C++类模板5分钟入门教程

C++ 除了支持函数模板，还支持**类模板（Class Template）**。函数模板中定义的类型参数可以用在函数声明和函数定义中，类模板中定义的类型参数可以用在类声明和类实现中。类模板的目的同样是将数据的类型参数化。

声明类模板的语法为：

```
template<typename 类型参数1 , typename 类型参数2 , ...> class 类名{
    //TODO:
};
```

类模板和函数模板都是以 template 开头（当然也可以使用 class，目前来讲它们没有任何区别），后跟类型参数；类型参数不能为空，多个类型参数用逗号隔开。

一但声明了类模板，就可以将类型参数用于类的成员函数和成员变量了。换句话说，原来使用 int、float、char 等内置类型的地方，都可以用类型参数来代替。

假如我们现在要定义一个类来表示坐标，要求坐标的数据类型可以是整数、小数和字符串，例如：

- x = 10、y = 10
- x = 12.88、y = 129.65
- x = "东经180度"、y = "北纬210度"

这个时候就可以使用类模板，请看下面的代码：

```
01.  template<typename T1, typename T2>  //这里不能有分号
02.  class Point{
03.  public:
04.      Point(T1 x, T2 y): m_x(x), m_y(y){ }
05.  public:
06.      T1 getX() const;  //获取x坐标
07.      void setX(T1 x);  //设置x坐标
08.      T2 getY() const;  //获取y坐标
09.      void setY(T2 y);  //设置y坐标
10.  private:
11.      T1 m_x;  //x坐标
12.      T2 m_y;  //y坐标
13.  };
```

x 坐标和 y 坐标的数据类型不确定，借助类模板可以将数据类型参数化，这样就不必定义多个类了。

> 注意：模板头和类头是一个整体，可以换行，但是中间不能有分号。

上面的代码仅仅是类的声明，我们还需要在类外定义成员函数。在类外定义成员函数时仍然需要带上模板头，格式为：

```
template<typename 类型参数1 , typename 类型参数2 , ...>
返回值类型 类名<类型参数1 , 类型参数2, ...>::函数名(形参列表){
    //TODO:
}
```

第一行是模板头，第二行是函数头，它们可以合并到一行，不过为了让代码格式更加清晰，一般是将它们分成两行。

下面就对 Point 类的成员函数进行定义：

```
01.  template<typename T1, typename T2>  //模板头
02.  T1 Point<T1, T2>::getX() const /*函数头*/ {
03.      return m_x;
04.  }
05.
06.  template<typename T1, typename T2>
07.  void Point<T1, T2>::setX(T1 x){
08.      m_x = x;
09.  }
10.
11.  template<typename T1, typename T2>
12.  T2 Point<T1, T2>::getY() const{
13.      return m_y;
14.  }
15.
16.  template<typename T1, typename T2>
17.  void Point<T1, T2>::setY(T2 y){
18.      m_y = y;
19.  }
```

请读者仔细观察代码，除了 template 关键字后面要指明类型参数，类名 Point 后面也要带上类型参数，只是不加 typename 关键字了。另外需要注意的是，在类外定义成员函数时，template 后面的类型参数要和类声明时的一致。

使用类模板创建对象

上面的两段代码完成了类的定义，接下来就可以使用该类创建对象了。使用类模板创建对象时，需要指明具体的数据类型。请看下面的代码：

```
Point<int, int> p1(10, 20);
Point<int, float> p2(10, 15.5);
Point<float, char*> p3(12.4, "东经180度");
```

与函数模板不同的是，类模板在实例化时必须显式地指明数据类型，编译器不能根据给定的数据推演出数据类型。

除了对象变量，我们也可以使用对象指针的方式来实例化：

```
Point<float, float> *p1 = new Point<float, float>(10.6, 109.3);
Point<char*, char*> *p = new Point<char*, char*>("东经180度", "北纬210度");
```

需要注意的是，赋值号两边都要指明具体的数据类型，且要保持一致。下面的写法是错误的：

```
//赋值号两边的数据类型不一致
Point<float, float> *p = new Point<float, int>(10.6, 109);
//赋值号右边没有指明数据类型
Point<float, float> *p = new Point(10.6, 109);
```

综合示例

【实例1】将上面的类定义和类实例化的代码整合起来，构成一个完整的示例，如下所示：

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

template<class T1, class T2>  //这里不能有分号
class Point{
public:
    Point(T1 x, T2 y): m_x(x), m_y(y){ }
public:
    T1 getX() const;  //获取x坐标
    void setX(T1 x);  //设置x坐标
    T2 getY() const;  //获取y坐标
    void setY(T2 y);  //设置y坐标
private:
    T1 m_x;  //x坐标
    T2 m_y;  //y坐标
};
```

