第 5 章　模板与标准模板库

C++模板允许将数据类型当作模板参数，使得相同的程序代码可以应用于不同的数据类型，进一步提高了程序代码的复用率。C++模板主要包括函数模板与类模板。函数模板或类模板只能在程序的全局范围以及命名空间或类范围内定义，不能在函数内部或语句块内部定义。C++标准程序库提供了标准模板库（Standard Template Library，STL）。标准模板库主要由容器（Container）、迭代器（Iterator）和算法（Algorithm）三部分组成，其中容器是 C++静态数组和动态数组的有益补充，迭代器则为处理容器内部数据提供了一种统一的广义的指针模式，算法部分包括比较、交换、查找、遍历操作、复制、修改、移除、反转和排序等常用算法。迭代器支撑容器和算法，并在算法和容器之间起到桥梁的作用。

5.1　自定义函数模板

下面给出一种自定义函数模板的格式：

```
template <模板参数列表>
函数定义
```

其中，模板参数列表由一系列类型参数组成。每个类型参数的定义格式为：

```
typename 标识符
```

其中，关键字 typename 可以替换为 class。相邻的类型参数之间用逗号分隔。在函数定义部分可以使用上面的标识符作为数据类型。除此之外，函数定义部分的代码编写方式基本上与普通函数定义相同。这些类型参数也称为模板的形式参数，简称为模板的形参。
函数模板不是函数。函数模板只是定义了函数的代码框架。将实际的类型参数代入函数模板从而生成函数的过程称为函数模板实例化，通过这种方式生成的函数称为模板函数。实际的类型参数称为模板的实参。函数模板的实例化以及相应的模板函数调用格式如下：

```
函数模板名称 <实际类型参数列表>  ( 模板函数的实参列表)
```

如果根据模板函数的实际参数列表，编译器就可以推导出函数模板的实际类型参数列表，那么可以省略函数模板的实际类型参数列表。这时，上面的格式简化为：

```
函数模板名称( 模板函数的实参列表)
```

// 示例 1：　模板实参为基本数据类型	//示例 2：　模板实参为类	行号
#include <iostream> using namespace std;	#include <iostream> using namespace std;	// 1 // 2

```
template<typename T>
T gt_sum(T x, T y)
{
    T sum = x + y;
    cout << sum << endl;
    return sum;
} // 函数模板 gt_sum 结束

int main()
{
    gt_sum(1, 2);
    gt_sum(1.1, 2.2);
    // gt_sum(10, 5.5);
    gt_sum<double>(10, 5.5);
    return 0;
} // main 函数结束
```

```
                                    // 3
class CP_A                          // 4
{                                   // 5
public:                             // 6
    int m_a;                        // 7
    CP_A(int i = 0) : m_a(i) {}     // 8
    void mb_out(){cout<<"A: ";}     // 9
}; //类 CP_A 定义结束                // 10
                                    // 11
template<typename T>                // 12
void gt_show(T &a)                  // 13
{                                   // 14
    a.mb_out();                     // 15
    cout << a.m_a << endl;          // 16
} // 函数模板 gt_show 结束          // 17
                                    // 18
int main()                          // 19
{                                   // 20
    CP_A a(10);                     // 21
    gt_show<CP_A>(a);               // 22
    gt_show(a);                     // 23
    return 0;                       // 24
} // main 函数结束                  // 25
```

可以对上面的代码分别进行编译、链接和运行。运行结果示例如下。

示例 1 运行结果	示例 2 运行结果
3 3.3 15.5	A: 10 A: 10

在示例 1 第 4～10 行定义的函数模板 gt_sum 中，模板的形参是 T，函数的形参是 x 和 y。对于第 14 行“gt_sum(1, 2);”，根据函数的实参 1 和 2 可以推断出模板的实参是 int。因此，第 14 行代码也可以写成为“gt_sum<int>(1, 2);”。对于第 15 行“gt_sum(1.1, 2.2);”，根据函数的实参 1.1 和 2.2 可以推断出模板的实参是 double。因此，第 15 行代码也可以写成为“gt_sum<double>(1.1, 2.2);”。对于第 17 行“gt_sum<double>(10, 5.5);”，因为函数实参 10 的数据类型是 int，函数实参 5.5 的数据类型是 double，所以无法从函数实参推断出模板实参。因此，第 17 行代码不可以写成为“gt_sum(10, 5.5);”；否则，会产生如下编译错误：

```
error C2782: “T gt_sum(T,T)”: 模板 参数“T”不明确。
```

