第18章 模　板

模板是C++中最复杂和性能最强大的工具之一。尽管最初C++不包括模板，但模板在几年前已被添加到C++中并被所有现代的C++编译器所支持。使用模板，可以创建通用的函数和类。在通用的函数和类中，函数或类操作的数据类型被指定为一个参数。因此，可以使用一个带多种不同数据类型的函数或类，而不必针对每一个数据类型明确地对具体版本进行再编码。本章将对通用函数和类进行讨论。

18.1 通用函数

通用函数定义一组应用于各种数据类型的普通操作，函数操作的数据类型作为参数传递给该函数。通过利用通用函数，可以把一个通用的过程应用于各种不同的数据。或许你已经知道，许多算法操作的数据类型虽然不同，但算法在逻辑上是相同的。例如，无论把快速排序算法应用于整型数据，还是应用于浮点型数据，该算法逻辑上是相同的，不同的只是被排序的数据类型。通过创建一个通用函数，可以定义独立于所有数据的算法特性。一旦定义了通用函数，编译器将针对函数执行时实际使用的数据类型生成正确的代码。从本质上讲，当创建一个通用函数时，实际上是在创建一个能够对自身进行重载的函数。

创建通用函数时要使用关键字template。template（模板）这个词的含义准确地反映出它在C++中的用途。使用它来创建一个描述函数功能的模板（或框架），具体细节留待编译器根据需要去填充。模板函数定义的一般形式如下：

```
template <class Ttype> ret-type func-name(parameter list)
{
   // body of function
}
```

其中，Ttype是函数使用的数据类型的占位名。这个名字可以在函数定义中使用，然而它只是一个占位符，当创建一个特定版本的函数时，编译器将用实际的数据类型自动取代该占位名。虽然传统上在模板声明中使用关键字class指定一般的数据类型，但使用关键字typename也是可以的。

下面的例子创建一个用来交换两个变量值的通用函数。因为交换两个变量值的一般过程独立于变量类型，所以这种情况很适合用通用函数处理。

```
// Function template example.
#include <iostream>
using namespace std;

// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
  X temp;
```

```
  temp = a;
  a = b;
  b = temp;
}

int main()
{
  int i=10, j=20;
  double x=10.1, y=23.3;
  char a='x', b='z';

  cout << "Original i, j: " << i << ' ' << j << '\n';
  cout << "Original x, y: " << x << ' ' << y << '\n';
  cout << "Original a, b: " << a << ' ' << b << '\n';

  swapargs(i, j); // swap integers
  swapargs(x, y); // swap floats
  swapargs(a, b); // swap chars

  cout << "Swapped i, j: " << i << ' ' << j << '\n';
  cout << "Swapped x, y: " << x << ' ' << y << '\n';
  cout << "Swapped a, b: " << a << ' ' << b << '\n';

  return 0;
}
```

让我们仔细看一看这个程序。下面这行代码告诉编译器两件事情：一个模板正在被创建以及一个通用函数正在开始定义：

```
template <class X> void swapargs(X &a, X &b)
```

这里，X是一个用做占位符的通用类型。在template后面声明了函数swapargs( )，该函数把X作为将要交换的值的数据类型。在main( )中，函数swapargs( )分别用三种不同的数据类型调用，这三种数据类型是int，double和char。因为swapargs( )是通用函数，所以编译器自动生成3个swapargs( )版本：一个用来交换整型值，一个用来交换浮点值，还有一个用来交换字符。

这里有一些与模板有关的重要术语。首先，通用函数（即前面有template语句的函数定义）也称为模板函数，这两个术语在本书中交替使用。当编译器生成这种函数的一个特定版本时，我们将其称为创建了一个说明（Specialization），也叫做生成函数。生成一个函数的行为称为实例化。换句话说，通用函数是模板函数的一个具体实例。

由于C++不将行尾识别为语句的结束符，所以通用函数定义的template子句不必与函数名在同一代码行上。下面的例子说明了另一种说明swapargs( )函数的常用方式。

```
template <class X>
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

如果采用这种形式，有一点特别重要，即：在template语句和通用函数定义之间不能有其他语句。例如，下面的代码段将不能编译：

```
// This will not compile.
template <class X>
int i; // this is an error
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

