第 19 章　模　　板

C++程序员可使用的一种功能强大的工具是参数化类型（模板）。模板的用途非常大，因此在C++语言中包含一个这样的库：它包含大量使用模板的函数。

本章介绍以下内容：

- 什么是模板？如何使用模板？
- 如何创建类模板？
- 如何创建函数模板？
- 什么是标准模板库（STL）？如何使用其中的一些模板？

19.1　什么是模板

在第2周课程末尾，读者知道了如何创建PartsList类并使用它来创建PartsCatalog类。如果要使用PartsList类来创建一个猫链表，将遇到一个问题：PartsList只能处理零件。

为解决这种问题，可创建一个List基类，然后将PartsList类的大部分代码剪贴到新的CatsList类声明中。以后当你需要创建一个Car对象的链表时，必须创建一个新类并剪贴代码。

这无疑不是一种令人满意的解决方案。随着时间的流逝，还可能需要对List及其派生类进行扩展。确保在所有相关类进行所需的修改将是一场恶梦。

另一种解决方案是，从Part类派生出Cat类，这样Part对象也可以存储Cat对象。显然，从确保类层次结构在概念上清晰的角度说，这样做存在问题，因为猫不是零件。

也可以创建一个List类，它可以包含对象，而这些对象都是从同一个基类派生而来的。然而，这破坏了强类型特征，使得难以让编译器来检查程序的正确性。你真正需要的是一种创建一系列相关类的方法，这些类之间惟一差别在于，它们处理的东西的类型不同；同时，需要修改这些类时只需在一个地方进行修改，从而减少维护工作量。

通过创建并使用模板，可解决这些问题。虽然模板不是最初的C++语言的组成部分，但现在已经是C++标准的组成部分，且是C++语言不可或缺的组成部分。像所有的C++特性一样，模板是类型安全的且非常灵活。然而，对C++编程新手来说，模板可能很可怕，但理解它们后，将发现它们是一种功能强大的语言特性。

模板让你能够创建一个这样的类：可以更改其处理的东西的类型。例如，可以使用模板来告诉编译器如何创建一个由任何对象组成的链表，而不是创建一组分别由特定对象组成的链表：PartsList是零件链表，CatsList是猫链表。它们之间惟一的区别在于，链表中包含的对象不同。使用模板时，链表中包含的对象类型将成为类定义的一个参数。根据模板可以创建一系列的类，其中每个类处理不同的对象。

数组类是几乎所有C++库都有的一个组件。分别为int、double和Animal创建一个数组类将非常繁琐且效率低下。使用模板可以声明一个参数化数组类，然后指定该类的每个实例将存储的对象类型。

虽然标准模板库提供了一组标准化的容器类，其中包括数组、链表等，但学习模板的最佳方式是创建自己的模板。接下来的几节将讨论编写自己的模板类需要完成的工作，让读者能够深入了解模板的工作原理。然而，在商业程序中，几乎肯定要使用标准模板库类而不是创建自己的模板类。另一方面，你可能需要为自

己的应用程序创建模板，以充分利用这种强大的功能。

实例化是根据模板创建一个具体的类型。根据模板创建的类被称为模板的实例。

注意：模板实例不同于使用模板创建的类的实例。“实例化”通常指的是根据类创建一个实例（对象）。使用或阅读“实例化”时，一定要注意上下文。

参数化模板提供了这样一种功能：创建一个通用类，然后通过将类型作为参数传递给这个类来创建特定的实例。

实例化模板之前，必须定义它。

19.2 创建模板定义

模板的基本声明由关键字 template 和表示类型的参数组成，格式如下：

```
template <class T>
```

其中，template 和 class 都是关键字；T 是一个占位符，类似于变量名，因此可以使用任何名称，但通常使用 T 或 Type。T 的值为一种数据类型。

由于在这种上下文中，使用关键字 class 可能令人迷惑，因此可以使用关键字 typename 类代替：

```
template <typename T>
```

在本章中将使用关键字 class，因为在已有的程序中，使用关键字 class 的情况更常见。然而，在参数为基本类型而不是类时，关键字 typename 更明确地指出了你要定义什么。

回到创建自己的数组类的问题，可以使用模板语句给 Array 类声明一个参数化类型，从而创建一个数组模板，如下所示：

```
template <class T>   // declare the template and the parameter
class Array          // the class being parameterized
{
      public:
            Array();
    // full class declaration here
};
```

上述代码是名为 Array 的模板类的基本声明。在每个模板类的声明和定义开头需要使用关键字 template，随后是模板参数。和函数一样，参数是随每个实例而异的。在前一个例子中创建的数组模板中，你希望存储在数组中的对象的类型是可变的，一个实例可能是 int 数组，而另一个实例可能是 Animals 数组。

在这个例子中使用了关键字 class，它后面是标识符 T。正如前面指出的，关键字 class 表示该参数是一个类型。标识符 T 在模板定义的其余部分都用来表示参数化类型。由于这个类现在是一个模板，因此一个实例可能用 int 代替 T，而另一个实例可能用 Cat 代替。如果编写正确，模板应该能够接受将 T 的值设置为任何合法的数据类型（或类）。

在声明模板实例时指定参数的具体类型，格式如下：

```
className<type> instance;
```

其中 classname 为模板类的名称，instance 是要创建的实例的名称，type 是要用于该实例的数据类型。

例如，要声明参数化 Array 类的一个 int 和 Animals 实例，可以这样做：

```
Array<int> anIntArray;
Array<Animal> anAnimalArray;
```

anIntArray 的类型为 int 数组，anAnimalArray 的类型为 Animals 数组。现在可以在任何可使用类型的地方使用类型 Array<int>：将其用作函数的返回类型、函数的参数类型等。程序清单 19.1 列出了的 Array 模板

的完整声明。注意，这不是一个完整的程序，而只是演示如何定义模板。

程序清单 19.1 类模板 Array

```
0:  //Listing 19.1 A template of an array class
1:  #include <iostream>
2:  using namespace std;
3:  const int DefaultSize = 10;
4:
5:  template <class T>   // declare the template and the parameter
6:  class Array          // the class being parameterized
7:  {
8:    public:
9:      // constructors
10:     Array(int itsSize = DefaultSize);
11:     Array(const Array &rhs);
12:     ~Array() { delete [] pType; }
13:
14:     // operators
15:     Array& operator=(const Array&);
16:     T& operator[](int offSet) { return pType[offSet]; }
17:
18:     // accessors
19:     int getSize() { return itsSize; }
20:
21:   private:
22:     T *pType;
23:     int  itsSize;
24: };
```

该程序清单没有输出，因为它不是一个完整的程序，而只是一个简化模板的定义。

分析：

模板定义从第 5 行开始，它以关键字 template 开头，紧接着是参数。在这个例子中，使用关键字 class 标识参数为类型，标识符 T 用于表示参数化类型。正如前面指出的，也可以用关键字 typename 代替 class：

```
5:  template <typename T>  // declare the template and the parameter
```

应使用对自己来说更清晰的关键字，虽然推荐的做法是，在类型为类时使用关键字 class，在类型不是类时使用关键字 typename。

从第 6～24 行的模板末尾，声明与其他类声明相同。惟一区别在于，在通常使用对象类型的地方使用了标识符 T。例如，operator[ ]本应返回数组中对象的引用，但实际上被声明为返回 T 的引用。

定义 int 数组实例时，将用 int 替换 T，因此该数组的 operator=将返回 int 引用。这相当于下述代码：

```
int& operator[](int offSet) { return pType[offSet]; }
```

声明 Animal 数组实例时，该 Animal 数组的 operator=将返回一个 Animal 引用：

```
Animal& operator[](int offSet) { return pType[offSet]; }
```

从某种程度上说，这与宏的工作原理极其相似，事实上，创建模板旨在为了 C++对宏的需求。

19.2.1 使用名称

在类声明中使用 Array 时无需对其进行限定。在程序的其他地方，将使用 Array<T>来引用这个类。例如，如果没有在类声明中实现构造函数，则实现它时必须这样做：

```
template <class T>
Array<T>::Array(int size):
itsSize = size
{
      pType = new T[size];
      for (int i = 0; i < size; i++)
      pType[i] = 0;
}
```

由于这是模板的一部分，为指出函数的归属（class T），上述代码段中第 1 行的声明是必不可少的。在第 2 行，模板的名称为 Array<T>，函数的名称为 Array(int size)。另外，该函数接受一个 int 参数。

该函数的其他代码与非模板函数相同，只是在本需要数组类型的地方使用了参数 T（参见声明新数组的代码行：new T[size]）。

提示：一种常见也是推荐的方法是，先创建一个可行的类及其函数，然后将其转换为模板。这可以简化开发工作，让你将重点放在编程目标上，然后再使用模板推广解决方案。

另外，必须在声明模板的文件中实现模板函数。不同于其他类（可以将类及其成员函数的声明放在头文件中，而将必要的函数实现放在.cpp 文件中），模板要求将声明和实现都放在头文件或.cpp 文件中。如果与项目的其他部分共享模板，常见的做法是，要么在模板类声明中以内联方式实现成员函数，要么在位于头文件中的模板类声明后面实现它们。

19.2.2 实现模板

定义模板后，可能想使用它。要完整地实现模板类 Array，需要实现复制构造函数、operator=等。程序清单 19.2 提供了模板类的 Array 定义和实现以及一个使用该模板的 main( )函数。

注意：一些较老的编译器不支持模板。然而，模板是 ANSI C++标准的组成部分，所有主要的编译器厂商都在其最新的编译器版本中支持模板。如果你使用的是很旧的编译器，将无法编译和运行本章的程序。然而，通读本章并在对编译器升级后回过头来完成这些程序仍不失为一个不错的主意。