在函数模板实例化之后，即在编译器将模板实参代入模板形参形成模板函数之后，才能最终确定函数模板及其实例化结果是否符合语法要求。模板函数“gt_sum<int>”和“gt_sum<double>”在实例化之后的示意性代码分别如下：

// gt_sum<int>	// gt_sum<double>	行号
int gt_sum(int x, int y)	double gt_sum(double x, double y)	// 1
{	{	// 2
int sum = x + y;	double sum = x + y;	// 3
cout << sum << endl;	cout << sum << endl;	// 4
return sum;	return sum;	// 5
}	}	// 6

注意事项：

函数模板实例化的编译过程决定了函数模板的实例化必须与函数模板的定义代码要么在同一个源文件中，要么通过文件包含语句加载到同一个源文件中，否则，编译器无法完成将类型参数代入函数模板的工作。因此，通常将函数模板的完整定义代码放在头文件当中，从而方便实例化。

在示例 2 中，第 12～17 行定义函数模板 gt_show。第 21 和 22 行分别调用函数模板 gt_show，调用的模板实参都是类 CP_A。因为根据函数实参 a 可以推断出模板实参，所以可以省略模板实参类 CP_A，如第 22 行代码所示。因为函数模板 gt_show 用到了模板形参 T 拥有成员变量 m_a 和成员函数 mb_out，所以调用的模板实参类 CP_A 也必须拥有成员变量 m_a 和成员函数 mb_out，否则，无法通过编译。

5.2　自定义类模板

这里首先介绍自定义类模板的一种代码格式：

```
template <模板参数列表>
类定义
```

与函数模板相同，模板参数列表也是由一系列类型参数组成。每个类型参数的定义格式为

```
typename 标识符
```

其中，关键字 typename 可以替换为 class。相邻的类型参数之间用逗号分隔。在类定义中可以使用上面的标识符作为数据类型。除此之外，类定义部分的代码编写方式基本上与普通类定义相同。这些类型参数也称为模板的形式参数，简称为模板的形参。

类模板不是类。类模板只是定义了类的代码框架。将实际的类型参数代入类模板从而生成类的过程称为类模板的实例化。类模板实例化的格式如下：

```
类模板名称 <实际类型参数列表>
```

实际的类型参数称为模板的实参。

// 示例 1：　成员函数在类模板内部实现	//示例 2：　成员函数在类模板外部实现	行号
#include <iostream>	#include <iostream>	// 1

```
using namespace std;

template<typename T>
class CT_A
{
public:
   T m_a;
public:
   CT_A() {}
   CT_A(T a) : m_a(a) {}
   void mb_out(){cout << m_a;}
}; // 模板 CT_A 定义结束

int main()
{
   CT_A<int> a;//不能写: CT_A a;
   a.m_a = 10;
   a.mb_out();

   CT_A<double> b(20.5);
   b.mb_out();
   return 0;
} // main 函数结束
```

```
using namespace std;

template<typename T>
class CT_A
{
public:
   T m_a;
public:
   CT_A() {}
   CT_A(T a) : m_a(a) {}
   void mb_out();
}; // 模板 CT_A 定义结束

template<typename T>
void CT_A<T>::mb_out()
{
   cout << m_a << endl;
} // CT_A 的成员函数 mb_out 结束

int main()
{
   CT_A<int> a;// 不能写: CT_A a;
   a.m_a = 10;
   a.mb_out();

   CT_A<double> b(20.5);
   b.mb_out();
   return 0;
} // main 函数结束
```

```
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
// 10
// 11
// 12
// 13
// 14
// 15
// 16
// 17
// 18
// 19
// 20
// 21
// 22
// 23
// 24
// 25
// 26
// 27
// 28
// 29
// 30
```