就像注释语句隐含说明的那样，函数定义必须紧接在 template 说明之后。

18.1.1　带有两个通用类型的函数

可以使用逗号（,）分隔的列表在一个 template 语句中定义一个以上的通用数据类型。例如，下面的程序创建了一个具有两个通用数据类型的模板函数：

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

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
  cout << x << ' ' << y << '\n';
}

int main()
{
  myfunc(10, "I like C++");

  myfunc(98.6, 19L);

  return 0;
}
```

在这个例子中，当编译器在main( )中生成myfunc( )的特定实例时，占位类型type1和type2分别被编译器用数据类型 int 和 char * 以及 double 和 long 取代。

注意： 当创建模板函数时，实际上是允许编译器创建多个不同的函数版本，这些版本是处理各种函数调用方式所必需的。

18.1.2　显式重载通用函数

尽管通用函数可以根据需要对自身进行重载，但也可以显式地对其进行重载。这种形式的重载称为显式说明。如果重载了一个通用函数，则重载的函数将覆盖（或者说隐藏）与那个特定版本相对应的通用函数。例如，看一看前面参数交换例子的以下修订版本：

```
// Overriding a template function.
#include <iostream>
```

```
using namespace std;
template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
  cout << "Inside template swapargs.\n";
}

// This overrides the generic version of swapargs() for ints.
void swapargs(int &a, int &b)
{
  int temp;

  temp = a;
  a = b;
  b = temp;
  cout << "Inside swapargs int specialization.\n";
}

int main()
{
  int i=10, j=20;
  double x=10.1, y=23.3;
  char a='x', b='z';

  cout << "Original i, j: " << i << ' ' << j << '\n';
  cout << "Original x, y: " << x << ' ' << y << '\n';
  cout << "Original a, b: " << a << ' ' << b << '\n';

  swapargs(i, j); // calls explicitly overloaded swapargs()
  swapargs(x, y); // calls generic swapargs()
  swapargs(a, b); // calls generic swapargs()

  cout << "Swapped i, j: " << i << ' ' << j << '\n';
  cout << "Swapped x, y: " << x << ' ' << y << '\n';
  cout << "Swapped a, b: " << a << ' ' << b << '\n';

  return 0;
}
```

这个程序的输出如下所示：

```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swapargs int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

就像程序中注释语句说明的那样，当调用swapargs(i, j)时，它调用程序中定义的显式的重

载swapargs( )版本。由于这个通用函数被显式的重载所覆盖，因此编译器不生成swapargs( )函数的通用版本。

最近，C++引入了一种表示显式函数说明的新型语法，这种新方法使用关键字template。例如，利用新的说明语法，前面程序中重载的swapargs( )函数应如下所示：

```
// Use new-style specialization syntax.
template<> void swapargs<int>(int &a, int &b)
{
  int temp;

  temp = a;
  a = b;
  b = temp;
  cout << "Inside swapargs int specialization.\n";
}
```

可以看到，这种新型语法利用template<>结构来表示说明。创建该说明时采用的数据类型被放在函数名后的尖括号中。使用这种语法来特指任何通用函数类型。虽然此时用一种说明语法代替另一种并无优势可言，但是，从长期来讲这种新型语法可能是一个更好的方法。

模板的说明允许你剪裁一种通用函数版本以满足某种特殊需要，例如，或许利用只适用于某种数据类型的高性能。然而，作为一种普遍规则，如果对不同的数据类型采用不同的函数版本，则应该使用重载函数而不是模板。

18.1.3 重载函数模板

除了创建通用函数的显式的重载版本外，还可以对模板说明本身进行重载。要达到这个目的，只需创建该模板的另一种版本，其参数列表与所有其他版本的参数不同，例如：

```
// Overload a function template declaration.
#include <iostream>
using namespace std;

// First version of f() template.
template <class X> void f(X a)
{
  cout << "Inside f(X a)\n";
}

// Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
  cout << "Inside f(X a, Y b)\n";
}

int main()
{
  f(10);     // calls f(X)
  f(10, 20); // calls f(X, Y)

  return 0;
}
```

