模板与泛型编程

龚成 南开大学软件学院

cheng-gong@nankai.edu.cn

函数模板

函数模板概念

函数模板说明

类型与非类型模板参数

函数模板使用

函数模板重载

函数模板概念

- 通常我们设计的算法是可以处理多种数据类型的。但目前来说，处理这样的问题，即使设计为重载函数也只是使用相同的函数名，函数体仍然要分别定义。
- 例如对两个 int 型数据、double 型数据、char 型数据等，求出其中最大值的函数：

```
int max (int a, int b) {
    //函数 max，求两个 int 型数据的最大值
    if(a>b) return a;
    else return b;
}
double max (double a, double b) {
    //重载函数 max，求两个 double 型数据的最大值
    if(a>b) return a;
    else return b;
}
```

```
char max (char a, char b) {
    //重载函数 max，求两个 char 型数据的最大值
    if(a>b) return a;
    else return b;
}
```

函数模板概念

- 若“提取”出一个可变化的类型参数 T，上述那些函数则可以“综合”成为如下的同一个函数（模板）：

```
template<class T>
T max (T a, T b) {
    if(a>b) return a;
    else return b;
}
```

- **函数模板的概念，是它实际上代表着一组函数**。只是该组函数的参数类型以及返回值类型可以变化而已。

函数模板概念

- 我们定义了一个简单的函数模板，当为类型参数 T 指定为某一个具体的类型时，函数模板 max 即为一个具体求较大元的 max 函数。当然，类型参数 T 的“实参”应该是一个可以进行大于运算“＞”的系统定义的类型（如 int、float、char 等）或**已重载了运算符“＞”的用户自定义类型**。

```
template<class T>
T max (T a, T b) {
    if(a>b) return a;
    else return b;
}
```

- 因此，概括地说，利用函数模板（带类型参数的函数）一次就可定义出具有共性（除类型参数外，其余全相同）的一组（可以处理多种不同类型数据的）函数。

函数模板说明

- 函数模板定义的一般格式为：

  template < <模板参数表> > < 函数定义 >;

- template：关键字，指明为函数模板或类模板。
- 模板参数表：用尖括号括起来，一个或多个模板参数，用“，”分开。
- 模板参数：通过“class〈参数名〉”方式进行指定，其中的参数名是一个标识符，该参数名对应的实参可以是系统预定义的类型，如 int、char 等，也可以是用户自定义类型。
- 函数定义：与一般函数定义一样：〈返回类型〉〈函数名〉(〈参数表〉){〈函数体〉}；应注意的是，**在模板参数表中的参数名应出现在上述的“〈返回类型〉” 或“〈参数表〉”或“〈函数体〉”之中**（否则将没有可变性，只能定义出一个具体的函数）。

类型与非类型模板参数

- 类型模板参数：通过“class〈参数名〉”方式进行指定，其中的参数名是一个标识符，该参数名对应的实参可以是系统预定义的类型，如 int、char等，也可以是用户自定义类型。
- 非类型模板参数：必须是整型或者整型类的值，比如int、char、bool、枚举等。

```
template <typename T, int SIZE>
class MyArray {
    T arr[SIZE];
    // ...
};

int main() {
    MyArray<int, 10> myArray;  // 这里的10就是一个非类型模板参数
    return 0;
}
```

不能是浮点型

函数模板的使用

- 函数模板可以像一般函数那样**直接使用**，用户只需给出具体的实参，而系统则根据所提供的实参信息，分析确定出函数模板的各“参数名”所对应的具体类型，从而将其实例化为一个具体的函数（也称模板函数），而后再去调用执行。

```
// 函数模板
template <typename T>
void print(T a) {
    std::cout << "Value: " << a << std::endl;
}

int main() {
    print(10);  // 实例化为 void print<int>(int a) 并调用
    print(10.5);  // 实例化为 void print<double>(double a) 并调用
    print('a');  // 实例化为 void print<char>(char a) 并调用
    return 0;
}
```