可以对上面的代码分别进行编译、链接和运行。运行结果示例如下：

示例 1 运行结果	示例 2 运行结果
1020.5	10 20.5

在**示例 1**中，**类模板 CT_A 的定义与实例化都放在同一个源文件中**，其中定义位于第 4~13 行，实例化位于第 17 行和第 21 行。**在类模板内部实现成员函数**与在类体中实现成员函数基本上相同，如第 10～12 行所示。第 17 行“CT_A<int> a;”将模板实参 int 代入类模板 CT_A 实例化为类“CT_A<int>”，并生成这个类的实例对象 a。第 21 行“CT_A<double> b(20.5);”将模板实参 double 代入类模板 CT_A 实例化为类“CT_A< double >”，并生成这个类的实例对象 b。这体现出了**类模板的优势**，即可以复用代码生成适应不同数据类型的类。

在**示例 2**中，如第 12 行所示，成员函数 mb_out 在类模板 CT_A 内部声明，并在类模板 CT_A 外部实现，如第 15～19 行所示。与类相比，**类模板成员函数在外部实现需要**：

（1）以“template<模板参数列表>”开头，如第 15 行代码“template<typename T>”所示；

（2）在实现成员函数的头部中，在“::成员函数名称”之前必须是“类模板名称<模板参数列表>”，其中在模板参数列表中没有关键字 typename 或 class，如第 16 行代码所示。

注意事项：

类模板实例化的编译过程决定了类模板的实例化和类模板的定义代码要么在同一个源文件中，要么通过文件包含语句加载到同一个源文件中，否则，编译器无法完成将类型参数代入类模板的工作。因此，通常将完整的类模板定义代码放在头文件当中，从而方便类模板的实例化。例如，在编程规范中，有可能会要求将示例 1 第 4～13 行代码和示例 2 第 4～19 行代码放在头文件当中。

// 示例 3： CP_TemplateC.h	//示例 4： CP_TemplateD.h	行号
#ifndef CP_TEMPLATEC_H	#ifndef CP_TEMPLATED_H	// 1
#define CP_TEMPLATEC_H	#define CP_TEMPLATED_H	// 2
		// 3
template<typename T>	template<typename T>	// 4
class CT_B	class CT_B	// 5
{	{	// 6
public：	public：	// 7
T m_b;	T m_b;	// 8
CT_B() ： m_b(0) {}	CT_B() ： m_b(0) {}	// 9
}; // 模板 CT_B 定义结束	}; // 模板 CT_B 定义结束	// 10
		// 11
class CP_C ： public CT_B<int>	template<typename T>	// 12
{	class CP_D ： public CT_B<T>	// 13
public：	{	// 14
CP_C() { m_b = 10; }	public：	// 15
void mb_out(){cout << m_b;}	void mb_out();	// 16
}; // 类 CP_C 定义结束	}; // 类 CP_D 定义结束	// 17
#endif		// 18
	template<typename T>	// 19
	void CP_D<T>::mb_out()	// 20
	{	// 21
	cout << CT_B<T>::m_b << endl;	// 22
	} // CP_D 的成员函数 mb_out 结束	// 23
	#endif	// 24

// 示例 3： CP_TemplateCMain.cpp	//示例 4： CP_TemplateDMain.cpp	行号
#include <iostream>	#include <iostream>	// 1
using namespace std;	using namespace std;	// 2
#include "CP_TemplateC.h"	#include "CP_TemplateD.h"	// 3
		// 4
int main( )	int main( )	// 5
{	{	// 6
CP_C c;	CP_D<int> d;	// 7

```
    c.mb_out();                                    d.mb_out();                                    // 8
    return 0;                                      return 0;                                      // 9
} // main 函数结束                                } // main 函数结束                                // 10
```