程序清单 19.2 模板类 Array 的实现

```
 0: //Listing 19.2 The Implementation of the Template Array
 1: #include <iostream>
 2:
 3: const int DefaultSize = 10;
 4:
 5: // declare a simple Animal class so that we can
 6: // create an array of animals
 7:
 8: class Animal
 9: {
10:   public:
11:     Animal(int);
12:     Animal();
13:     ~Animal() {}
14:     int GetWeight() const { return itsWeight; }
15:     void Display() const { std::cout << itsWeight; }
16:   private:
17:     int itsWeight;
18: };
```

```
19:
20: Animal::Animal(int weight):
21:    itsWeight(weight)
22: {}
23:
24: Animal::Animal():
25:    itsWeight(0)
26: {}
27:
28:
29: template <class T>   // declare the template and the parameter
30: class Array          // the class being parameterized
31: {
32:  public:
33:    // constructors
34:    Array(int itsSize = DefaultSize);
35:    Array(const Array &rhs);
36:    ~Array() { delete [] pType; }
37:
38:    // operators
39:    Array& operator=(const Array&);
40:    T& operator[](int offSet) { return pType[offSet]; }
41:    const T& operator[](int offSet) const
42:      { return pType[offSet]; }
43:    // accessors
44:    int GetSize() const { return itsSize; }
45:
46:  private:
47:    T *pType;
48:    int itsSize;
49: };
50:
51: // implementations follow...
52:
53: // implement the Constructor
54: template <class T>
55: Array<T>::Array(int size):
56: itsSize(size)
57: {
58:    pType = new T[size];
59:    // the constructors of the type you are creating
60:    // should set a default value
61: }
62:
63: // copy constructor
64: template <class T>
65: Array<T>::Array(const Array &rhs)
66: {
67:    itsSize = rhs.GetSize();
68:    pType = new T[itsSize];
69:    for (int i = 0; i<itsSize; i++)
70:       pType[i] = rhs[i];
```

```
71: }
72:
73: // operator=
74: template <class T>
75: Array<T>& Array<T>::operator=(const Array &rhs)
76: {
77:    if (this == &rhs)
78:       return *this;
79:    delete [] pType;
80:    itsSize = rhs.GetSize();
81:    pType = new T[itsSize];
82:    for (int i = 0; i<itsSize; i++)
83:       pType[i] = rhs[i];
84:    return *this;
85: }
86:
87: // driver program
88: int main()
89: {
90:    Array<int> theArray;      // an array of integers
91:    Array<Animal> theZoo;     // an array of Animals
92:    Animal *pAnimal;
93:
94:    // fill the arrays
95:    for (int i = 0; i < theArray.GetSize(); i++)
96:    {
97:       theArray[i] = i*2;
98:       pAnimal = new Animal(i*3);
99:       theZoo[i] = *pAnimal;
100:      delete pAnimal;
101:   }
102:   // print the contents of the arrays
103:   for (int j = 0; j < theArray.GetSize(); j++)
104:   {
105:      std::cout << "theArray[" << j << "]:\t";
106:      std::cout << theArray[j] << "\t\t";
107:      std::cout << "theZoo[" << j << "]:\t";
108:      theZoo[j].Display();
109:      std::cout << std::endl;
110:   }
111:
112:   return 0;
113: }
```

输出：

```
theArray[0]:    0          theZoo[0]:    0
theArray[1]:    2          theZoo[1]:    3
theArray[2]:    4          theZoo[2]:    6
theArray[3]:    6          theZoo[3]:    9
theArray[4]:    8          theZoo[4]:    12
theArray[5]:    10         theZoo[5]:    15
theArray[6]:    12         theZoo[6]:    18
```

```
theArray[7]:    14          theZoo[7]:    21
theArray[8]:    16          theZoo[8]:    24
theArray[9]:    18          theZoo[9]:    27
```

分析：

这个程序非常简单，但演示了如何创建和使用模板。它首先定义了模板 Array，然后使用它来实例化类型为 int 和 Animal 的 Array 对象。int 数组的元素被设置为其索引的两倍。由 Animal 对象组成的数组名为 theZoo，其中每个 Animal 对象的体重被设置为相应索引的 3 倍。

接下来深入讨论这些代码。第 8～26 行创建了一个简化的 Animal 类，以便可以将用户定义类型的对象存储到数组中。

第 29 行的语句指出，接下来是一个模板，模板的参数是类型，名为 T。正如前面指出的，这里的 class 也可用 typename 代替。

Array 类有两个构造函数（第 34 和 35 行）。第一个接受一个长度参数，该参数的默认值为 int 常量 DefaultSize。

声明了赋值运算符和下标运算符，其中后者包含 const 和非 const 版本。提供的惟一一个存取器函数是 GetSize( )（第 44 行），它返回数组的长度。

可以想像一个更完整的接口，对于任何严格数组程序而言，这里提供的功能是不够的。至少还需要删除元素、扩大数组、缩小数组等运算符。如果使用标准模板库中的 Array 类，将发现它提供了这些功能。本章后面将更详细地介绍。

Array 模板类的私有数据包括数组长度和指向对象数组的指针。

从第 53 行开始，是该模板类的一些成员函数的实现。由于这些实现位于模板类声明之外，因此必须再次指出它们是模板的一部分。为此，需要使用位于类声明前的语句，如第 54 行所示。另外，通过在类名后面包含参数，指出了 Array 是一个模板类。在第 54 行，已经将类型参数声明为 T，因此使用 Array<T>来指出函数为 Array 类的成员函数，如第 55 行所示。

在成员函数中，可以在通常应使用数组类型的地方使用参数 T。例如，在第 58 行，类的指针成员 pType 被设置为新创建的数组；该数组的元素的类型将为 T；使用该模板类实例化对象时指定的类型。创建每个指定类型的元素时，其构造函数将对其进行初始化。

第 64～71 行实现复制构造函数以及第 74～85 行重载运算符=时重复了上述过程。

对模板类 Array 的实际使用位于第 90 和 91 行。第 90 行使用它来实例化一个名为 theArray 的对象，它存储 int 数组；第 91 行实例化一个名为 theZoo 的、类型为 Animal 的数组。

该程序清单中的其他代码完成前面描述的功能，它们很容易理解。

19.3　将实例化的模板对象传递给函数

要将 Array 对象传递给常规函数，必须传递 Array 的一个实例，而不是模板。因此，要创建一个接受一个 Array 实例作为参数的函数，需要这样声明参数类型：

```
void SomeFunction(Array<theType>&);
```

其中 SomeFunction 是要将 Array 对象传递给它的函数的名称；theType 是要传递的对象的类型。因此，如果 SomeFunction( )接受一个 int 数组作为参数，可以这样声明它：

```
void SomeFunction(Array<int>&);     // ok
```

但不能这样声明：

```
void SomeFunction(Array<T>&);       //error!
```

因为无法知道 T&是什么。也不能这样声明：

```
void SomeFunction(Array &);          //error!
```

因为没有 Array 类，只有模板和实例。

要创建具有模板实现的某些优点的非成员函数，可以声明模板函数，方法与声明模板类并定义模板成员函数类似——首先指出函数是一个模板，然后在本应使用类型和类名的地方使用模板参数：

```
template <class T>
void MyTemplateFunction(Array<T>&); //ok
```

在这个例子中，第 1 行代码导致函数 MyTemplateFunction( )被声明为一个模板函数。注意，和其他函数一样，模板函数可以为任何名称。

除参数化形式外，模板函数还可接受模板实例作为参数，如下例所示：

```
template <class T>
void MyOtherFunction(Array<T>&, Array<int>&); //ok
```

注意，该函数接受两个数组作为参数：一个参数化数组和一个 int 数组。前者可以是由任何对象组成的数组，但后者只能是 int 数组。本章后面将介绍如何使用模板函数。

19.4 模板和友元

第 16 章介绍过友元，模板类可声明 3 种友元：

- 非模板友元类或函数。
- 通用模板友元类或函数。
- 特定类型的模板友元类或函数。

下面首先介绍前两种情况。

19.4.1 非模板友元类和函数

可将任何类或函数声明为模板类的友元。模板的每个实例都将正确处理友元关系，就像友元关系是在该实例中声明的一样。

程序清单 19.3 在 Array 类的模板定义中添加了友元函数 Intrude( )，然后在 main( )函数中调用了 Intrude( )。

由于 Intrude( )是友元函数，因此可以访问 Array 的私有数据。由于 Intrude( )不是模板函数，因此只能将 int 类型的 Array 传递给它。

程序清单 19.3 非模板友元函数

```
0:  // Listing 19.3 - Type specific friend functions in templates
1:
2:  #include <iostream>
3:  using namespace std;
4:
5:  const int DefaultSize = 10;
6:
7:  // declare a simple Animal class so that we can
8:  // create an array of animals
9:
10: class Animal
11: {
12:  public:
13:    Animal(int);
14:    ~Animal();
```

```
15:    ~Animal() {}
16:    int GetWeight() const { return itsWeight; }
17:    void Display() const { cout << itsWeight; }
18:  private:
19:    int itsWeight;
20: };
21:
22: Animal::Animal(int weight):
23:   itsWeight(weight)
24: {}
25:
26: Animal::Animal():
27:   itsWeight(0)
28: {}
29:
30: template <class T>  // declare the template and the parameter
31: class Array          // the class being parameterized
32: {
33:  public:
34:    // constructors
35:    Array(int itsSize = DefaultSize);
36:    Array(const Array &rhs);
37:    ~Array() { delete [] pType; }
38:
39:    // operators
40:    Array& operator=(const Array&);
41:    T& operator[](int offSet) { return pType[offSet]; }
42:    const T& operator[](int offSet) const
43:    { return pType[offSet]; }
44:    // accessors
45:    int GetSize() const { return itsSize; }
46:
47:    // friend function
48:    friend void Intrude(Array<int>);
49:
50:  private:
51:    T *pType;
52:    int itsSize;
53: };
54:
55: // friend function. Not a template, can only be used
56: // with int arrays! Intrudes into private data.
57: void Intrude(Array<int> theArray)
58: {
59:   cout << endl << "*** Intrude ***" << endl;
60:   for (int i = 0; i < theArray.itsSize; i++)
61:      cout << "i: " << theArray.pType[i] << endl;
62:   cout << endl;
63: }
64:
65: // implementations follow...
66:
```

```
67: // implement the Constructor
68: template <class T>
69: Array<T>::Array(int size):
70:   itsSize(size)
71: {
72:    pType = new T[size];
73:    // the constructors of the type you are creating
74:    // should set a default value
75: }
76:
77: // copy constructor
78: template <class T>
79: Array<T>::Array(const Array &rhs)
80: {
81:   itsSize = rhs.GetSize();
82:    pType = new T[itsSize];
83:   for (int i = 0; i < itsSize; i++)
84:      pType[i] = rhs[i];
85: }
86:
87: // operator=
88: template <class T>
89: Array<T>& Array<T>::operator=(const Array &rhs)
90: {
91:    if (this == &rhs)
92:       return *this;
93:    delete [] pType;
94:    itsSize = rhs.GetSize();
95:    pType = new T[itsSize];
96:    for (int i = 0; i < itsSize; i++)
97:       pType[i] = rhs[i];
98:    return *this;
99: }
100:
101: // driver program
102: int main()
103: {
104:    Array<int> theArray;      // an array of integers
105:    Array<Animal> theZoo;     // an array of Animals
106:    Animal *pAnimal;
107:
108:   // fill the arrays
109:    for (int i = 0; i < theArray.GetSize(); i++)
110:    {
111:       theArray[i] = i*2;
112:       pAnimal = new Animal(i*3);
113:       theZoo[i]   *pAnimal;
114:    }
115:
116:    int j;
117:    for (j = 0; j < theArray.GetSize(); j++)
118:    {
```

```
119:      cout << "theZoo[" << j << "];\t";
120:      theZoo[j].Display();
121:      cout << endl;
122:    }
123:    cout << "Now use the friend function to ";
124:    cout << "find the members of Array<int>";
125:    Intrude(theArray);
126:
127:    cout << end_ <<"Done." << endl;
128:    return 0;
129:  }
```