函数模板的使用

- 函数模板在被调用时与同名的函数调用没有什么区别，那么系统是如何处理的呢？
  - 首先搜索程序说明中是否有参数表恰与调用参数表完全相同的**同名函数**，如果有，则调用此函数代码执行，否则执行下一步。
  - 检查是否有**函数模板**经适当实例化成为参数匹配的同名函数。如果有，调用此实例化的模板函数代码付诸执行。否则执行下一步。
  - 检查是否有**同名函数可经参数的自动转换后实现参数匹配**。如果有，则调用该函数代码执行。
  - 如果以上3种情况都未找到匹配函数，则按出错处理。

函数模板的使用


```cpp
#include <iostream>

// 同名函数
void foo(int a) {
    std::cout << "Non-template function called with int: " << a << std::endl;
}

// 函数模板
template <typename T>
void foo(T a) {
    std::cout << "Template function called with T: " << a << std::endl;
}

int main() {
    foo(10);  // 调用同名函数，因为参数表完全相同
    foo(10.5);  // 调用函数模板，因为没有参数表完全相同的同名函数，但有可以实例化的函数模板
    foo('a');  // 调用函数模板，因为没有参数表完全相同的同名函数，也没有可以自动转换参数的同名函数，但有可以实例化的函数模板
    return 0;
}
```

函数模板的使用

- 值得注意的是，模板函数调用时，与一般函数不同之处在于**它不允许类型的转换**。也就是说，调用函数的实参表在类型上必须与某一实例化了的模板函数的函数形参表完全匹配。相反，对于一般的函数定义，系统将进行实参到形参类型的自动转换。

函数模板的使用

```
template <class T>
T max (T a, T b) { //函数模板 max 的定义
    if(a>b) return a;
    else return b;
}

int main() {
    int i1= -11, i2=0;
    double d1, d2;
    cout<<max(i1,i2)<<endl; //由实参 i1、i2，系统可确定 T 对应于 int
    cout<<max(23,-56)<<endl; //由实参 23、-56，系统可确定 T 对应于 int
    cout<<max('f', 'k')<<endl; //由实参'f'，'k'，系统可确定 T 对应于 char
    cout<<"d1,d2=? ";
    cin>>d1>>d2;
    cout<<max(d1,d2)<<endl; //由实参 d1、d2，系统可确定 T 对应于 double
    //cout<<"max(23,-5.6) = "<<max(23, -5.6)<<endl;
    //出错！参数具有二义性，“double”?“int”?
    //模板函数调用时，不进行实参到形参类型的自动转换
}
```

函数模板的重载

- **函数模板也可以重载**。同样，重载的条件是两同名函数模板必须有不同的参数表。

```
//函数模板 min，求出 a 与 b 中的小者返回，a、b 应可进行“<”运算
template <class type> type min (type a, type b) {
    return (a<b?a:b);
}
//函数 min，它有两个字符串型的参数，不能直接使用“<”来进行比较，
//要通过函数 strcmp 来实现
char* min (char* a, char* b) {
    return (strcmp(a,b)<0?a:b); //返回 a、b 串中的小者
}
void main() {
    cout<<min(3, -10)<<endl; //编译器首先匹配函数 min，不成功后又去匹配并使用函数模板 min
    cout<<min(2.5,99.5)<<endl; //两个 double 数据，使用函数模板 min
    cout<<min('m','f')<<endl; //两个 char 数据，使用函数模板 min
    char* str1="The C program", * str2="The C++ program";
    //两个字符串求最小，与重载函数 min 匹配成功，使用重载函数
    cout<<min(str1, str2)<<endl;
}
```

函数模板的重载

```
//函数模板 sum 的功能为：求出 array 数组的前 size 个元素之和并返回
template <class Type> Type sum (Type * array, int size ) {
    Type total=0;
    for (int i=0;i<size;i++) //累加前 size 个数到 total
       total+=*(array+i);
    return total; }
// 函数模板 sum 的功能为：求出数组 a1、a2 的前 size 个元素之和并返回
template <class Type> Type sum (Type * a1, Type * a2, int size ) {
    Type total=0;
    for (int i=0;i<size;i++) //累加数组 a1、a2 的前 size 个数到 total
       total+=a1[i]+a2[i];
    return total; }
void main() {
    int a1[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int a2[8]={2, 4, 6, 8, 10, 12, 14, 16};
    double af[10]={1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 11};
    cout<<sum(a1,10)<<endl; //求 a1 数组前 10 个元素之和
    cout<<sum(af,10)<<endl; //求 af 数组前 10 个元素之和
    cout<<sum(a1,a2,8)<<endl; } //求 a1 数组与 a2 数组的前 8 个元素之和
```