可以对上面的代码分别进行编译、链接和运行。运行结果示例如下：

示例 3 运行结果	示例 4 运行结果
10	0

示例 3 由头文件“CP_TemplateC.h”和源文件“CP_TemplateCMain.cpp”组成。**在类模板 CT_B 实例化之后，“CT_B<int>”就是一个类**。因此，**可以派生出子类 CP_C**，如头文件“CP_TemplateC.h”第 12 行代码“class CP_C : public CT_B<int>”所示。子类 CP_C 成员函数 mb_out 可以使用父类 CT_B<int>的成员变量 m_b，如头文件“CP_TemplateC.h”第 16 行代码所示。

示例 4 由头文件“CP_TemplateD.h”和源文件“CP_TemplateDMain.cpp”组成。如头文件“CP_TemplateD.h”第 13 行代码“class CP_D : public CT_B<T>”所示，**类模板 CT_B 是类模板 CP_D 的父类模板**。这实际上也是将**类模板 CT_B 实例化为类 CT_B<T>，从而作为类模板 CP_D 的父类**。因此，不能删除位于第 13 行代码处的“<T>”；否则，将产生编译错误，无法给父类模板 CT_B 指定具体的类型参数。

因为类模板并不是类，子类模板有可能会无法直接看到其父类模板的成员，所以**子类模板在使用父类模板的成员需要加上父类模板实例化的完整路径**，如头文件“CP_TemplateD.h”第 22 行代码“CT_B<T>::m_b”所示。如果将这代码改为：

```
    cout << m_b << endl;                                                    // 22
```

则有可能会出现编译错误“找不到标识符 m_b”。

5.3 向量类模板 vector

向量（vector）是类模板，是容器，可以看作是增强版的数组，由相同数据类型的元素组成，而且允许改变元素个数。使用向量 vector 需要包含头文件<vector>，具体语句如下：

```
#include <vector>
```

5.3.1 向量的构造函数、长度和容量

向量采用**“容量-长度”内存管理机制**，从而减少在改变元素个数时重新分配内存的次数。如图 5-1 所示，在需要给向量分配内存时通常会多分配一些内存。在向量所拥有的内存中可以容纳的元素个数称为**向量的容量**，向量实际在用的元素的个数称为**向量的长度**。向量的长度不大于容量。只有当需要的内存空间超过容量时，才会给向量重新分配内存空间。除非通过向量交换，减小向量的长度通常不会改变向量的容量。

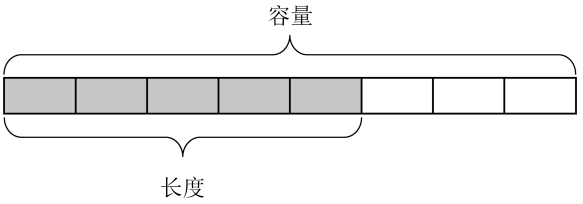

图 5-1　向量的内存示意图

小甜点：

除非调用向量的交换内容成员函数，向量自身的容量通常只会增加，不会减少。

下面介绍向量的 2 个最基本的构造函数。

函数 5　vector<T>::vector

声明：　vector();
说明：　构造容量与长度均为 0 的向量。
参数：　模板参数 T 是元素的数据类型。
头文件：　#include <vector>

函数 6　vector<T>::vector

声明：　vector(size_type n, const T& value = T( ));
说明：　构造容量与长度均为 n 的向量，而且各个元素的值为 value。
参数：　① 模板参数 T 是元素的数据类型。
② 函数参数 n 指定元素个数。
③ 函数参数 value 指定元素的值。
头文件：　#include <vector>

