

# 容器类和模板类

容器类是用来存放多个其它类型的对象的类。`string` 类就是一个容器类，它是用来存放 `char` 类型的变量的。对于其它类型，如 `double`、`string` 或新的类型，如 `Student`，我们也可以定义它们的容器类。通常一个容器类只可以存储一个类型的多个对象。模板容器类是可以存放不同类型的对象的类。定义了一个模板容器类，相当于定义了多个容器类，它们分别可以存放不同类型的对象。

## 1. 容器类 `Vector`

容器类 `container` 是用来存放其它类型的对象的类型。下面我们介绍一个存放 `double` 类型变量的容器类。这个容器类封装了一个数组，这个封装提供了一种可变长度的数组，同时免除了用户动态分配内存空间的麻烦。我们先看看它的使用法，然后再讲解它的实现。

```
1 int main() {  
2     Vector vector1;  
3     for (int i = 0; i < 10; ++ i) {  
4         vector1.push_back(i + 0.5);  
5     }  
6     for (int i = 0; i < vector1.size(); ++ i) {  
7         cout << vector1[i] << endl;  
8     }  
9 }
```

在第二行，我们定义了一个容器类 `Vector` 的对象 `vector1`。

这个时候，容器类对象 `vector1` 存放的 `double` 变量的个数为 0。然后，我们调用对象函数 `push_back`，它向容器类对象 `vector1` 添加一个 `double` 变量。容器类对象 `vector1` 中的 `double` 变量是按顺序存放的，每次调用 `push_back`，容器的大小（所存放 `double` 变量的个数）便增大 1，且新增的 `double` 变量被放在最后。

当执行到第 6 行之前，容器类对象 `vector1` 中已经有 10 个 `double` 变量，对象函数 `vector1.size()` 将返回 10。最后，在第 7 行我们看到，容器类 `Vector` 还提供对象操作符函数 `operator []`，通过它可以获得容器中各个 `double` 变量的引用。

容器类 `Vector` 的实现细节与类 `string` 相似。在这个 `Vector` 类中的 `elements` 是存放一个 `double` 数组的地址变量，所存放的 `double` 数组是运行时动态分配的。`capacity` 是所分配的 `double` 数组的容量（大小）。在我们将要讲解的这个容器类 `Vector` 中，我们不浪费存储空间（这不一定是好事），所以实际上存放 `double` 变量的个数就是 `capacity`。看下面的程序，如果没有问题，可以跳过这个小节余下的讲解内容。

```
1 class Vector
2 {
3 private:
4     double * elements;
5     int capacity;
6
7     void copy0(double to[], const double from[],
8               int num) const {
```

```

9         for (int i = 0; i < num; ++ i) {
10             to[i] = from[i];
11         }
12     }
13
14     void ensureCapacity0(int capacity2) {
15         double * temp = new double[capacity2];
16         copy0(temp, elements, capacity);
17         delete [] elements;
18         elements = temp;
19         capacity = capacity2;
20     }
21
22     void assign0(const Vector & vector2) {
23         if (elements != 0) {
24             delete [] elements;
25         }
26         capacity = vector2.capacity;
27         elements = new double[capacity];
28         copy0(elements, vector2.elements, capacity);
29     }
30
31 public:
32     Vector() {
33         // have to waste one element when empty
34         elements = new double[1];
35         capacity = 0;
36     }
37
38     ~Vector() {
39         delete [] elements;
40     }
41
42     Vector(const Vector & vector2) {
43         elements = 0;
44         assign0(vector2);
45     }
46
47     Vector & operator = (const Vector & vector2) {
48         assign0(vector2);
49     }
50
51     int size() const {
52         return capacity;

```

```

53     }
54
55     void push_back(double elem) {
56         ensureCapacity0(capacity + 1);
57         elements[capacity - 1] = elem;
58     }
59
60     double & operator [] (int index) const {
61         return elements[index];
62     }
63
64 };

```

因为对象中包含地址且存放的是为每个对象动态分配的内存空间，所以这个类中必须定义析构函数，拷贝构造函数，等于号操作符，它们位于 38-49 行。在 32-36 行定义了一个默认构造函数，它构造一个空的容器。第 34 行本来只需要分配 0 个长度的数组，但是 `new` 操作符申请的内存空间的最小长度为 1。

函数 `ensureCapacity0` 在容器类对象所封装的数组容量不够大的时候增加它的容量，并用函数 `copy0` 恢复来的内容。因为在这个容器类中，实际上存放 `double` 变量的个数就是 `capacity`，所以，在 52 行中返回 `capacity`，在 56 行中添加元素后的大小为 `capacity + 1`。因为在函数 `ensureCapacity0` 中 `capacity` 被赋值为新的大小，所以在 57 行中被添加在最后的元素的索引为 `capacity - 1`。

## 2. 模板容器类

以上所定义的容器类 `Vector` 只能用来存储 `double` 类型的变量。如果我们需要定义一个，容器类 `Vector` 用来存储 `string` 类型的对象，那么我们只需要把中的各个 `double` 替换成 `string` 即可。但是，这产生了几乎重复的代码。

我们可不可以定义一个单一的容器类 `Vector`，它能存放 `double` 类型的变量，或者 `string` 类型的对象，或者任意类型的对象呢？也就是说，我们可不可以定义一个容器类 `Vector`，它的元素可以是未知的类型呢？在 C++ 中，我们可以定义带有未知类型参数的类，这种类叫做模板类。模板类的最主要用途就是模板容器类。我们即将定义一个模板容器类，它的元素的类型为该模板类的类型参数。