类模板

类模板说明

类模板实例化

类模板的静态成员

类模板的友元

类模板的继承与派生

类模板说明

- 类模板又被被称为**类的类**：利用类模板（带类型参数或普通参数的类）一次就可定义出具有共性（除类型参数或普通参数外，其余全相同）的一组类；
- 通过使用类模板，可使得所定义类中的某些数据成员、某些成员函数的参数、某些成员函数的返回值都可以是**任意类型的**。通过类模板可将程序所处理对象和数据的**类型参数化**，从而使得同一段程序可用于处理多种不同类型的对象和数据，提高了程序的抽象层次与可重用性。

类模板说明

- 类模板的说明就是一个带有模板参数的类定义，其格式为：

template < <模板参数表> > **class** < 类模板名 >
{ <类模板定义体> **};**

- 模板参数表：用尖括号<、>括起来，用来说明若干个**类型形参**或**普通形参**：
  - 说明类型形参时，使用“class <**类型形参**>”的方式；
  - 说明普通形参时，使用“<**类型**> <**非类型形参**>”的方式。

```
template <class T, int i> class TestClass {
    //具有类型形参 T 与普通形参 i的类模板 TestClass
    T buffer[i];
    //T 类型的数组 buffer，数组大小随普通形参 i的大小而变化
    ...
}
```

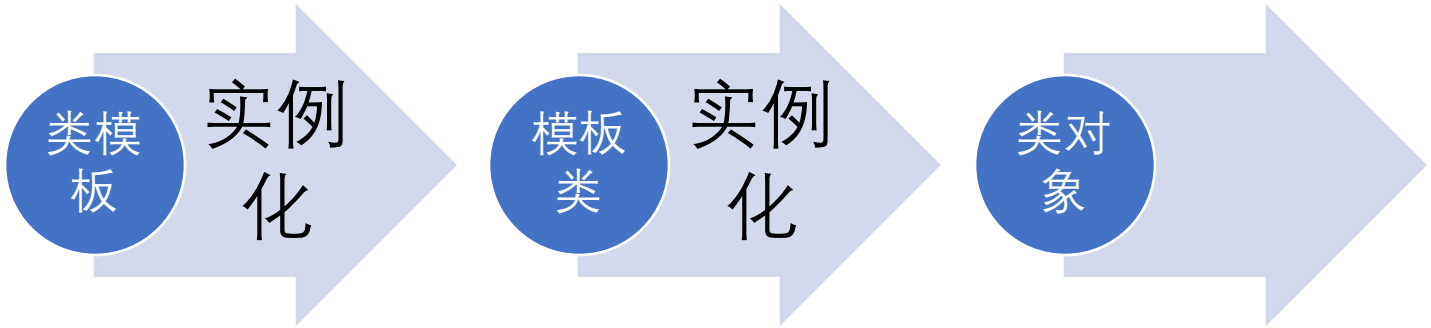
类模板的实例化

- 不能使用类模板来直接生成对象，因为其类型参数是不确定的，故需首先对模板参数指定“实参”，从而将**类模板实例化为某个具体的类**，称为模板类：

```
<类模板名><<具体实参表>>
//以 char 取代类型形参 T 并以 10 取代普通形参 i后形成的具体类
TestClass<char, 10>
TestClass<double, 8>
```

- 然后再利用模板类来说明对应的对象：

```
TestClass<double, 8> dobj1, dobj2;
```



类模板的实例化

- 不能使用类模板来直接生成对象，因为其类型参数是不确定的，故需首先对模板参数指定“实参”，从而将类模板实例化为某个具体的类，称为模板类：

```
<类模板名><<具体实参表>>
//以 char 取代类型形参 T 并以 10 取代普通形参 i后形成的具体类
TestClass<char, 10>
TestClass<double, 8>
```