例程 5-1　通过中括号修改向量元素的值。

例程功能描述：创建含有 5 个元素的向量实例对象，其中每个元素的值为 50。通过中括号将其中下标为 1 的元素的值修改为 100，并输出修改前后该元素的值。

例程代码：只包含 1 个源文件“CP_ModifyVectorElementTestMain.cpp”，其内容如下。

```
// 文件名： CP_ModifyVectorElementTestMain.cpp；开发者： 雍俊海                行号
#include <iostream>                                                          // 1
using namespace std;                                                         // 2
#include <vector>                                                            // 3
                                                                             // 4
int main(int argc, char* args[])                                             // 5
{                                                                            // 6
    vector<int> v(5, 50);                                                    // 7
    cout << "修改之前： v[1] = " << v[1] << "。" << endl;                     // 8
    v[1] = 100;                                                              // 9
    cout << "修改之后： v[1] = " << v[1] << "。" << endl;                     // 10
    return 0; // 返回 0 表明程序运行成功                                     // 11
} // main 函数结束                                                           // 12
```

可以对上面的代码进行编译、链接和运行。下面给出一个运行结果示例。

```
修改之前： v[1] = 50。
修改之后： v[1] = 100。
请按任意键继续. . .
```

例程分析：第 7 行代码“vector<int> v(5, 50);”创建了向量 v，它共包含 5 个 int 类型的元素，每个元素的值均为 50。通过第 8~10 行代码可以看出，访问向量元素可以采用与访问数组相同的方式，即通过中括号的方式。元素下标同样也是从 0 开始，并且小于向量的长度。例如，对于本例程中的向量 v，v[0]是向量 v 的第 1 个元素，v[1]是向量 v 的第 2 个元素，v[4]是向量 v 的最后 1 个元素。用来访问向量元素的中括号运算的具体说明如下：

运算符 4 vector<T>::[ ]

声明： reference operator[ ](size_type n);
说明： 返回下标为 n 的元素的引用。
参数： ① 模板参数 T 是元素的数据类型。
② n 是向量元素的下标。这里要求 n 必须满足 0≤n<向量的长度。
返回值： 下标为 n 的元素的引用。
头文件： #include <vector>

📖说明📖：

为了缩短篇幅，本章的部分代码示例只提供代码片断。这些代码片断都可以代替本例程第 7～10 行代码，形成完整的代码，进行编译、链接和运行。下面将不再重复这个说明。

因为向量的中括号运算返回元素的引用，所以通过中括号运算既可以读取元素的值，如例程 5-1 第 8 行代码所示，也可以修改元素的值，如例程 5-1 第 9 行代码所示。

向量的成员函数 at 拥有与向量的中括号运算完全相同的功能，该函数的具体说明如下：

函数 7 vector<T>::at

声明： reference at(size_type n);
说明： 返回下标为 n 的元素的引用。
参数： ① 模板参数 T 是元素的数据类型。
② n 是向量元素的下标。这里要求 n 必须满足 0≤n<向量的长度。
返回值： 下标为 n 的元素的引用。
头文件： #include <vector>

下面给出成员函数 at 的示例代码片断。

```
vector<int> v(3, 5); // 结果： 创建向量 v： v[0]=v[1]=v[2]=5。          // 1
v.at(0) = 10; // 结果： v[0]=10。                                      // 2
int & a = v.at(1); // a=v[1]=5。a 是元素 v[1]的别名。                  // 3
a = 20; // 结果： v[1]的值变为 20。                                    // 4
int b = v.at(2); // b=v[2]=5。变量 b 和元素 v[2]具有不变的内存空间。   // 5
b = 30; // 结果： v[2]的值保持不变，仍然是 5。                         // 6
```

获取向量的第 1 个元素和最后 1 个元素的引用，还可以通过成员函数 **front** 和 **back**：

函数 8　vector<T>::front

声明：　reference front( );
说明：　返回第 1 个元素的引用。调用本成员函数的前提是向量的长度不为 0。
参数：　模板参数 T 是元素的数据类型。
返回值：　第 1 个元素的引用。
头文件：　#include <vector>

函数 9　vector<T>::back

声明：　reference back( );
说明：　返回最后 1 个元素的引用。调用本成员函数的前提是向量的长度不为 0。
参数：　模板参数 T 是元素的数据类型。
返回值：　最后 1 个元素的引用。
头文件：　#include <vector>