输出：

```
theZoo[0]:    0
theZoo[1]:    3
theZoo[2]:    6
theZoo[3]:    9
theZoo[4]:    12
theZoo[5]:    15
theZoo[6]:    18
theZoo[7]:    21
theZoo[8]:    24
theZoo[9]:    27
Now use the friend function to find the members of Array<int>
*** Intrude ***
i: 0
i: 2
i: 4
i: 6
i: 8
i: 10
i: 12
i: 14
i: 16
i: 18

Done.
```

分析：

对模板 Array 的声明进行了扩展，包括友元函数 Intrude( )。新增的第 48 行代码让每个 int Array 实例将 Intrude( )视为友元函数；因此，Intrude( )可以访问 Array 实例的私有成员数据和函数。

函数 Intrude( )的实现位于第 57～63 行。在第 60 行，Intrude( )直接访问 itsSize；在 61 行，它直接访问 pType。这种使用这些成员的方式并非必要的，因为 Array 类提供了这些数据的公有存取器函数，但演示了如何在模板中声明友元函数。

19.4.2 通用模板友元类和函数

给 Array 类添加一个显示运算符将很有帮助，这样可以将值发送给输出流，进而基于类型被正确处理。一种方法是，为每种可能的 Array 类型声明一个显示运算符，但这有悖于将 Array 作为模板的初衷。

所需要的是一个插入运算符，它可用于任何类型的 Array：

```
ostream& operator<< (ostream&, Array<T>&);
```

为使其可行，必须将 operator<<声明为模板函数：

```
template <class T>
ostream& operator<< (ostream&, Array<T>&);
```

将 operator<<声明为模板函数后，只需提供其实现即可。程序清单 19.4 对模板 Array 进行了扩展，以包含上述声明并提供了 operator<<的实现。

程序清单 19.4　使用运算符<<

```
0:  //Listing 19.4 Using Operator ostream
1:  #include <iostream>
2:  using namespace std;
3:
4:  const int DefaultSize = 10;
5:
6:  class Animal
7:  {
8:   public:
9:      Animal(int);
10:    Animal();
11:    ~Animal() {}
12:    int GetWeight() const { return itsWeight; }
13:    void Display() const { cout << itsWeight; }
14:  private:
15:     int itsWeight;
16: };
17:
18: Animal::Animal(int weight):
19:    itsWeight(weight)
20: {}
21:
22: Animal::Animal():
23:    itsWeight(0)
24: {}
25:
26: template <class T>   // declare the template and the parameter
27: class Array          // the class being parameterized
28: {
29:  public:
30:    // constructors
31:    Array(int itsSize = DefaultSize);
32:    Array(const Array &rhs);
33:    ~Array() { delete [] pType; }
34:
35:    // operators
36:    Array& operator=(const Array&);
37:    T& operator[](int offSet) { return pType[offSet]; }
38:    const T& operator[](int offSet) const
39:       { return pType[offSet]; }
40:    // accessors
41:    int GetSize() const { return itsSize; }
42:    // template <class T>
43:    friend ostream& operator<< (ostream&, Array<T>&);
```

```
44:
45:  private:
46:    T *pType;
47:    int  itsSize;
48: };
49:
50: template <class T>
51: ostream& operator<< (ostream& output, Array<T>& theArray)
52: {
53:   for (int i = 0; i < theArray.itsSize; i++)
54:     output << "[" << i << "] " << theArray[i] << endl;
55:   return output;
56: }
57:
58: // implementations follow...
59:
60: // implement the Constructor
61: template <class T>
62: Array<T>::Array(int size):
63:   itsSize(size)
64: {
65:   pType = new T[size];
66:   for (int i = 0; i < size; i++)
67:     pType[i] = 0;
68: }
69:
70: // copy constructor
71: template <class T>
72: Array<T>::Array(const Array &rhs)
73: {
74:   itsSize = rhs.GetSize();
75:   pType = new T[itsSize];
76:   for (int i = 0; i < itsSize; i++)
77:     pType[i] = rhs[i];
78: }
79:
80: // operator=
81: template <class T>
82: Array<T>& Array<T>::operator=(const Array &rhs)
83: {
84:   if (this == &rhs)
85:     return *this;
86:   delete [] pType;
87:   itsSize = rhs.GetSize();
88:   pType = new T[itsSize];
89:   for (int i = 0; i < itsSize; i++)
90:     pType[i] = rhs[i];
91:   return *this;
92: }
93:
94: int main()
95: {
```

```
 96:    bool Stop = false;      // flag for looping
 97:    int offset, value;
 98:   Array<int> theArray;
 99:
100:    while (Stop == false)
101:   {
102:      cout << "Enter an offset (0-9) ";
103:      cout << "and a value. (-1 to stop): ";
104:      cin >> offset >> value;
105:
106:     if (offset < 0)
107:        break;
108:
109:     if (offset > 9)
110:     {
111:        cout << "***Please use values between 0 and 9.***\n";
112:        continue;
113:     }
114:
115:      theArray[offset] = value;
116:   }
117:
118:    cout << "\nHere's the entire array:\n";
119:    cout << theArray << endl;
120:    return 0;
121: }
```

注意：如果读者使用的是 Microsoft 编译器，请取消对第 42 行的注释。根据 C++标准，这行代码是不必要的，然而使用 Microsoft C++编译器进行编译时，这行代码是必不可少的。

输出：

```
Enter an offset (0-9) and a value. (-1 to stop):1 10
Enter an offset (0-9) and a value. (-1 to stop):2 20
Enter an offset (0-9) and a value. (-1 to stop):3 30
Enter an offset (0-9) and a value. (-1 to stop):4 40
Enter an offset (0-9) and a value. (-1 to stop):5 50
Enter an offset (0-9) and a value. (-1 to stop):6 60
Enter an offset (0-9) and a value. (-1 to stop):7 70
Enter an offset (0-9) and a value. (-1 to stop):8 80
Enter an offset (0-9) and a value. (-1 to stop):9 90
Enter an offset (0-9) and a value. (-1 to stop):10 10
***Please use values between 0 and 9.***
Enter an offset (0-9) and a value. (-1 to stop):-1 -1

Here's the entire array:
[0] 0
[1] 10
[2] 20
[3] 30
[4] 40
[5] 50
[6] 60
```

```
[7] 70
[8] 80
[9] 90
```

分析：

在第 43 行，函数模板 operator<<( )被声明为模板类 Array 的友元函数。由于 operator<<( )是作为一个模板函数实现的，因此 Array 的每个实例都自动有友元函数 operator<<( )。

该运算符的实现从第 50 行开始，它使用一个简单的循环（第 53 和 54 行）依次输出数组的每个成员。仅当为数组存储在每种对象都定义了 operator<<( )时，这种做法才可行。

需要指出的是，该程序清单也必须重载 operator[ ]。从第 37 行可知，这也是使用模板类型来实现的。

19.5 使用模板对象

可以像使用其他类型那样使用模板：可以按值或按引用将它们作为参数传递给函数，也可以按值或按引用从函数返回它们。程序清单 19.5 演示了如何传递模板对象。

程序清单 19.5 将模板对象传递给函数和从函数返回模板对象

```
0:  //Listing 19.5 Passing Template Objects to and from Functions
1:  #include <iostream>
2:  using namespace std;
3:
4:  const int DefaultSize = 10;
5:
6:  // A trivial class for adding to arrays
7:  class Animal
8:  {
9:    public:
10:     // constructors
11:     Animal(int);
12:     Animal();
13:     ~Animal();
14:
15:    // accessors
16:     int GetWeight() const { return itsWeight; }
17:     void SetWeight(int theWeight) { itsWeight = theWeight; }
18:
19:     // friend operators
20:     friend ostream& operator<< (ostream&, const Animal&);
21:
22:   private:
23:     int itsWeight;
24: };
25:
26: // extraction operator for printing animals
27: ostream& operator<<
28: (ostream& theStream, const Animal& theAnimal)
29: {
30:    theStream << theAnimal.GetWeight();
31:    return theStream;
```

```
32: }
33:
34: Animal::Animal(int weight):
35: itsWeight(weight)
36: {
37:    // cout << "Animal(int)" << endl;
38: }
39:
40: Animal::Animal():
41: itsWeight(0)
42: {
43:   // cout << "Animal()" << endl;
44: }
45:
46: Animal::~Animal()
47: {
48:    // cout << "Destroyed an animal..." << endl;
49: }
50:
51: template <class T>   // declare the template and the parameter
52: class Array          // the class being parameterized
53: {
54:   public:
55:     Array(int itsSize = DefaultSize);
56:     Array(const Array &rhs);
57:    ~Array() { delete [] pType; }
58:
59:     Array& operator=(const Array&);
60:     T& operator[](int offSet) { return pType[offSet]; }
61:     const T& operator[](int offSet) const
62:       { return pType[offSet]; }
63:     int GetSize() const { return itsSize; }
64:     // friend function:
65:     // template <class T>
66:     friend ostream& operator<< (ostream&, const Array<T>&);
67:
68:  private:
69:     T *pType;
70:     int itsSize;
71: };
72:
73: template <class T>
74: ostream& operator<< (ostream& output, const Array<T>& theArray)
75: {
76:    for (int i = 0; i < theArray.itsSize; i++)
77:      output << "[" << i << "] " << theArray[i] << endl;
78:    return output;
79: }
80:
81: // implementations follow...
82:
83: // implement the Constructor
```

```
84: template <class T>
85: Array<T>::Array(int size):
86: itsSize(size)
87: {
88:    pType = new T[size];
89:    for (int i = 0; i < size; i++)
90:       pType[i] = 0;
91: }
92:
93: // copy constructor
94: template <class T>
95: Array<T>::Array(const Array &rhs)
96: {
97:    itsSize = rhs.GetSize();
98:    pType = new T[itsSize];
99:   for (int i = 0; i < itsSize; i++)
100:      pType[i] = rhs[i];
101: }
102:
103: void IntFillFunction(Array<int>& theArray);
104: void AnimalFillFunction(Array<Animal>& theArray);
105:
106: int main()
107: {
108:    Array<int> intArray;
109:    Array<Animal> animalArray;
110:    IntFillFunction(intArray);
111:    AnimalFillFunction(animalArray);
112:    cout << "intArray...\n" << intArray;
113:    cout << "\nanimalArray...\n" << animalArray << endl;
114:    return 0;
115: }
116:
117: void IntFillFunction(Array<int>& theArray)
118: {
119:   bool Stop = false;
120:   int offset, value;
121:    while (Stop == false)
122:   {
123:      cout << "Enter an offset (0-9) ";
124:      cout << "and a value. (-1 to stop): " ;
125:      cin >> offset >> value;
126:      if (offset < 0)
127:         break;
128:      if (offset > 9)
129:      {
130:         cout << "***Please use values between 0 and 9.***\n";
131:         continue;
132:      }
133:       theArray[offset] = value;
134:   }
135: }
```

```
136:
137:
138: void AnimalFillFunction(Array<Animal>& theArray)
139: {
140:    Animal * pAnimal;
141:    for (int i = 0; i < theArray.GetSize(); i++)
142:    {
143:       pAnimal = new Animal;
144:       pAnimal->SetWeight(i*100);
145:       theArray[i] = *pAnimal;
146:       delete pAnimal;  // a copy was put in the array
147:    }
148: }
```