这里，用于f( )的模板被重载，以接受一个或两个参数。

18.1.4 将标准参数与模板函数一起使用

在模板函数中可以将标准参数和通用类型的参数混合起来使用，这些非通用类型的参数操作起来就像在其他函数中一样，例如：

```
// Using standard parameters in a template function.
#include <iostream>
using namespace std;

const int TABWIDTH = 8;

// Display data at specified tab position.
template<class X> void tabOut(X data, int tab)
{
  for(; tab; tab--)
    for(int i=0; i<TABWIDTH; i++) cout << ' ';

    cout << data << "\n";
}

int main()
{
  tabOut("This is a test", 0);
  tabOut(100, 1);
  tabOut('X', 2);
  tabOut(10/3, 3);

  return 0;
}
```

这个程序的输出如下所示：

```
This is a test
        100
                X
                        3
```

在这个程序中，函数tabOut( )在第二个参数（tab）给出的位置显示第一个参数。因为第一个参数是一个通用类型，所以可以用tabOut( )显示任何数据类型。参数tab是一个标准的按值调用的参数。把通用类型和非通用类型的参数混合起来不会引起任何麻烦，实际上，这种做法既普遍又实用。

18.1.5 通用函数的限制

除了更受限制之外，通用函数与重载函数非常相似。当函数重载时，在每个函数体内可能执行不同的操作，但是通用函数的所有版本必须执行相同的操作，只是数据类型可能不同。现在看一看下面程序中的重载函数，由于这些函数所做的事情不同，所以不能用通用函数代替。

```
#include <iostream>
#include <cmath>
using namespace std;

void myfunc(int i)
```

```
{
  cout << "value is: " << i << "\n";
}

void myfunc(double d)
{
  double intpart;
  double fracpart;

  fracpart = modf(d, &intpart);
  cout << "Fractional part: " << fracpart;
  cout << "\n";
  cout << "Integer part: " << intpart;
}

int main()
{
  myfunc(1);
  myfunc(12.2);

  return 0;
}
```

18.2 应用通用函数

通用函数是C++中最实用的特性之一，可以适用于所有情形。如前所述，只要某个函数定义为一种可归纳的算法，那么就可以将其做成模板函数。一旦这样做，就可以将其用于任何数据类型，而不必对其重新编码。在讨论通用类之前，我们先来看看两个使用通用函数的例子，这两个例子说明了利用C++的这个强大特性是非常容易的。

18.2.1 通用排序

排序正是一种可以设计通用函数的操作类型。在很大程度上，无论被排序的数据是何种类型，其排序算法都是相同的。下面的程序通过创建一个通用的冒泡排序算法说明了这一点。虽然该冒泡排序算法相当简单，但是它的操作非常清晰明了，而且易于理解。函数bubble()可以给任何数组排序，该函数通过一个指向数组中第一个数据元素的指针和数组元素的个数被调用。

```
// A Generic bubble sort.
#include <iostream>
using namespace std;

template <class X> void bubble(
  X *items, // pointer to array to be sorted
  int count) // number of items in array
{
  register int a, b;
  X t;

  for(a=1; a<count; a++)
    for(b=count-1; b>=a; b--)
      if(items[b-1] > items[b]) {
        // exchange elements
        t = items[b-1];
```

```
        items[b-1] = items[b];
        items[b] = t;
      }
  }

  int main()
  {
    int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
    double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};

    int i;

    cout << "Here is unsorted integer array: ";
    for(i=0; i<7; i++)
      cout << iarray[i] << ' ';
    cout << endl;

    cout << "Here is unsorted double array: ";
    for(i=0; i<5; i++)
      cout << darray[i] << ' ';
    cout << endl;

    bubble(iarray, 7);
    bubble(darray, 5);

    cout << "Here is sorted integer array: ";
    for(i=0; i<7; i++)
      cout << iarray[i] << ' ';
    cout << endl;

    cout << "Here is sorted double array: ";
    for(i=0; i<5; i++)
      cout << darray[i] << ' ';
    cout << endl;

  return 0;
  }
```

该程序的输出如下所示：

```
Here is unsorted integer array: 7 5 4 3 9 8 6
Here is unsorted double array: 4.3 2.5 -0.9 100.2 3
Here is sorted integer array: 3 4 5 6 7 8 9
Here is sorted double array: -0.9 2.5 3 4.3 100.2
```