- 利用类模板说明类对象时，要使用模板类来说明对应的对象：

```
TestClass<double, 8> dobj1, dobj2;
```

- 也可以用由用户定义的类型来进行对类模板实例化为模板类：

```
TestClass <complex, 15> dobj1, dobj2;
TestClass <point, 20> dobj3, dobj4;
```

类模板的成员函数说明

- 类模板的成员函数**都是模板函数**。
- 类模板的成员函数既可以在类体内进行说明和定义，也可以在类模板外进行定义。
- 在类模板内定义成员函数和普通类成员函数一样。
- 若在类体外定义类模板的成员函数时：
  - **类模板的成员函数实际上是一个函数模板，其定义格式类似于函数模板的定义；**
  - 类外定义的函数**必须在类内有说明；**
  - 要记住**应在函数模板名前加上类限定符（定义模板函数）**。

类模板的成员函数说明

- 在类体外定义类模板成员函数的一般格式如下：

  template <模板参数表>
  函数类型 类模板名 <模板参数名字表>::成员函数名 (函数参数表)
  {
  … //成员函数的函数体
  };

- 例如（注意类外定义的函数**必须在类内有说明**）：

```
template <class T, int i>
T TestClass<T, i>::getData(int j) {
    ... //成员函数的函数体
};
```

类模板的模板参数

- **类型参数：**
  - 类型参数是模板定义的主要用法，其主要反应了不同类型数据对应的同一种操作，比如算数运算和逻辑运算等，对于强类型语言十分有必要。
  - 例如矩阵可以定义为一个类，而一个反映矩阵特征的类模板，可以一次定义出分别由不同类型的元素组成的矩阵类。
- **非类型参数：**
  - 非类型参数主要用于拓展模板的应用范围，使其除了可以定义不同类型的相同操作外，还可以类似函数一样设置不同类型的参数进行区分。
  - 例如可以定义一种类模板，表示不同维数、不同元素类型的矩阵类。于是这样的一个类模板的模板参数可能除了一个类型之外，还有两个整型参数，指明矩阵的行数和列数。
  - 非类型参数只能为整型。

类模板的使用

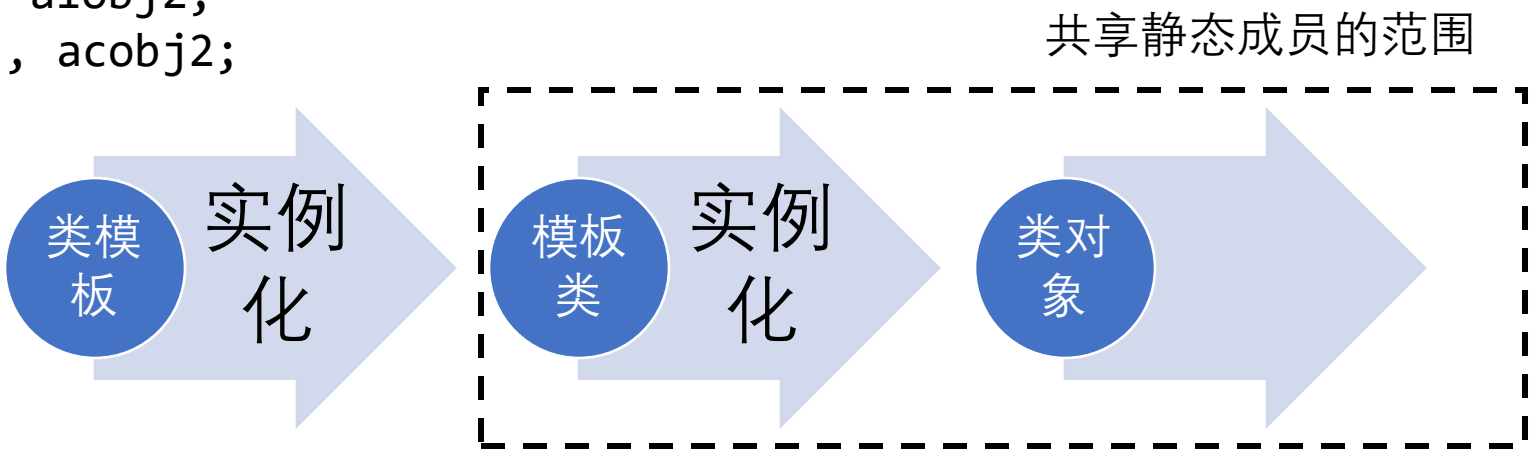
```
template <class T, int i> class TestClass {
    //类模板 TestClass，使用类型形参 T 和普通形参 i
public:
    T buffer[i]; //数组大小由普通形参 i的值指定
    T getData(int j); //负责返回 T 类型的 buffer（数组）的第 j个分量
};
template <class T, int i>T TestClass<T,i>::getData(int j) { //类外定义成员函数
    return *(buffer+j); //返回 buffer（数组）的第 j个分量
};

int main() { //创建类模板的实例对象并对它们进行使用
    TestClass<char, 5> ClassInstA;
    char cArr[6]="abcde";
    strcpy(ClassInstA.buffer, cArr); //为对象 ClassInstA的 buffer 数组赋值
    for(int i=0; i<5; i++) {
        char res=ClassInstA.getData(i); //取出 buffer 数组元素
        cout<<res<<" "; }
    cout<<endl
}
```