在讲述模板容器类版本的 `Vector` 之前，我们先看看如何使用这个模板容器类。

```
1 class Student
2 {
3 public:
4     int id;
5     string name;
6 };
7
8 ostream & operator << (ostream & out,
9 const Student & student) {
10     out << "id=" << student.id <<
11     ", name=" << student.name;
12     return out;
13 }
14
15 istream & operator >> (istream & in,
16 Student & student) {
```

```

17     in >> student.id >> student.name;
18     return in;
19 }
20
21 int main() {
22     Vector<Student> vector1;
23     for (int i = 0; i < 3; ++ i) {
24         Student s;
25         cin >> s;
26         vector1.push_back(s);
27     }
28     for (int i = 0; i < vector1.size(); ++ i) {
29         cout << vector1[i] << endl;
30     }
31 }

```

在第 1-19 行，我们定义了一个 **Student** 类，以及它的输入和输出操作符。在第 22 行，我们定义了一个容器对象 **vector1**，它的类型是 **Vector<Student>**，其中 **Vector** 是模板容器类的名字，类型参数 **Student** 表明容器的元素是 **Student** 类型的对象。

下面我们看模板容器类的实现。实际上我们只需要在非模板类的 **Vector** 中作少许的改动。首先，每个模板类需要说明类型参数，就是在类的定义前面加上以下的一行。

```

1 template <typename T>
2 class Vector
3 {
4     private:
5         T * elements;
6         int capacity;

```

其中，**template** 和 **typename** 是保留字，**T** 是类型参数的名字。类型参数的名字习惯上是单个的大写字母。其次，把非

模板类的 `Vector` 中的 `double` 都替换为 `T` 就可以了。因为 `T` 可能是对象，为了提高程序运行的效率，在函数参数表中我们使用 `T &`，具体改动如下。

```
8      void copy0(T to[], const T from[], int num)
9      const {

15     void ensureCapacity0(int capacity2) {
16         T * temp = new T[capacity2];

23     void assign0(const Vector & vector2) {
28         elements = new T[capacity];

33     Vector() {
35         elements = new T[1];

56     void push_back(const T & elem) {

61     T & operator [] (int index) const {
```

### 3. C++的模板类

与 Java 等较新的语言不同，C++的模板类比较简单。实际上，所定义模板类并不直接产生代码。编译器在看到所需要的实际类型（如 `Vector<double>`、`Vector<Student>`）时才生成这些类。因此，在可执行程序中，实际上可能有多份几乎相同的类的代码。

另外，对于模板类，有些编译器支持不是很好。例如，有些能在普通类中使用的语法在模板类中不能使用。而且，如果模板类中有错，错误信息一般显示得非常复杂，甚至无法理解。因此，写模板类前一般先实现一个对应的一般类（如

用 `double` 代替掉类型参数 `T`），等这个一般类编译通过后再改为模板类（用类型参数 `T` 代替 `double`）。

## 习题

### 1. 用自己的话解释下面的概念

容器类

模板类

类型参数

模板容器类

### 2. 把 `Matrix` 类改为模板容器类

在之前的作业中，`Matrix` 中的元素是 `double` 类型的变量，请实现一个 `Matrix` 模板容器类，它的元素是类型参数。

主程序：

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "constructor 1" << endl;
6     Matrix<double> matrix1(3, 3);
7     matrix1.print();
8
9     const double values2[] = {
10         1, 2, 3,
11         4, 5, 6,
12         7, 8, 9,
13     };
14     cout << "constructor 2" << endl;
15     Matrix<double> matrix2(3, 3, values2);
16     matrix2.print();
```



```

17
18     cout << "copy constructor" << endl;
19     Matrix<double> matrix3 = matrix2;
20     printMatrix(matrix3);
21
22     cout << "operator =" << endl;
23     matrix3.get(1, 1) = 10.0;
24     matrix3 = matrix2;
25     matrix3.print();
26
27     cout << "getColumn" << endl;
28     matrix2.getColumn(2).print();
29     cout << "getRow" << endl;
30     matrix2.getRow(2).print();
31
32     cout << "concatenateRows" << endl;
33     matrix1.concatenateRows(matrix2).print();
34     cout << "concatenateColumns" << endl;
35     matrix1.concatenateColumns(matrix2).print();
36
37     cout << "reshape" << endl;
38     matrix1.concatenateColumns(matrix2).
39         reshape(6, 3).print();
40
41     cout << "transpose" << endl;
42     matrix2.transpose().print();
43
44     cout << "operator +" << endl;
45     (matrix2 + matrix2).print();
46     cout << "operator +" << endl;
47     (matrix2 + 10).print();
48     cout << "operator -" << endl;
49     (matrix2.transpose() - matrix2).print();
50     cout << "operator -" << endl;
51     (matrix2 - 10).print();
52
53     cout << "operator *" << endl;
54     (matrix2.transpose() * matrix2).print();
55     cout << "operator *" << endl;
56     (matrix2 * 2).print();
57
58     cout << "max" << endl;
59     cout << matrix2.max().max().get(1, 1) << endl;
60     cout << "min" << endl;

```

```
61     cout << matrix2.min().min().get(1, 1) << endl;
62     cout << "sum" << endl;
63     cout << matrix2.sum().sum().get(1, 1) << endl;
64 }
```