注意：如果读者使用的是 Microsoft 编译器，请取消对第 65 行的注释。根据 C++标准，这行代码是不必要的，然而使用 Microsoft C++编译器进行编译时，这行代码是必不可少的。

输出：

```
Enter an offset (0-9) and a value. (-1 to stop):1 10
Enter an offset (0-9) and a value. (-1 to stop):2 20
Enter an offset (0-9) and a value. (-1 to stop):3 30
Enter an offset (0-9) and a value. (-1 to stop):4 40
Enter an offset (0-9) and a value. (-1 to stop):5 50
Enter an offset (0-9) and a value. (-1 to stop):6 60
Enter an offset (0-9) and a value. (-1 to stop):7 70
Enter an offset (0-9) and a value. (-1 to stop):8 80
Enter an offset (0-9) and a value. (-1 to stop):9 90
Enter an offset (0-9) and a value. (-1 to stop): 10 10
***Please use values between 0 and 9.***
Enter an offset (0-9) and a value. (-1 to stop): -1 -1

intArray:...
[0] 0
[1] 10
[2] 20
[3] 30
[4] 40
[5] 50
[6] 60
[7] 70
[8] 80
[9] 90

animalArray:...
[0] 0
[1] 100
[2] 200
[3] 300
[4] 400
[5] 500
[6] 600
[7] 700
```

```
[8] 800
[9] 900
```

分析：

为节省篇幅，省略了 Array 类的大部分实现。Animal 类是在第 7～24 行声明的。虽然这是一个精简的类，但它提供了插入运算符（<<）以支持打印 Animals。从第 27～32 行的插入运算符定义可知，只是打印 Animal 的重量而已。

注意 Animal 有一个默认构造函数。这是必不可少的，因为将对象添加到数组中时，将使用默认构造函数来创建对象。这带来了一些麻烦，稍后读者将看到。

第 103 行声明了函数 IntFillFunction()。该原型表明，这个函数接受一个 int 型 Array 作为参数。注意，它不是模板函数。IntFillFunction()只能接受一种类型的 Array（int 数组）作为参数。与此相似，第 104 行将 AnimalFillFunction()声明为接受一个 Animal 类型的 Array 作为参数。

这两个函数的实现相同，因为填充 int 数组的方式与填充 Animals 数组不同。

19.5.1 使用具体化函数

在程序清单 19.5 中，如果取消对 Animal 的构造函数和析构函数中打印语句的注释，将发现一些没有预料到的 Animals 的构造和析构。

将对象添加到数组中时，将调用该对象的默认构造函数；然而，Array 构造函数仍将零赋给数组的每个元素，如程序清单 19.5 的第 89 和 90 行所示。

代码 someAnimal = (Animal) 0;将调用 Animal 的默认 operator=。这导致使用接受一个 int 参数的构造函数来创建一个临时 Animal 对象。该临时对象用于 operator=的右边，赋值完毕后被销毁。

这是在令人遗憾地浪费时间，因为 Animal 对象已经被正确初始化。然而，不能删除这行代码，因为 int 不会被自动初始化为 0。解决方案是，让模板不要对 Animal 使用该构造函数，而使用一个专门针对 Animal 的构造函数。

可以显式地为 Animal 类型的 Array 提供一个构造函数实现，如程序清单 19.6 所示。这种具体化被称为模板具体化。

程序清单 19.6 具体化模板实现

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int DefaultSize = 3;
4:
5:  // A trivial class for adding to arrays
6:  class Animal
7:  {
8:    public:
9:      // constructors
10:     Animal(int);
11:     Animal();
12:     ~Animal();
13:
14:    // accessors
15:     int GetWeight() const { return itsWeight; }
16:     void SetWeight(int theWeight) { itsWeight = theWeight; }
17:
18:     // friend operators
19:     friend ostream& operator<< (ostream&, const Animal&);
```

```
20:
21:   private:
22:     int itsWeight;
23: };
24:
25: // extraction operator for printing animals
26: ostream& operator<<
27: (ostream& theStream, const Animal& theAnimal)
28: {
29:   theStream << theAnimal.GetWeight();
30:   return theStream;
31: }
32:
33: Animal::Animal(int weight):
34: itsWeight(weight)
35: {
36:   cout << "Animal(int) ";
37: }
38:
39: Animal::Animal():
40: itsWeight(0)
41: {
42:   cout << "Animal() ";
43: }
44:
45: Animal::~Animal()
46: {
47:   cout << "Destroyed an animal...";
48: }
49:
50: template <class T>   // declare the template and the parameter
51: class Array          // the class being parameterized
52: {
53:  public:
54:    Array(int itsSize = ::DefaultSize);
55:    Array(const Array &rhs);
56:    ~Array() { delete [] pType; }
57:
58:   // operators
59:    Array& operator=(const Array&);
60:    T& operator[](int offSet) { return pType[offSet]; }
61:    const T& operator[](int offSet) const
62:    { return pType[offSet]; }
63:
64:   // accessors
65:    int GetSize() const { return itsSize; }
66:    // friend function
67:    // template <class T>
68:    friend ostream& operator<< (ostream&, const Array<T>&);
69:
70:  private:
71:    T *pType;
```

```
72:     int itsSize;
73: };
74:
75: template <class T>
76: Array<T>::Array(int size = DefaultSize):
77: itsSize(size)
78: {
79:    pType = new T[size];
80:    for (int i = 0; i < size; i++)
81:       pType[i] = (T)0;
82: }
83:
84: template <class T>
85: Array<T>& Array<T>::operator=(const Array &rhs)
86: {
87:    if (this == &rhs)
88:       return *this;
89:    delete [] pType;
90:   itsSize = rhs.GetSize();
91:    pType = new T[itsSize];
92:    for (int i = 0; i < itsSize; i++)
93:       pType[i] = rhs[i];
94:    return *this;
95: }
96:
97: template <class T>
98: Array<T>::Array(const Array &rhs)
99: {
100:    itsSize = rhs.GetSize();
101:   pType = new T[itsSize];
102:    for (int i = 0; i < itsSize; i++)
103:       pType[i] = rhs[i];
104: }
105:
106:
107: template <class T>
108: ostream& operator<< (ostream& output, const Array<T>& theArray)
109: {
110:    for (int i = 0; i<theArray.GetSize(); i++)
111:       output << "[" << i << "] " << theArray[i] << endl;
112:    return output;
113: }
114:
115:
116: Array<Animal>::Array(int AnimalArraySize):
117: itsSize(AnimalArraySize)
118: {
119:    pType = new Animal[AnimalArraySize];
120: }
121:
122:
123: void IntFillFunction(Array<int>& theArray);
```

```
124: void AnimalFillFunction(Array<Animal>& theArray);
125:
126: int main()
127: {
128:    Array<int> intArray;
129:   Array<Animal> animalArray;
130:   IntFillFunction(intArray);
131:    AnimalFillFunction(animalArray);
132:    cout << "intArray...\n" << intArray;
133:    cout << "\nanimalArray...\n" << animalArray << endl;
134:    return 0;
135: }
136:
137: void IntFillFunction(Array<int>& theArray)
138: {
139:   bool Stop = false;
140:   int offset, value;
141:    while (Stop == false)
142:   {
143:      cout << "Enter an offset (0-2) and a value. ";
144:      cout << "(-1 to stop): " ;
145:     cin >> offset >> value;
146:     if (offset < 0)
147:        break;
148:     if (offset > 2)
149:     {
150:        cout << "***Please use values between 0 and 2.***\n";
151:        continue;
152:     }
153:      theArray[offset] = value;
154:   }
155: }
156:
157:
158: void AnimalFillFunction(Array<Animal>& theArray)
159: {
160:    Animal * pAnimal;
161:    for (int i = 0; i<theArray.GetSize(); i++)
162:   {
163:      pAnimal = new Animal(i*10);
164:     theArray[i] = *pAnimal;
165:     delete pAnimal;
166:   }
167: }
```

注意：如果读者使用的是 Microsoft 编译器，请取消对第 67 行的注释。根据 C++标准，这行代码是不必要的，然而使用 Microsoft C++编译器进行编译时，这行代码是必不可少的。

注意：为方便分析方，给输出添加了行号；但在输出中并没有行号。

输出：

第一次运行：

```
1: Animal() Animal() Animal() Enter an offset (0-2) and a value.
```

```
        (-1 to stop): 0 0
 2: Enter an offset (0-2) and a value. (-1 to stop):0 1
 3: Enter an offset (0-2) and a value. (-1 to stop):1 2
 4: Enter an offset (0-2) and a value. (-1 to stop):2 3
 5: Enter an offset (0-2) and a value. (-1 to stop): -1 -1
 6: animal(int) Destroyed an animal...animal(int) Destroyed an
       animal...animal(int) Destroyed an animal...initArray...
 7: [0] 0
 8: [1] 1
 9: [2] 2
10:
11: animal array...
12: [0] 0
13: [1] 10
14: [2] 20
15:
16: Destroyed an animal...Destroyed an animal...Destroyed an animal...
```

第二次运行：