可以看到，该程序创建了两个数组：一个是int型，另一个是double型。然后，程序对每一个数据进行排序。因为bubble( )是一个模板函数，所以它被自动重载以适应两种不同的数据类型。读者或许想尝试一下利用bubble( )对其他的数据类型（包括你所创建的类）进行排序，如果是这样，编译器将针对每一种情况创建相应的函数版本。

18.2.2 压缩数组

另一个从被做成模板而获益的函数是compact( )，该函数用于压缩数组中的元素。我们常常想要在一个数组的中间删掉一些元素，然后再把剩下的元素向下移，从而使所有不用的元素位于数组的一端。因为这种操作独立于被操作的数据类型，所以该操作对所有类型的数组来说

是相同的。在下面的程序中，通用函数compact( )是通过一个指向第一个数组元素的指针、数组中的元素个数和被删除元素的起始下标和终止下标来调用的。这个函数可以删除这些元素并对该数组进行压缩。为便于说明，这里把数组中放置无用元素一端的下标定为0。

```
// A Generic array compaction function.
#include <iostream>
using namespace std;

template <class X> void compact(
  X *items,  // pointer to array to be compacted
  int count, // number of items in array
  int start, // starting index of compacted region
  int end)   // ending index of compacted region
{
  register int i;

  for(i=end+1; i<count; i++, start++)
    items[start] = items[i];

  /* For the sake of illustration, the remainder of
     the array will be zeroed. */
  for( ; start<count; start++) items[start] = (X) 0;
}

int main()
{
  int nums[7] = {0, 1, 2, 3, 4, 5, 6};
  char str[18] = "Generic Functions";

  int i;

  cout << "Here is uncompacted integer array: ";
  for(i=0; i<7; i++)
    cout << nums[i] << ' ';
  cout << endl;

  cout << "Here is uncompacted string: ";
  for(i=0; i<18; i++)
    cout << str[i] << ' ';
  cout << endl;

  compact(nums, 7, 2, 4);
  compact(str, 18, 6, 10);

  cout << "Here is compacted integer array: ";
  for(i=0; i<7; i++)
    cout << nums[i] << ' ';
  cout << endl;

  cout << "Here is compacted string: ";
  for(i=0; i<18; i++)
    cout << str[i] << ' ';
  cout << endl;

  return 0;
}
```

这个程序压缩了两种类型的数组，一种是整型数组，另一种是字符串数组。当然，函数compact( )可以对任何类型的数组进行压缩。该程序的输出如下所示：

```
Here is uncompacted integer array: 0 1 2 3 4 5 6
Here is uncompacted string: G e n e r i c  F u n c t i o n s
Here is compacted integer array: 0 1 5 6 0 0 0
Here is compacted string: G e n e r i c t i o n s
```

如前面的例子所示，一旦按照模板的观点思考问题，就会自然而然地想到它有很多用途。只要函数的基本逻辑独立于数据，就可以使其成为通用函数。

18.3 通用类

除了通用函数之外，还可以定义通用类。若要定义通用类，需要创建一个定义该类使用的所有算法的类。然而，当创建该类的对象时，真正被操作的数据类型将作为参数指定。

当一个类采用可被归纳的逻辑时，通用类是很有用的。例如，维护一个整数队列的算法同样适用于字符型队列，维护邮件地址链表的算法也同样适用于汽车零件信息的链表。当创建一个通用类时，该类可以执行你所定义的适用于所有数据类型的操作，例如维护队列或链表。编译器将根据创建对象时指定的数据类型自动生成相应的对象类型。