```
17.
18.  template<class T1, class T2>  //模板头
19.  T1 Point<T1, T2>::getX() const /*函数头*/ {
20.      return m_x;
21.  }
22.
23.  template<class T1, class T2>
24.  void Point<T1, T2>::setX(T1 x){
25.      m_x = x;
26.  }
27.
28.  template<class T1, class T2>
29.  T2 Point<T1, T2>::getY() const{
30.      return m_y;
31.  }
32.
33.  template<class T1, class T2>
34.  void Point<T1, T2>::setY(T2 y){
35.      m_y = y;
36.  }
37.
38.  int main(){
39.      Point<int, int> p1(10, 20);
40.      cout<<"x="<<p1.getX()<<", y="<<p1.getY()<<endl;
41.
42.      Point<int, char*> p2(10, "东经180度");
43.      cout<<"x="<<p2.getX()<<", y="<<p2.getY()<<endl;
44.
45.      Point<char*, char*> *p3 = new Point<char*, char*>("东经180度", "北纬210度");
46.      cout<<"x="<<p3->getX()<<", y="<<p3->getY()<<endl;
47.
48.      return 0;
49.  }
```

运行结果：
x=10, y=20
x=10, y=东经180度
x=东经180度, y=北纬210度

在定义类型参数时我们使用了 class，而不是 typename，这样做的目的是让读者对两种写法都熟悉。

【实例2】用类模板实现可变长数组。

```
01. #include <iostream>
02. #include <cstring>
03. using namespace std;
04. template <class T>
05. class CArray
06. {
07.     int size; //数组元素的个数
08.     T *ptr; //指向动态分配的数组
09. public:
10.     CArray(int s = 0);  //s代表数组元素的个数
11.     CArray(CArray & a);
12.     ~CArray();
13.     void push_back(const T & v); //用于在数组尾部添加一个元素v
14.     CArray & operator=(const CArray & a); //用于数组对象间的赋值
15.     T length() { return size; }
16.     T & operator[](int i)
17.     {//用以支持根据下标访问数组元素，如a[i] = 4;和n = a[i]这样的语句
18.         return ptr[i];
19.     }
20. };
21. template<class T>
22. CArray<T>::CArray(int s):size(s)
23. {
24.      if(s == 0)
25.          ptr = NULL;
26.     else
27.         ptr = new T[s];
28. }
29. template<class T>
30. CArray<T>::CArray(CArray & a)
31. {
32.     if(!a.ptr) {
33.         ptr = NULL;
34.         size = 0;
35.         return;
36.     }
37.     ptr = new T[a.size];
38.     memcpy(ptr, a.ptr, sizeof(T ) * a.size);
39.     size = a.size;
40. }
41. template <class T>
42. CArray<T>::~CArray()
43. {
44.      if(ptr) delete [] ptr;
```

```
45. }
46. template <class T>
47. CArray<T> & CArray<T>::operator=(const CArray & a)
48. { //赋值号的作用是使"="左边对象里存放的数组，大小和内容都和右边的对象一样
49.     if(this == & a) //防止a=a这样的赋值导致出错
50.     return * this;
51.     if(a.ptr == NULL) {  //如果a里面的数组是空的
52.         if( ptr )
53.             delete [] ptr;
54.         ptr = NULL;
55.         size = 0;
56.         return * this;
57.     }
58.      if(size < a.size) { //如果原有空间够大，就不用分配新的空间
59.          if(ptr)
60.             delete [] ptr;
61.         ptr = new T[a.size];
62.     }
63.     memcpy(ptr,a.ptr,sizeof(T)*a.size);
64.     size = a.size;
65.      return *this;
66. }
67. template <class T>
68. void CArray<T>::push_back(const T & v)
69. {  //在数组尾部添加一个元素
70.     if(ptr) {
71.         T *tmpPtr = new T[size+1]; //重新分配空间
72.     memcpy(tmpPtr,ptr,sizeof(T)*size); //拷贝原数组内容
73.     delete []ptr;
74.     ptr = tmpPtr;
75. }
76.     else  //数组本来是空的
77.         ptr = new T[1];
78.     ptr[size++] = v; //加入新的数组元素
79. }
80. int main()
81. {
82.     CArray<int> a;
83.     for(int i = 0;i < 5;++i)
84.         a.push_back(i);
85.     for(int i = 0; i < a.length(); ++i)
86.         cout << a[i] << " ";
87.     return 0;
88. }
```