```
 1: animal(int)  Destroyed an animal...
 2: animal(int)  Destroyed an animal...
 3: animal(int)  Destroyed an animal...
 4: Enter an offset (0-9) and a value. (-1 to stop):0 0
 5: Enter an offset (0-9) and a value. (-1 to stop):1 1
 6: Enter an offset (0-9) and a value. (-1 to stop):2 2
 7: Enter an offset (0-9) and a value. (-1 to stop):3 3
 8: animal(int)
 9: Destroyed an animal...
10: animal(int)
11: Destroyed an animal...
12: animal(int)
13: Destroyed an animal...
14: initArray...
15: [0] 0
16: [1] 1
17: [2] 2
18:
19: animal array...
20: [0] 0
21: [1] 10
22: [2] 20
23:
24: Destroyed an animal...
25: Destroyed an animal...
26: Destroyed an animal...
```

分析：

程序清单 19.6 取消了对 Animal 的构造函数和析构函数中打印语句的注释，让读者能够看到临时 Animal 对象的创建和销毁。为简化输出，将 DefaultSize 的值从 10 缩小到 3。

第 33～48 行的 Animal 构造函数和析构函数分别打印一条消息，指出它们被调用。

第 75～82 行定义了 Array 构造函数的模板行为。在第 116～120 行是一个专门用于 Animal 类型的 Array 的构造函数。注意，在这个特殊构造函数中，使用 Animal 的默认构造函数为每个 Animal 设置初始值，但没

有进行显式赋值。

该程序第一次运行时，显示第一部分输出。输出的第 1 行表明，创建 Animal 数组时调用了默认构造函数 3 次；接下来用户输入 4 个数，它们被存储到 int 数组中。

接下来执行 AnimalFillFunction()。第 163 行在堆上创建了一个临时 Animal 对象，第 164 行使用它来修改数组中的 Animal 对象。第 165 行将临时 Animal 对象销毁。对数组中的每个元素重复这一过程，如输出的第 6 行所示。

在程序的末尾，数组被销毁，导致对数组中每个对象调用析构函数，将它们全部销毁，如输出的第 16 行所示。

第二次运行程序时，将第 116～120 行的 Animal 类型的 Array 的专用构造函数实现注释掉了。在这种情况下，创建 Animal 数组时将调用第 75～82 行的模板构造函数。这导致对于数组中的每个成员，都将创建一个临时 Animal 对象（第 80 和 81 行），第二部分输出的第 1～3 行反映了这一点。

正如读者预期的，在其他方面，两次运行的输出相同。

19.5.2 静态成员和模板

模板可以声明静态数据成员，在这种情况下，将为根据模板可创建的每个类创建一组静态数据。也就是说，如果在 Array 模板中添加了一个静态成员（如记录创建了多少个数组的计数器），则每种类型的 Array 都有一个这样的成员：所有 Animals 数组共用一个，所有 int 数组共用一个。程序清单 19.7 在 Array 模板中添加了一个静态成员和一个静态函数。

程序清单 19.7 在模板中使用静态成员数据和函数

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int DefaultSize = 3;
4:
5:  // A trivial class for adding to arrays
6:  class Animal
7:  {
8:    public:
9:      // constructors
10:     Animal(int);
11:     Animal();
12:     ~Animal();
13:
14:    // accessors
15:     int GetWeight() const { return itsWeight; }
16:     void SetWeight(int theWeight) { itsWeight = theWeight; }
17:
18:     // friend operators
19:     friend ostream& operator<< (ostream&, const Animal&);
20:
21:   private:
22:     int itsWeight;
23: };
24:
25: // extraction operator for printing animals
26: ostream& operator<<
27:    (ostream& theStream, const Animal& theAnimal)
28: {
```

```
29:    theStream << theAnimal.GetWeight();
30:    return theStream;
31:  }
32:
33:  Animal::Animal(int weight):
34:    itsWeight(weight)
35:  {
36:    //cout << "animal(int) ";
37:  }
38:
39:  Animal::Animal():
40:    itsWeight(0)
41:  {
42:    //cout << "animal() ";
43:  }
44:
45:  Animal::~Animal()
46:  {
47:    //cout << "Destroyed an animal...";
48:  }
49:
50:  template <class T>   // declare the template and the parameter
51:  class Array          // the class being parameterized
52:  {
53:    public:
54:      // constructors
55:      Array(int itsSize = DefaultSize);
56:      Array(const Array &rhs);
57:      ~Array() { delete [] pType;  itsNumberArrays--; }
58:
59:      // operators
60:      Array& operator=(const Array&);
61:      T& operator[](int offSet) { return pType[offSet]; }
62:      const T& operator[](int offSet) const
63:      { return pType[offSet]; }
64:      // accessors
65:      int GetSize() const { return itsSize; }
66:      static int GetNumberArrays() { return itsNumberArrays; }
67:
68:      // friend function
69:      friend ostream& operator<< (ostream&, const Array<T>&);
70:
71:    private:
72:      T *pType;
73:      int  itsSize;
74:      static int itsNumberArrays;
75:  };
76:
77:  template <class T>
78:  int Array<T>::itsNumberArrays = 0;
79:
80:  template <class T>
```

```
81: Array<T>::Array(int size = DefaultSize):
82:   itsSize(size)
83: {
84:    pType = new T[size];
85:    for (int i = 0; i < size; i++)
86:      pType[i] = (T)0;
87:    itsNumberArrays++;
88: }
89:
90: template <class T>
91: Array<T>& Array<T>::operator=(const Array &rhs)
92: {
93:    if (this == &rhs)
94:       return *this;
95:    delete [] pType;
96:    itsSize = rhs.GetSize();
97:    pType = new T[itsSize];
98:    for (int i = 0; i < itsSize; i++)
99:      pType[i] = rhs[i];
100: }
101:
102: template <class T>
103: Array<T>::Array(const Array &rhs)
104: {
105:    itsSize = rhs.GetSize();
106:    pType = new T[itsSize];
107:    for (int i = 0; i < itsSize; i++)
108:       pType[i] = rhs[i];
109:   itsNumberArrays++;
110: }
111:
112: template <class T>
113: ostream& operator<< (ostream& output, const Array<T>& theArray)
114: {
115:    for (int i = 0; i < theArray.GetSize(); i++)
116:       output << "[" << i << "] " << theArray[i] << endl;
117:    return output;
118: }
119:
120: int main()
121: {
122:   cout << Array<int>::GetNumberArrays() << " integer arrays\n";
123:   cout << Array<Animal>::GetNumberArrays();
124:   cout << " animal arrays" << endl << endl;
125:   Array<int> intArray;
126:    Array<Animal> animalArray;
127:
128:    cout << intArray.GetNumberArrays() << " integer arrays\n";
129:    cout << animalArray.GetNumberArrays();
130:    cout << " animal arrays" << endl << endl;
131:
132:    Array<int> *pIntArray = new Array<int>;
```

```
133:
134:    cout << Array<int>::GetNumberArrays() << " integer arrays\n";
135:    cout << Array<Animal>::GetNumberArrays();
136:    cout << " animal arrays" << endl << endl;
137:
138:     delete pIntArray;
139:
140:    cout << Array<int>::GetNumberArrays() << " integer arrays\n";
141:    cout << Array<Animal>::GetNumberArrays();
142:    cout << " animal arrays" << endl << endl;
143:     return 0;
144: }
```

输出：

```
0 integer arrays
0 animal arrays

1 integer arrays
1 animal arrays

2 integer arrays
1 animal arrays

1 integer arrays
1 animal arrays
```

分析：

第 74 行在 Array 模板中添加了静态变量 itsNumberArrays，由于该数据是私有的，因此第 66 行添加了一个静态公有存取器函数 GetNumberArrays( )。

对静态数据进行初始化，用模板对其进行了全限定，如第 77 和 78 行所示。对 Array 的构造函数和析构函数都进行了修改，以记录当前有多少个数组。

访问模板静态成员的方法与访问其他类的静态成员相同：可以像第 134 和 135 行那样通过对象来访问，也可以像第 128 和 129 行通过类来访问。注意，访问静态数据时，必须通过特定类型的数组。每种类型的数组都有一个静态变量 itsNumberArrays。

应该：

当一个概念适用于不同类（或基本数据类型）的对象时，应使用模板来实现它。

应使用模板函数参数将其实例限制为类型安全的。

必要时应在模板中使用静态成员。

应根据类型覆盖模板的函数，以具体化模板的行为。

不应该：

不要就此结束对模板的学习。本章只介绍了模板的部分用途，对模板的详细介绍超出了本书的范围。

即使没有完全理解如何创建自己的模板也不要着急，知道如何使用模板更重要。读者在下一节将看到，STL 包含大量可供你使用的模板。

19.6 标准模板库

俗话说得好，做重复的劳动毫无意义；使用 C++进行编程时也不例外。这就是标准模板库（STL）得以

流行的原因。和标准 C++库的其他组成部分一样，STL 也是可以在各种操作系统之间移植的。

现在，所有的主流编译器厂商都在其编译器中提供了 STL。STL 是一个基于模板的容器类库，其中包括向量、链表、队列和堆栈。STL 还包含大量常用的算法，如排序和搜索。

STL 旨在避免你为满足常见需求而做重复的工作。STL 经过了测试和调试，提供了很高的性能且是免费的。最重要的是，STL 是可重用的：知道如何使用某个 STL 容器后，便可以在所有的程序中使用它而无需重新开发。

19.6.1 使用容器

容器是用于存储其他对象的对象。标准 C++库提供了一系列的容器类，它们是功能强大的工具，可帮助 C++开发人员应对常见的编程任务。

标准模板库容器类分两种：顺序容器和关联容器。顺序容器支持对其成员（元素）的顺序和随机访问。关联容器经过了优化，可根据键值访问其元素。所有的 STL 容器类都是在名称空间 std 中定义的。

19.6.2 理解顺序容器

标准模板库中的顺序容器提供了对一系列对象进行顺序访问的高效途径。标准 C++库提供了 5 种顺序容器：向量、链表、堆栈、双端队列和队列。

1. 向量容器

你经常使用数组来存储和访问大量的元素。同一个数组中的元素都是同一种类型的，可以使用下标来访问。STL 提供了一个 vector 容器类，其行为类似于数组，但相对于标准 C++数组功能更强大，使用起来更安全。

vector 是经过优化的容器，提供了通过索引快速访问其元素的功能。容器类 vector 是在位于名称空间 std 中的头文件<vector>中定义的（有关名称空间用途的更详细信息，请参阅第 18 章）。