通用类声明的语法形式如下所示：

```
template <class Ttype> class class-name {
  .
  .
  .
}
```

其中，Ttype是占位符类型名，它在类实例化时指定。必要时，可以使用一个用逗号分隔的列表来定义多个通用数据类型。

一旦创建了一个通用类，就可以按照以下语法形式创建该类的特定实例了：

```
class-name <type> ob;
```

其中，type是该类将要操作的数据类型名。通用类的成员函数本身自动成为通用的，不需要用template显式地指定。

在下面的程序中，stack类（最早在第11章中引入）被修改为通用类，因此可以用于存储任何类型的对象。在这个例子中，虽然创建了一个字符堆栈和一个浮点堆栈，但也可以采用任何数据类型。

```
// This function demonstrates a generic stack.
#include <iostream>
using namespace std;

const int SIZE = 10;

// Create a generic stack class
template <class StackType> class stack {
  StackType stck[ SIZE] ; // holds the stack
  int tos; // index of top-of-stack
```

```
public:
  stack() { tos = 0; } // initialize stack
  void push(StackType ob); // push object on stack
  StackType pop(); // pop object from stack
};

// Push an object.
template <class StackType> void stack<StackType>::push(StackType ob)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = ob;
  tos++;
}
// Pop an object.
template <class StackType> StackType stack<StackType>::pop()
{
  if(tos==0) {
    cout << "Stack is empty.\n";
    return 0; // return null on empty stack
  }
  tos--;
  return stck[tos];
}

int main()
{
  // Demonstrate character stacks.
  stack<char> s1, s2; // create two character stacks
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

  // demonstrate double stacks
  stack<double> ds1, ds2; // create two double stacks
  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";
```

```
  return 0;
}
```

从中可以看出，通用类的声明与通用函数的声明类似，堆栈中存储的数据类型是类声明中的通用类型。直到声明了一个堆栈对象，真正的数据类型才被确定。当声明了一个特定的stack实例后，编译器自动生成处理实际数据所需的所有函数和变量。在这个例子中，我们声明了两个不同的堆栈类型：两个是int型的堆栈，两个是double型的堆栈。下面我们仔细看一看这些声明：

```
stack<char> s1, s2; // create two character stacks
stack<double> ds1, ds2; // create two double stacks
```

注意所需要的数据类型是如何在尖括号内传递的。通过改变创建stack对象时指定的数据类型，可以改变相应堆栈中存储的数据类型。例如，利用下面的声明，可以创建另一个存储字符指针的堆栈：

```
stack<char *> chrptrQ;
```

也可以创建能够存储你所创建的数据类型的堆栈，例如，如果想用下面的结构来存储地址信息，

```
struct addr {
  char name[ 40] ;
  char street[ 40] ;
  char city[ 30] ;
  char state[ 3] ;
  char zip[ 12] ;
} ;
```

那么可以利用stack生成一个存储addr类型对象的堆栈。声明如下：

```
stack<addr> obj;
```

就像stack类说明的那样，因为通用函数和通用类使你能够定义对象的一般形式，这种形式的对象可以和任何数据类型一起使用，所以它们是非常有效的工具，可以最大限度地提高编程效率。这样就可以把你从为每个使用该算法的数据类型创建各个单独的实现中解放出来。编译器将自动创建通用类的各个特定版本。

18.3.1 具有两个通用数据类型的范例

模板类可以含有一个以上的通用数据类型，此时只需在template说明中声明该类所需的所有数据类型，各个数据类型之间用逗号分开。例如，下面这个简单的例子创建了一个使用两个通用数据类型的类：

```
/* This example uses two generic data types in a
   class definition.
*/
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
  Type1 i;
```

```
  Type2 j;
public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('X', "Templates add power.");
  ob1.show(); // show int, double
  ob2.show(); // show char, char *

return 0;
}
```

这个程序的输出如下所示：

```
10 0.23
X Templates add power.
```

该程序声明了两种对象类型：ob1使用int和double型的数据，ob2使用字符数据和字符指针数据。两种情况下，编译器都将自动生成相应的数据和函数以适应创建对象的具体情形。

18.3.2 模板类的应用——通用数组类

为了说明模板类的实际功用，我们来看一看一种普遍使用模板类的情形。正如第15章介绍的那样，可以重载[]操作符，这样做会允许你创建自己的数组实现，包括提供运行时边界检查的“安全数组”。我们知道，用C++编写的程序在运行时有可能出现数组越界而又不给出运行错误信息的情况。然而，如果创建了一个含有数组的类，并且只允许通过重载[]下标运算符来访问该数组，那么就可以截获越界的数组下标。