下面给出调用向量的成员函数 front 和 back 的示例代码片断。

```
vector<int> v(3, 5); // 结果：创建向量 v：v[0]=v[1]=v[2]=5。          // 1
v.front( ) = 10;     // 结果：v[0]的值变为 10。                      // 2
v.back( ) = 20;      // 结果：v[2]的值变为 20。                      // 3
```

下面介绍获取向量长度与容量的成员函数。

函数 10　vector<T>::size

声明：　size_type size( ) const;
说明：　返回向量的长度。
参数：　模板参数 T 是元素的数据类型。
返回值：　向量的长度。
头文件：　#include <vector>

函数 11　vector<T>::capacity

声明：　size_type capacity() const;
说明：　返回向量的容量。
参数：　模板参数 T 是元素的数据类型。
返回值：　向量的容量。
头文件：　#include <vector>

下面给出向量长度与容量函数的示例代码片断。

```
vector<int> v;                                                        // 1
cout << "长度 = " << v.size() << "。";     // 输出：长度 = 0。        // 2
cout << "容量 = " << v.capacity() << "。"; // 输出：容量 = 0。        // 3
```

下面介绍判断向量长度是否为 **0** 的成员函数。

函数 12　vector<T>::empty

声明：　bool empty() const;

说明： 判断向量的长度是否为 0。
参数： 模板参数 T 是元素的数据类型。
返回值： 如果向量的长度为 0，则返回 true；否则返回 false。
头文件： #include <vector>

下面给出成员函数 empty 的示例代码片断。

```
vector<int> v1;                                                    // 1
vector<int> v2(10, 100);                                           // 2
cout << "向量 v1： " << v1.empty() << "。"; // 输出： 向量 v1： 1。    // 3
cout << "向量 v2： " << v2.empty() << "。"; // 输出： 向量 v2： 0。    // 4
```

在上面代码片断中，因为向量 v1 的长度是 0，所以 v1.empty()返回 true；因为向量 v2 的长度是 10，所以 v2.empty()返回 false。

要注意向量成员函数 size 与 max_size 的区别，成员函数 **max_size** 的具体说明如下：

函数 13 vector<T>::max_size

声明： size_type max_size() const;
说明： 返回向量所允许的最大长度。注意，这不是向量的容量，也不是向量的长度。在申请向量的容量时，不应当超过这个所允许的最大长度。
参数： 模板参数 T 是元素的数据类型。
返回值： 向量所允许的最大长度。
头文件： #include <vector>

下面给出向量所允许的最大长度函数的示例代码片断。

```
vector<int> v1;  //向量 v1 的长度是 0。                                  // 1
vector<double> v2(5, 100); //向量 v2 的长度是 5。                        // 2
cout << "向量 v1： " << v1.max_size() << "。"; // 向量 v1： 1073741823。 // 3
cout << "向量 v2： " << v2.max_size() << "。"; // 向量 v2： 536870911。  // 4
```

从上面代码片断的输出可以看出，向量所允许的最大长度与向量元素的数据类型有关。

向量的拷贝构造函数的具体说明如下：

函数 14 vector<T>::vector

声明： vector(vector<T>& x);
说明： 构造向量，复制向量 x 的内容，包括向量长度以及各个元素的值。新向量的容量不小于长度，但容量的具体数值取决于具体的 C++语言支撑平台。
参数： ① 模板参数 T 是元素的数据类型。
② 函数参数 x 是被复制的向量。
头文件： #include <vector>

下面给出调用向量拷贝构造函数的示例代码片断。

```
vector<int> v1(5, 50); // 结果： v1 的长度和容量均为 5，各元素值为 50。   // 1
v1.resize(3); // 结果： v1 的长度变为 3，容量仍然是 5，各元素值为 50。     // 2
vector<int> v2(v1); // 结果： v2 的长度为 3，各元素值为 50。              // 3
cout << "长度 = " << v2.size() << "。";     // 输出： 长度 = 3。          // 4
```