向量在必要时会自动增大。假设你创建了一个包含 10 个元素的向量。用 10 个对象填充后，该向量便填满了。如果这时再添加一个对象，向量将自动增大其容量，以便能够容纳第 11 个对象。下面的代码说明了 vector 类是如何被定义的：

```
template <class T, class Allocator = allocator<T>> class vector
{
    //class members
};
```

第一个参数（class T）是向量中元素的类型；第二个参数（class Allocator）是一个分配器类。分配器是负责为容器中的元素分配和释放内存的内存管理器。分配器的概念和实现属于高级主题，超出了本书的范围。

默认情况下，使用运算符 new()来创建元素，使用运算符 delete()来释放。也就是说，调用类 T 的默认构造函数创建新元素。这意味着用户必须显式地为自己创建的类定义默认构造函数；如果不这样做，你将无法使用标准向量容器来存储自己创建的类的实例。

可以这样定义用于存储整数和浮点数的向量：

```
vector<int> vInts;        //vector holding int elements
vector<float> vFloats;    //vector holding float elements
```

通常，你对向量将包含多少个元素有一定的了解。例如，假设在你的学校，每班的学生最多为 50 名。为创建用于存储一个班的学生的向量，该向量必须能够存储 50 个元素。标准向量类提供了一个这样的构造函数：接受元素数目作为其参数。因此可以这样定义可存储 50 名学生的向量：

```
vector<Student> MathClass(50);
```

编译器将分配足以存储 50 个 Student 对象的内存空间，其中每个元素都是使用默认构造函数 Student::Student()创建的。

向量中的元素数目可使用成员函数 size( )来检索。对于刚才定义的 Student 向量 MathClass，MathClass.size( ) 返回 50。

成员函数 capacity( )指出向量能容纳多少个元素，包含的元素达到这个数目后将增大向量。这将在后面详细介绍。

如果向量中没有元素，则称向量为空，即向量的大小为 0。为方便检测向量是否为空，vector 类提供了成员函数 empty( )，如果向量为空，则其返回值为 true。

要将 Student 对象 Harry 赋给 MathClass，可使用下标运算符[ ]：

```
MathClass[5] = Harry;
```

下标从 0 开始。读者可能注意到了，这里使用 Student 类的重载赋值运算符将 Harry 赋给 Mathclass 的第 6 个元素。同样，要获悉 Harry 的年龄，可使用下面的方式访问其记录：

```
MathClass[5].GetAge();
```

正如前面指出的，当添加的元素数超过向量的处理能力时，向量将自动增大。例如，假设在你的学校中，某门课很受欢迎，学生人数超过了 50。当然这种情况也许不会在数学课中发生，但谁知道呢，奇怪的事情确实发生了。将第 51 个学生 Sally 添加到 MathClass 时，它将自动扩大以容纳 Sally。

将元素添加到向量中的方式有多种，其中之一是使用 push_back( )：

```
MathClass.push_back(Sally);
```

该成员函数将新 Student 对象 Sally 放在向量 MathClass 的末尾。现在 MathClass 有 51 个元素，Sally 存储在 MathClass[50]中。

为使该函数能够正确工作，Student 类必须定义一个复制构造函数；否则 push_back( )函数将无法复制对象 Sally。

STL 没有规定向量的最大元素数，这由编译器厂商决定。vector 类提供了成员函数 max_size( )，它返回编译器指定的这个神奇数字。

程序清单 19.8 演示了迄今为止讨论的 vector 类成员。在该程序清单中，使用了标准 string 类来简化字符串处理工作。有关 string 类的详情，请参阅编译器文档。

程序清单 19.8 创建向量及访问其中的元素

```
 0: #include <iostream>
 1: #include <string>
 2: #include <vector>
 3: using namespace std;
 4:
 5: class Student
 6: {
 7:   public:
 8:     Student();
 9:     Student(const string& name, const int age);
10:     Student(const Student& rhs);
11:     ~Student();
12:
13:     void    SetName(const string& name);
14:     string  GetName()  const;
15:     void    SetAge(const int age);
16:     int     GetAge()   const;
17:
18:     Student& operator=(const Student& rhs);
19:
20:   private:
```

```
21:     string itsName;
22:     int itsAge;
23: };
24:
25: Student::Student()
26: : itsName("New Student"), itsAge(16)
27: {}
28:
29: Student::Student(const string& name, const int age)
30: : itsName(name), itsAge(age)
31: {}
32:
33: Student::Student(const Student& rhs)
34: : itsName(rhs.GetName()), itsAge(rhs.GetAge())
35: {}
36:
37: Student::~Student()
38: {}
39:
40: void Student::SetName(const string& name)
41: {
42:    itsName = name;
43: }
44:
45: string Student::GetName() const
46: {
47:    return itsName;
48: }
49:
50: void Student::SetAge(const int age)
51: {
52:    itsAge = age;
53: }
54:
55: int Student::GetAge() const
56: {
57:    return itsAge;
58: }
59:
60: Student& Student::operator=(const Student& rhs)
61: {
62:    itsName = rhs.GetName();
63:    itsAge = rhs.GetAge();
64:    return *this;
65: }
66:
67: ostream& operator<<(ostream& os, const Student& rhs)
68: {
69:    os << rhs.GetName() << " is " << rhs.GetAge() << " years old";
70:    return os;
71: }
72:
```

```
73: template<class T>
74: // display vector properties
75: void ShowVector(const vector<T>& v);
76:
77: typedef vector<Student>  SchoolClass;
78:
79: int main()
80: {
81:    Student Harry;
82:    Student Sally("Sally", 15);
83:    Student Bill("Bill", 17);
84:    Student Peter("Peter", 16);
85:
86:    SchoolClass  EmptyClass;
87:    cout << "EmptyClass:" << endl;
88:    ShowVector(EmptyClass);
89:
90:    SchoolClass GrowingClass(3);
91:    cout << "GrowingClass(3):" << endl;
92:    ShowVector(GrowingClass);
93:
94:    GrowingClass[0] = Harry;
95:    GrowingClass[1] = Sally;
96:    GrowingClass[2] = Bill;
97:    cout << "GrowingClass(3) after assigning students:" << endl;
98:    ShowVector(GrowingClass);
99:
100:   GrowingClass.push_back(Peter);
101:   cout << "GrowingClass() after added 4th student:" << endl;
102:   ShowVector(GrowingClass);
103:
104:   GrowingClass[0].SetName("Harry");
105:   GrowingClass[0].SetAge(18);
106:   cout << "GrowingClass() after Set\n:";
107:   ShowVector(GrowingClass);
108:
109:   return 0;
110: }
111:
112: //
113: // Display vector properties
114: //
115: template<class T>
116: void ShowVector(const vector<T>& v)
117: {
118:    cout << "max_size() = " << v.max_size();
119:    cout << "\tsize() = " << v.size();
120:    cout << "\tcapacity() = " << v.capacity();
121:    cout << "\t" << (v.empty()? "empty": "not empty");
122:    cout << endl;
123:
124:    for (int i = 0; i < v.size(); ++i)
```

```
125:        cout << v[i] << endl;
126:
127:    cout << endl;
128: }
```

输出：

```
EmptyClass:
max_size() = 214748364  size() = 0      capacity() = 0  empty

GrowingClass(3):
max_size() = 214748364  size() = 3      capacity() = 3  not empty
New Student is 16 years old
New Student is 16 years old
New Student is 16 years old
GrowingClass(3) after assigning students:
max_size() = 214748364  size() = 3      capacity() = 3  not empty
New Student is 16 years old
Sally is 15 years old
Bill is 17 years old

GrowingClass() after added 4th student:
max_size() = 214748364  size() = 4      capacity() = 6  not empty
New Student is 16 years old
Sally is 15 years old
Bill is 17 years old
Peter is 16 years old

GrowingClass() after Set:
max_size() = 214748364  size() = 4      capacity() = 6  not empty
Harry is 18 years old
Sally is 15 years old
Bill is 17 years old
Peter is 16 years old
```

分析：

Student 类在第 5～23 行定义的，其成员函数的实现位于第 25～65 行。这是一个简单的适合用向量容器来存储的类。基于前面讨论过的原因，定义默认构造函数、复制构造函数和重载的赋值运算符。注意，其成员变量 itsName 被定义为一个 string 类实例。从中可知，使用 C++ string 比使用 C-风格字符串 char*容易得多。

模板函数 ShowVector( )在第 73 和 75 行声明的，其实现位于第 115～128 行。它演示了一些向量成员函数的用法：maxsize( )、size( )、capacity( )和 empty( )。从输出可知，在 Visual C++中，向量最多能够容纳 214 748 364 个 Student 对象。对于其他类型的元素，最多可容纳的数量可能不同。例如，int 向量最多可容纳 1 073 741 823 个元素。如果你使用的是其他编译器，可容纳的最大元素数可能与此不同。

第 124 和 125 行使用第 67～71 行定义的重载插入运算符（<<）显示每个元素的值。

在 main( )函数中，第 81～84 行创建了 4 个 Student 对象。第 86 行使用 vector 类的默认构造函数定义了一个名为 EmptyClass 的空向量。以这种方式创建向量时，编译器不会为其分配内存空间。从 ShowVector(EmptyClass)的输出可知，EmptyClass 的大小和容量都是 0。

第 90 行定义了一个可存储 3 个 Student 对象的向量，其大小和容量都是 3。第 94～96 行使用下标运算符[]将 Student 对象赋给 GrowingClass 的元素。

第 100 行将第 4 个 Student 对象 Peter 添加到向量中。这使向量的大小增加到 4。有趣的是，此时其容量被设置为 6。这意味着编译器分配了足以容纳 6 个 Student 对象的内存空间。

由于必须给向量分配连续的内存块，因此扩大它们时需要执行一系列的操作。首先，新分配一个足以容纳 4 个 Student 对象的新内存块；然后将原来的 3 个元素复制到新分配的内存中，并将第 4 个元素存储到第 3 个元素的后面；最后，将原来的内存块释放。当向量中包含大量的元素时，这种释放和重新分配的过程将耗用大量的时间。因此，编译器采用一种优化策略来降低这些昂贵操作的可能性。在这个例中，如果在向量中再添加 1 或 2 个对象，将不需要释放和重新分配内存。