通过把运算符重载和模板类结合起来使用，就可以创建一个通用的安全数组类型，可以用于创建任何数据类型的安全数组。下面的程序展示了这种数组类型：

```
// A generic safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 10;

template <class AType> class atype {
  AType a[SIZE];
public:
  atype() {
    register int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType> AType &atype<AType>::operator[](int i)
```

```
{
  if(i<0 || i> SIZE-1) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype<int> intob; // integer array
  atype<double> doubleob; // double array

  int i;

  cout << "Integer array: ";
  for(i=0; i<SIZE; i++) intob[i] = i;
  for(i=0; i<SIZE; i++) cout << intob[i] << " ";
  cout << '\n';

  cout << "Double array: ";
  for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
  for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
  cout << '\n';

  intob[12] = 100; // generates runtime error

  return 0;
}
```

上面这个程序实现了一个通用的安全数组类型，并通过创建一个int型数和一个double型数组说明了它的用途。读者应该尝试一下创建其他类型的数组。就像这个例子所示的，通用类的功效之一是可以使编程人员只编写一次代码并进行调试，之后就可以将其应用于任何数据类型而不必对每一个具体的应用进行重新编码。

18.3.3 对通用类使用无类型参数

在通用类的模板说明中，还可以指定无类型参数。即在一个模板说明中，可以指定一个常用的标准参数类型（例如整型或指针型），所采用的语法在本质上与普通函数的参数相同，即只包含参数类型和参数名。例如，下面的程序为实现前面程序段中的安全数组类提供了一种更好的方法，它使你能够指定数组的大小：

```
// Demonstrate non-type template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;

// Here, int size is a non-type argument.
template <class AType, int size> class atype {
  AType a[size]; // length of array is passed in size
public:
  atype() {
    register int i;
```

```
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype<int, 10> intob;       // integer array of size 10
  atype<double, 15> doubleob; // double array of size 15

  int i;

  cout << "Integer array: ";
  for(i=0; i<10; i++) intob[i] = i;
  for(i=0; i<10; i++) cout << intob[i] << " ";
  cout << '\n';

  cout << "Double array: ";
  for(i=0; i<15; i++) doubleob[i] = (double) i/3;
  for(i=0; i<15; i++) cout << doubleob[i] << " ";
  cout << '\n';

  intob[12] = 100; // generates runtime error

  return 0;
}
```

现在我们仔细看一看atype的模板说明。注意，size被声明为int型，然后这个参数用在atype中声明数组a的大小。即使size在程序中被描述为一个“变量”，但它的值在编译时就已经知道了，这就使其可以用来设定数组大小。此外，size还可以用在函数operator[ ]( )中检查边界的范围。我们可以注意到在main( )中整型数组和浮点型数组是如何创建的。第二个参数指定每个数组的大小。

无类型参数只限于整型、指针型和引用，其他类型（例如浮点型float）则不能使用。传递给无类型参数的变元要么由一个整型常量、要么由一个指向全局函数或对象的指针或引用组成。由于无类型参数的值不能改变，所以无类型参数本身应该被看做常量。例如，在operator[ ]( )内不允许出现如下语句：

```
size = 10; // Error
```

因为无类型参数被当做常量，所以可以用来设定数组大小，因此具有很大的实用功效。就像关于安全数组的例子中说明的那样，利用无类型参数可以有效地扩展模板类的功能。

尽管无类型参数中所包含的信息必须在编译时获知，但这种限制与它的功效相比是微不足道的。

18.3.4 在模板类中使用默认参数

模板类可以包含与通用类型相关的默认参数。例如：

```
template <class X=int> class myclass { //...
```

这里，当myclass的对象被实例化时，如果没有指定其他的数据类型，则将使用类型int。

无类型参数也可以使用默认参数，当类被实例化时，如果没有显式地指定它的值，则使用默认值。指定无类型参数的默认参数的语法与指定函数参数的默认参数的语法相同。

下面是安全数组类的另一个版本，该版本使用了数据类型和数组大小的默认参数。

```
// Demonstrate default template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;

// Here, AType defaults to int and size defaults to 10.
template <class AType=int, int size=10> class atype {
  AType a[size]; // size of array is passed in size
public:
  atype() {
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype<int, 100> intarray;  // integer array, size 100
  atype<double> doublearray; // double array, default size
  atype<> defarray;          // default to int array of size 10

  int i;

  cout << "int array: ";
  for(i=0; i<100; i++) intarray[i] = i;
  for(i=0; i<100; i++) cout << intarray[i] << " ";
  cout << '\n';

  cout << "double array: ";
```

```
  for(i=0; i<10; i++) doublearray[i] = (double) i/3;
  for(i=0; i<10; i++) cout << doublearray[i] << " ";
  cout << '\n';

  cout << "defarray array: ";
  for(i=0; i<10; i++) defarray[i] = i;
  for(i=0; i<10; i++) cout << defarray[i] << " ";
  cout << '\n';

  return 0;
}
```

