定究竟需要多少参数。

虽然applist()的定义相当复杂，其中的一部分不能指望一个初学者去理解它，但是它的使用则非常清晰而简单，初学者仅仅需要知晓完成什么而非怎样完成，所以初学者可以容易地使用它。我们应尽量把程序组件分成不同的类别，仅仅关心所要完成的目标而不需关心底层的实现细节。棘手的细节问题仅仅是设计者的任务。

当然，这些功能类型会牢固地联系着tstack类，所以通常我们应该随tstack一道在头文件中找到这些函数模板，加以分析利用。

15.10.3 成员函数模板

把applist()安排为成员函数模板也是可行的，这是一个和类模板相对独立的模板定义，而且它仍然是类的成员。因此能够使用下面更巧妙的语句：

```
dogs.applist(&gromit::sit);
```

这和在类内部引出普通函数的做法（第2章）相类似[2]。

15.11 控制实例

显式实例化一个模板有时是有用的，这会告诉编译器为模板的特定版本安排代码，即使并不打算生成一个对象。为了实现它，可以重用下面的模板关键字：

```
template class bobbin<thread>;
template void sort<char>(char *[]);
```

这是SORTED.CPP例子的一个版本（见15.7），在使用它之前会显式实例化一个模板：

```
//: GENERATE.CPP -- Explicit instantiation
#include "..\14\sorted.h"
#include "..\14\integer.h"
// Explicit instantiation:
template class sorted<integer>;

main() {
  sorted<integer> is;
  urand<47> rand1;
```

[1] 参见对Nick Park的英国活泼短片《糟糕的裤子》。

[2] 检查我们的编译器版本信息，看它是否支持函数模板。

```
  for(int k = 0; k < 15; k++)
    is.add(new integer(rand1()));
  for(int l = 0; l < is.count(); l++)
    cout << *is[l] << endl;
}
```

在该例子中，显式实例并不真正地完成什么事情，它未参与程序的运作。显式实例仅在外部控制的特定情况下才是必须的。

• 模板特殊化

sorted向量仅工作于用户类型的对象上。例如，它不能对 char*型数组进行排序。为了创建一个特定版本，我们可以自己写一个实例版本，就像编译器已经通过并把我们的类型代入了模板参数一样。但是要把我们自己的代码放入特殊化的函数体中。下面的例子揭示char*型sorted向量：

```
//: SPECIAL.CPP -- Template specialization
// A special sort for char*
#include "..\14\sorted.h"
#include <iostream.h>
class sorted<char> :  public tstash<char> {
  void bubblesort();
public:
  int add(char* element) {
    tstash<char>::add(element);
    bubblesort();
    return 0; // Sort moves the element
  }
};

void sorted<char>::bubblesort() {
  for(int i = count(); i > 0; i--)
    for(int j = 1; j < i; j++)
      if(strcmp(storage[j], storage[j-1]) < 0) {
        // Swap the two elements:
        char* t = storage[j-1];
        storage[j-1] = storage[j];
        storage[j] = t;
      }
}

char* words[] = {
  "is", "running", "big", "dog", "a",
};
const wsz = sizeof words/sizeof *words;

main() {
  sorted<char> sc;
  for(int k = 0; k < wsz; k++)
```

```
    sc.add(words[k]);
  for(int l = 0; l < sc.count(); l++)
    cout << sc[l] << endl;
}
```

在bubblesort()中，我们可以看到使用的是 strcmp()而不是>。

15.12 小结

包容器类是OOP的一个基本部分，它是简化和隐藏实施细节，提高开发效率的另一种方法。另外，它通过对C中的旧式数组和粗糙的数据结构技术的更新替代从而大大地提高了灵活性和安全性。

由于包容器是客户程序员所需要的，所以包容器的实质是便于使用，这样，模板就被引入进来。对源代码进行重用（相反的是继承和组合实施对对象代码的重用）的模板语句可对初学者来说变得十分平常。实际上，使用模板实施代码重用比继承和组合容易得多。

虽然在本书中我们已经学习了包容器和循环子，但在实际中，学习编译器所带的包容器和循环子是更迅速的方法，要不然就从第三方供应商处购买一个库 [1]。标准C++库是很完备的，但在包容器和循环子方面并不充分。

本章简单地提及了包容器类设计方面的内容，我们可以加以总结以有更多的体会。一个复杂的包容器类库可能涉及所有的附加内容，包括持久性（在第 16章中介绍）和垃圾回收（在第12章中介绍），也包含处理所有权问题的附加方法。

15.13 练习

1. 修改第14章练习一的结果，以便使用tstack 和tstackIterator替代 shape指针数组。增加针对类层次的析构函数以便在tstack超出范围时观察shape对象被析构。

2. 修改第14章例子SSHAPE2.CPP以使用tstack替代数组。

3. 修改RECYCLE.CPP以使用tstash替代tstack。

4. 改变SETTEST.CPP以使用 sortedSet替代set.。

5. 为 tstash 类复制APPLIST.CPP的功能。

6. 将TSTACK.H拷贝到新的头文件中，并增加 APPLIST.CPP中的函数模板作为tstack的成员函数模板。本练习要求我们的编译器支持成员函数模板。

7. （高级）修改tstack类以促进增加所有权的区分粒度。为每一个链接增加标志以表明它是否拥有其指向的对象，并在 add()函数和析构函数中支持这一标志信息。增加用于读取和改变每一链接所有权的成员函数，并在新的上下文环境中确定 add()标志的含义。

8. （高级）修改tstack类，使每一个入口包含引用计数信息（不是它们所包容的对象)并且增加用于支持引用计数行为的成员函数。

9. （高级）改变SORTED.CPP中urand的底层实现以提高其空间效率（SORTED.CPP后面段落所描述的) 而非时间效率。

10. （高级）将GETMEM.H中的typedef cntr从整型改成长整型，并且修改代码以消除失去精度的警示信息，这是一个指针算术问题。

11. （高级）设计一个测试程序，用于比较创建在堆上和创建在栈上的 SString的执行速度。

---

[1] C++标准库包含一个非常地道的但并非详尽无遗的包容器和循环子集。