第 104 和 105 行再次使用下标运算符[ ]来修改 GrowingClass 中第一个对象的成员变量。

应该：

如果类的实例可能被存储到向量中，应为其定义默认构造函数。

还应为前面所说的类定义一个复制构造函数。

还应为前面所说的类定义一个重载的赋值运算符。

不应该：

不要创建自己的向量类，可以使用 STL 中的 vector 类。由于它是 C++标准的组成部分，因此遵循该标准的编译器应该提供了 vector 类。

容器类 vector 还有其他成员函数。函数 front( )返回对向量中第一个元素的引用；函数 back( )返回对向量中最后一个元素的引用；函数 at( )的作用与下标运算符[ ]相同，但比向量的[ ]实现更安全，因为它检查传递给它的下标是否在有效的范围内（当然，你也可以让下标运算符也执行这样的检查）。如果不在有效范围内，它将引发 outofrangge 异常（异常将在下一章介绍）。

函数 insert( )在向量的指定位置插入一个或多个节点；函数 popback( )删除向量中的最后一个元素；函数 remove( )从向量中删除一个或多个元素。

2. 链表容器

链表是需要频繁插入或删除元素时最适合使用的容器。STL 容器类 list 是在位于名称空间 std 中的头文件<list>中定义的。list 类是以双向链接的方式实现的，其中每个节点都有指向链表中前一个节点和下一个节点的链接。

vector 类有的成员函数在 list 类中都有。正如读者在“第 2 周复习”中看到的，可以沿每个节点提供的链接来遍历链表。链接通常是使用指针实现的，标准容器类 list 使用一种被称为迭代器的机制来实现链接。

迭代器是一种广义指针，使用它旨在避免指针带来的一些危险。

可以对迭代器解除引用来获得它指向的节点。程序清单 19.9 演示如何使用迭代器来访问链表中的节点。

程序清单 19.9 使用迭代器来遍历链表

```
 0:  #include <iostream>
 1:  #include <list>
 2:  using namespace std;
 3:
 4:  typedef list<int> IntegerList;
 5:
 6:  int main()
 7:  {
 8:     IntegerList  intList;
 9:
10:     for (int i = 1; i <= 10; ++i)
11:        intList.push_back(i * 2);
12:
13:     for (IntegerList::const_iterator ci = intList.begin();
14:                      ci != intList.end(); ++ci)
```

```
15:       cout << *ci << " ";
16:
17:    return 0;
18: }
```

输出：

```
2 4 6 8 10 12 14 16 18 20
```

分析：

程序清单 19.9 使用了 STL 中的链表模板。第 1 行使用#include 包含了必要的头文件，这导入了 STL 中链表模板的代码。

第 4 行使用 typedef 命令让你能够在程序清单中使用 IntegerList 而不是 list<int>，从而提高了程序的易读性。

第 8 行将 intList 声明为一个前面使用 typedef 定义的 IntegerList。第 10 和 11 行使用 push_back( ) 函数将前 10 个正偶数添加到该链表中。

第 13～15 行使用一个常量迭代器访问链表中的每个节点。这表明我们不打算通过该迭代器来修改节点。如果要修改迭代器指向的节点，应使用非常量迭代器：

```
intList::iterator
```

成员函数 begin( )返回了一个指向链表中第一个节点的迭代器。正如读者看到的，可以使用递增运算符++来让迭代器指向下一个节点。成员函数 end( )有点奇怪：它返回一个指向链表中最后一个节点后面的一个节点的迭代器。必须确保迭代器不到达 end( )返回的位置。

对迭代器解除引用（以返回它指向的节点）和指针一样，如第 15 行所示。

虽然这里是在讨论 list 类时介绍的迭代器，但 vector 类也提供了迭代器。除拥有 vector 类拥有的所有函数外，list 类还提供了 push_front( )和 pop_front( )函数，它们的功能与 push_back( )和 pop_back( )相同，只是它们不是在链表的末尾而是开头添加和删除元素。

3. 堆栈容器

在计算机编程中，最常用的数据结构之一是堆栈。然而，堆栈不是作为独立的容器类来实现的，而是作为容器包装器（也就是模板——译者注）来实现的。模板类 stack 是在名称空间 std 中的头文件<stack>中定义的。

为堆栈分配一个连续的内存块，可在堆栈的末尾增删元素端来增大或缩小它。只能访问或删除堆栈末尾的元素。顺序容器有类似的特征，尤其是 vector 和 deque。实际上，任何支持 back( )、push_back( )和 pop_back( )操作的顺序容器都可用来实现堆栈。堆栈不需要其他大部分容器方法，因此堆栈不暴露它们。

STL 模板类 stack 可存储任何类型的对象，惟一的限制是所有的元素都必须是同一种类型。

堆栈是一种 LIFO（后进先出）结构。对于堆栈，一个经典类比是：堆栈就像一叠盘子，在最上面加入盘子（压入堆栈），从最上面取盘子（取出最后加入的盘子，弹出堆栈）。

根据约定，堆栈的开口端通常被称为栈顶；对堆栈执行的操作通常被称为压入和弹出。stack 类采用了这些惯用术语。

注意：STL 中的 stack 类与编译器和操作系统使用的堆栈机制不同，在编译器和操作系统中，堆栈可以存储不同类型的对象。然而，它们基本功能极其相似。

4. 双端队列容器

双端队列类似于双向向量，它继承了容器类 vector 在顺序读写方面的效率。然而，容器类 deque 还提供了优化的前端和后端操作。这些操作的实现类似于容器类 list：只为新元素分配内存。deque 类的这种特性避免了像 list 类那样重新为整个容器分配内存。因此，双端队列非常适合在一端或两端进行插入和删除以及顺

序访问元素至关重要的应用。一个这样应用范例是列车组装模拟程序，在这种程序中，可以在列车两端添加车厢。

5. 队列容器

队列是计算机编程中另一种常用的数据结构。在队列的一端添加元素，从另一端取出元素。

队列像剧院前的购票队伍：人从队列的后面进入，从队列前面离开。这被称为 FIFO（先入先出）结构；堆栈是一种 LIFO（后进先出）结构。当然，有时当你在超市中排在一列长队的倒数第二个时，这时新开了一个收款台并将队伍中的最后一个人拉了过去——将本来应该是一个先进先出的队列变成了后进先出的堆栈，这让你恨得咬牙切齿。

和 stack 一样，queue 也是以容器包装器的方式实现的。容器必须支持 front( )、back( )、push_back( ) 和 pop_front( ) 操作。

19.6.3 理解关联容器

正如读者看到的，向量类似于改进的数组，它具备数组的所有特征，还新增了其他特性。不幸的是，向量也有数组的弱点之一：不能根据除下标之外的其他索引来查找元素。关联容器提供了根据关键字来快速存取值的功能。

顺序容器被设计为使用索引或迭代器来顺序和随机存取元素，而关联容器被设计为使用关键字来快速随机存取元素。标准 C++库提供了 5 种关联容器：map、multimap、set、multiset 和 bitset。

1. 映射容器

读者将学习的第一种关联容器是映射（map）。该名称源自它们包含映射：从关键字到相关值的映射，就像地图上的一点对应于地球上的某个地方。在下面的范例（程序清单 19.10）中，使用映射来实现了程序清单 19.8 中的 SchoolClass。

程序清单 19.10 容器类 map

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

class Student
{
  public:
    Student();
    Student(const string& name, const int age);
    Student(const Student& rhs);
    ~Student();

    void   SetName(const string& name);
    string  GetName()  const;
    void   SetAge(const int age);
    int     GetAge()  const;

    Student& operator=(const Student& rhs);

  private:
    string itsName;
    int itsAge;
```

```
23: };
24:
25: Student::Student()
26: : itsName("New Student"), itsAge(16)
27: {}
28:
29: Student::Student(const string& name, const int age)
30: : itsName(name), itsAge(age)
31: {}
32:
33: Student::Student(const Student& rhs)
34: : itsName(rhs.GetName()), itsAge(rhs.GetAge())
35: {}
36:
37: Student::~Student()
38: {}
39:
40: void Student::SetName(const string& name)
41: {
42:    itsName = name;
43: }
44:
45: string Student::GetName() const
46: {
47:    return itsName;
48: }
49:
50: void Student::SetAge(const int age)
51: {
52:    itsAge = age;
53: }
54:
55: int Student::GetAge() const
56: {
57:    return itsAge;
58: }
59:
60: Student& Student::operator=(const Student& rhs)
61: {
62:    itsName = rhs.GetName();
63:    itsAge = rhs.GetAge();
64:    return *this;
65: }
66:
67: ostream& operator<<(ostream& os, const Student& rhs)
68: {
69:    os << rhs.GetName() << " is " << rhs.GetAge() << " years old";
70:    return os;
71: }
72:
73: template<class T, class A>
74: void ShowMap(const map<T, A>& v);    // display map properties
```

```
75:
76: typedef map<string, Student>  SchoolClass;
77:
78: int main()
79: {
80:   Student Harry("Harry", 18);
81:   Student Sally("Sally", 15);
82:   Student Bill("Bill", 17);
83:   Student Peter("Peter", 16);
84:
85:   SchoolClass  MathClass;
86:   MathClass[Harry.GetName()] = Harry;
87:   MathClass[Sally.GetName()] = Sally;
88:   MathClass[Bill.GetName()] = Bill;
89:   MathClass[Peter.GetName()] = Peter;
90:
91:   cout << "MathClass:" << endl;
92:   ShowMap(MathClass);
93:
94:   cout << "We know that " << MathClass["Bill"].GetName()
95:        << " is " << MathClass["Bill"].GetAge()
96:        << " years old" << endl;
97:   return 0;
98: }
99:
100: //
101: // Display map properties
102: //
103: template<class T, class A>
104: void ShowMap(const map<T, A>& v)
105: {
106:   for (map<T, A>::const_iterator ci = v.begin();
107:             ci != v.end(); ++ci)
108:     cout << ci->first << ": " << ci->second << endl;
109:
110:   cout << endl;
111: }
```

输出：

```
MathClass:
Bill: Bill is 17 years old
Harry: Harry is 18 years old
Peter: Peter is 16 years old
Sally: Sally is 15 years old