我们仔细看一看下面这行代码：

```
template <class AType=int, int size=10> class atype {
```

这里，AType默认为int，size默认为10。就像程序中说明的那样，atype对象可以用3种方式创建：

- 显式地指定数组类型和数组大小
- 显示地指定数组类型，但将数组大小默认为10
- 将数组类型默认为int并将数组大小默认为10

默认参数（特别是默认类型）的使用使模板类具备了多功能性。虽然可以让使用你定义的类的用户根据需要指定相应的数据类型，但也可以将最常用的数据类型定义为默认类型。

18.3.5 显式的类说明

像使用模板函数一样，可以创建通用类的显式说明。要做到这一点，使用template<>结构，它的工作原理与创建显式函数说明一样。例如：

```
// Demonstrate class specialization.
#include <iostream>
using namespace std;

template <class T> class myclass {
  T x;
public:
  myclass(T a) {
    cout << "Inside generic myclass\n";
    x = a;
  }
  T getx() { return x; }
};

// Explicit specialization for int.
template <> class myclass<int> {
  int x;
public:
  myclass(int a) {
    cout << "Inside myclass<int> specialization\n";
    x = a * a;
  }
```

```
  int getx() { return x; }
};

int main()
{
  myclass<double> d(10.1);
  cout << "double: " << d.getx() << "\n\n";

  myclass<int> i(5);
  cout << "int: " << i.getx() << "\n";

  return 0;
}
```

这个程序的输出如下所示：

```
Inside generic myclass
double: 10.1

Inside myclass<int> specialization
int: 25
```

我们仔细看一看程序中的下面这行代码：

```
template <> class myclass<int> {
```

这行代码告诉编译器正在创建一个myclass的显式整型说明。该语法可用于任何类型的类说明。

因为显式类说明在允许编译器自动处理所有其他情况的同时能够允许你很容易地处理一个或两个特例，因此扩展了通用类的功能。当然，如果发现创建的类说明过多，那么或许开始时就不使用模板类会更好。

18.4 关键字typename和export

最近，C++添加了两个与模板有关的关键字：typename和export，它们在C++中各有其特殊的作用。下面就简单介绍一下这两个关键字。

关键字typename有两个用途。第一个用途就像本书前面讲到的那样，可以代替模板声明中的关键字class。例如，模板函数swapargs( )可以被指定如下：

```
template <typename X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

其中，typename指定了通用类型X。在这个上下文中使用class和使用typename之间没有区别。

typename的第二个用途是通知编译器模板声明中使用的名字是一个类型名，而不是一个对象名。例如，下面的代码可以确保X::Name被当作一个类型名处理。

```
typename X::Name someObject;
```

关键字 export 在 template 声明之前，它可以使其他文件只通过指定模板声明而不是复制全部定义就能够使用在不同文件中定义的模板。

18.5　模板的功用

模板有助于创建可重用的代码。通过使用模板类，可以创建一些在各种编程中一再使用的框架。例如，考虑一下 stack 类。stack 类最早出现在第 11 章，它只能用来存储整型值。虽然基础算法适用于任何数据类型，但该类中数据类型的硬编码严格限制了它的应用。然而，通过把 stack 定义为通用类，就可以创建适用于任何数据类型的堆栈。

通用函数和通用类提供了一个强大的工具，可以用它来提高编程效率。一旦编写并调试了一个模板类，就有了一个可用于各种不同情形的固化软件组件，它可把你从创建各个单独的实现中解放出来，这些实现针对的是该类处理的各种数据类型。

虽然模板语法开始时看起来有些差强人意，但是它所带来的回报很值得让我们花些时间来适应。模板函数和类在编程时已经成为很平常的事了，而且这种趋势还在继续。例如，C++ 定义的标准模板库（Standard Template Library，STL）就是建立在模板基础之上的。最后，虽然模板在 C++ 中添加了一个抽象层，但最终仍然编译为同样的高性能对象代码。