We know that Bill is 17 years old
```

分析：

在这个例子中，创建了一个 SchoolClass 对象，并在其中添加了 4 个 Student 对象。然后，打印这些 Student 对象的内容。接下来打印 Student 对象 Bill 的年龄，然后打印年龄时，不是像前一个范例那样使用数字索引，而是对象的名称 Bill。这是因为这里使用了 map 模板。

仔细查看将发现，该程序清单的大部分代码为 Student 类，现在读者应该能够理解这些代码。

第 2 行包含了头文件<map>，因为要使用标准容器类 map。第 73 行是函数 ShowMap 的原型，这是一个模板函数，用于显示映射容器中的元素。

第 76 行使用 typedef 将 SchoolClass 定义为一种映射，每个元素都由一个关键字-值对组成。其中第一个为关键字，它是 Student 对象的名称，类型为 string；第二个为值，它是实际的 Student 对象。在映射容器中，元素的关键字必须是独一无二的，即任何两个元素的关键字都不能相同。在 STL 中，关键字-值是用包含两个成员的结构来实现的：first 和 second。可通过这些成员来访问节点的关键字和值。

来看看第 103～111 行的 ShowMap( )函数，它使用一个常量迭代器来访问映射容器中的元素。在第 108 行，ci->first 指向关键字（Student 对象的名称）；ci->second 指向 Student 对象本身。

现在只余下第 78～98 行的 main( )函数没有介绍。第 80～83 行创建了 4 个 Student 对象。第 85 行将 MathClass 声明为一个 SchoolClass 实例。在第 86～89 行使用下面的语法将 4 个 Student 对象添加到 MathClass 中：

```
map_object[key value] = object_value;
```

从第 86 行可知，使用的 key_value 为 Student 对象的名称，这是使用 Student 类的 GetName( )方法获得的；object_value 是 Student 对象。

也可以使用 push_back( )或 insert( )函数将关键字-值对添加到映射容器中，详情请参阅编译器文档。

将所有 Student 对象加到映射容器中后，可以使用关键字来访问其中的任何对象。第 94 和 95 行使用 MathClass["Bill"]来检索 Bill 的年龄，其中 Bill 是关键字。同样，可以使用其他 Student 对象的名称来访问其年龄。

2. 其他关联容器

容器类 multimap 是一个不要求关键字是独一无二的 map 类，多个元素可以有相同的关键字。

容器类 set 也类似于 map 类，惟一的差别是，其元素不是由关键字-值对组成，而只有关键字。容器类 multiset 是允许关键字重复的 set 类。

最后，容器类 bitset 是用于存储一系列位的模版。

19.6.4 使用算法类

容器非常适合用于存储一系列元素。所有标准容器都定义了操纵容器及其元素的操作。然而，自己实现所有这些操作很费力，也容易错误。因为对大部分容器来说，大部分操作都是相同的，一组通用算法可避免为每个新容器编写操作。标准库提供了大约 60 种标准算法，它们执行最基本、最常见的容器操作。

标准算法是在名称空间 std 中的头文件<algorithm>中定义的。

要理解标准算法的工作原理，必须理解函数对象的概念。函数对象是这样一个类的实例：它定义了重载的 operator( )，因此像函数那样调用它。程序清单 19.11 演示了函数对象的用法。

程序清单 19.11 函数对象

```
 0:  #include <iostream>
 1:  using namespace std;
 2:
 3:  template<class T>
 4:  class Print
 5:  {
 6:    public:
 7:      void operator()(const T& t)
 8:      {
 9:        cout << t << " ";
10:      }
11:  };
```

```
12:
13: int main()
14: {
15:    Print<int> DoPrint;
16:    for (int i = 0; i < 5; ++i)
17:       DoPrint(i);
18:    return 0;
19: }
```

输出：

```
0 1 2 3 4
```

分析：

第 3～10 行定义了一个名为 Print 的模板类，正如读者看到的，这是一个标准模板类。第 6～19 行将 operator( )重载为接受一个对象作为参数，并将其输出到标准输出设备。第 15 行将 DoPrint 定义为一个 int 类型的 Print 实例。然后便可以像使用函数一样使用 DoPrint 来打印任何整数值，如第 17 行所示。标准算法类类似于 Print 类，它们重载了 operator( )，让你能够像使用函数一样使用它们的对象。

1. 非修改型操作

非修改型操作来自算法库，它们执行的不修改元素的操作。这些操作包括 for_each( )、find( )、search( )和 count( )等。程序清单 19.12 演示了如何使用函数对象和 for_each 算法来打印向量中的元素。

程序清单 19.12 使用 for_each( )算法

```
0: #include <iostream>
1: #include <vector>
2: #include <algorithm>
3: using namespace std;
4:
5: template<class T>
6: class Print
7: {
8:   public:
9:     void operator()(const T& t)
10:    {
11:      cout << t << " ";
12:    }
13: };
14:
15: int main()
16: {
17:   Print<int>  DoPrint;
18:   vector<int>  vInt(5);
19:
20:   for (int i = 0; i < 5; ++i)
21:     vInt[i] = i * 3;
22:
23:   cout << "for_each:" << endl;
24:   for_each(vInt.begin(), vInt.end(), DoPrint);
25:   cout << endl;
26:
27:   return 0;
```

```
28: }
```

输出：

```
for_each()
0 3 6 9 12
```

分析：

注意，所有 C++标准算法都是在<algorithm>中定义的，因此第 2 行包含了这个文件。虽然其中的大部分代码都很简单，但有一行有必要解释。第 24 行调用 for_each( )函数来遍历向量 vInt 中的每个元素。对每个元素，它调用函数对象 DoPrint 并将该元素传递给 DoPrint.operator( )，从而将元素的值打印在屏幕上。

2. 修改型操作

修改型操作执行修改容器中元素的操作，其中包括填充和重新排序。程序清单 19.13 演示了 fill( )算法。

程序清单 19.13　修改型算法

```
 0: #include <iostream>
 1: #include <vector>
 2: #include <algorithm>
 3: using namespace std;
 4:
 5: template<class T>
 6: class Print
 7: {
 8:   public:
 9:     void operator()(const T& t)
10:     {
11:       cout << t << " ";
12:     }
13: };
14:
15: int main()
16: {
17:   Print<int>  DoPrint;
18:   vector<int> vInt(20);
19:
20:   fill(vInt.begin(), vInt.begin() + 5, 1);
21:   fill(vInt.begin() + 5, vInt.end(), 2);
22:
23:   for_each(vInt.begin(), vInt.end(), DoPrint);
24:   cout << endl << endl;
25:
26:   return 0;
27: }
```

输出：

```
1 1 1 1 1 2 2 2 2 2
```

分析：

在该程序清单中，惟一的新内容是第 20 和 21 行，这里使用了 fill( )算法。fill( )算法用指定的值填充容器中的元素。第 20 行将整数 1 赋给 vInt 中的前 5 个元素；第 21 行将整数 2 赋给 vInt 的后 5 个元素。

3. 排序及相关操作

第三类算法是排序及相关操作，这包括合并、部分排序、进行复制的部分排序、二分法搜索、下限和上限检查、计算交集、计算差集、计算最小值、计算最大值、置换等。有关这些操作的详情，请参阅编译器文档或C++标准文档。

注意：详细介绍每种算法及其他STL类超出了本书的范围。有关可用的类和算法以及它们的参数和用法，请参阅编译器文档或C++标准文档。另外，市面上有专门介绍STL及其用法的图书。

19.7 小 结

本章介绍了如何创建和使用模板。模板是C++标准的重要组成部分和C++内置的工具。模板用于创建参数化类型：根据创建时被传递的参数改变其行为的类型。这是一种安全而高效地重用代码的方式。

模板的定义决定了参数化类型。模板的每个实例是一个对象，可以像其他的对象一样使用：作为函数的参数、作为返回值等。

模板类可声明3种友元函数：非模板友元函数、通用模板友元函数、特定类型模板友元函数。模板可声明静态数据成员，这样，模板的每个实例都有自己的一组静态数据。

要根据实际类型具体化模板函数的行为，可以用特定类型作为参数覆盖模板函数。这一点也适用于成员函数。

在本章的后半部分，介绍了标准模板库（STL）。STL提供了大量的模板和算法供你使用。

19.8 问 与 答

问：在使用宏可行时为何要使用模板？

答：模板是类型安全的且是C++语言内置的，因此编译器将对它们进行检查——至少在你实例化模板类来创建变量时会这样做。

问：模板函数的参数化类型和常规函数的参数有何不同？

答：常规函数对其接受的参数进行处理。模板函数让你能够参数化参数的类型。也就是说，可以将函数参数的类型声明为Array<Type>，然后根据传递给它的实际参数是什么类型的Array模板实例来决定Type的类型。

问：什么时候使用模板？什么时候使用继承性？

答：除类处理的元素类型外，所有行为或几乎所有行为都不变时使用模板。如果需要复制类代码，且只修改一个或多个成员的类型，应考虑使用模板。另外，在试图修改类，使之将祖先类对象作为操作数时（降低类型安全性）或使两个不相关的类有共同的祖先以便你的类能够与它们协同工作（也会降低类型安全性）时，应使用模板。

问：什么时候使用通用模板友元类？

答：当每个实例（无论其类型是什么）都应该为当前类的友元时。

问：什么时候使用特定类型的模板友元类或函数？

答：要在两个类之间建立一对一的关系时。例如array<int>应对应于iterator<int>，而不是iterator<Animal>。

19.9 作 业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量

先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

19.9.1 测验

1. 模板和宏之间有何区别？
2. 模板参数和函数参数之间有何不同？
3. 特定类型的模板友元类和通用模板友元类之间有何区别？
4. 是否可以为模板的某个实例提供特殊行为而不给其他实例提供？
5. 如果在模板类定义中声明了一个静态成员，将创建多少个静态变量？
6. 要将类对象存储到标准容器中，该类必须具备什么样的属性？
7. STL 表示什么？为何 STL 至关重要？

19.9.2 练习

1. 将下面的 List 类转换为模板：

```
class List
{
   private:

   public:
            List():head(0),tail(0),theCount(0) {}
            virtual ~List();
            void insert( int value );
            void append( int value );
            int is_present( int value ) const;
            int is_empty() const { return head == 0; }
            int count() const { return theCount; }
   private:
            class ListCell
            {
            public:
                  ListCell( int value, ListCell *cell = 0):val(value),
                        next(cell){}
                  int val;
                  ListCell *next;
            };
            ListCell *head;
            ListCell *tail;
            int theCount;
};
```

2. 提供 List 类的实现。
3. 提供 List 模板的实现。
4. 声明 3 个 List 对象：一个 String 链表、一个 Cat 链表和一个 int 链表。
5. 查错：下面的代码有什么错误（假设 List 模板已经定义，Cat 是本书前面定义的类）？

```
List<Cat> Cat_List;
Cat Felix;
CatList.append( Felix );
cout << "Felix is "
     << ( Cat_List.is_present( Felix ) ) ? "" : "not "
     << "present" << endl;
```

提示：是什么使 cat 不同于 int?

6. 为 List 声明友元函数 operator==。
7. 为 List 实现友元函数 operator==。
8. operator==存在练习 5 中的问题吗?
9. 实现模板成员函数 swap( )，它交换两个变量的值。