类模板的静态成员

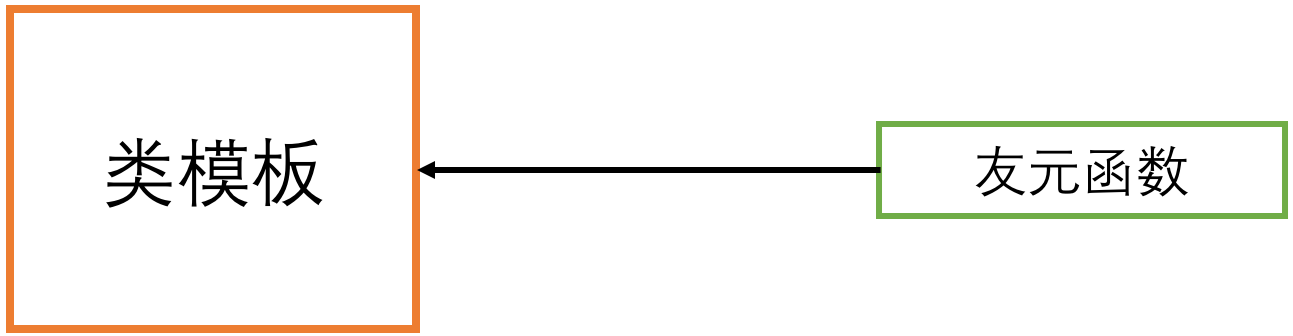
- 类模板也允许有静态成员。实际上它们是**类模板的实例化类的静态成员**。
  - 类模板的静态成员在**模板定义时不会被创建，其创建在类的实例化之后**。
  - 对于一个类模板的**同一个实例化类，其所有的对象共享其静态成员**。
  - 对于一个类模板的**不同实例化类，其模板类的对象之间不共享静态成员**。

```
template<class T>class CA {
    static T t; //类模板的静态成员 t
};
template <class T, int i> T TestClass<T,i>::t=1;
CA<int> aiobj1, aiobj2;
CA<char> acobj1, acobj2;
```



类模板的友元

- 类模板定义中允许包含友元。说明一个友元类，实际上相当于说明该类的成员函数都是友元函数：
  - 如果该友元函数为**一般函数，则它将是该类模板的所有实例化类的友元函数。**
  - 如果该友元函数是**一个函数模板，但其类型参数与类模板的类型参数无关，则该函数模板的所有实例化函数都是类模板的所有实例化类的友元。**
  - 如果该友元函数是**一个函数模板，且它与类模板的类型参数有关，该友元函数模板的实例有可能只是该类模板的某些特定实例化类的友元，而不是所有实例化的模板类的友元。**



类模板的继承与派生

类模板的继承与派生

- 一般类作基类，派生出类模板：

```
class CB
{ // CB为一般类，它将作为类模板 CA的基类
};
template <class T>
class CA : public CB
{// 被派生出的 CA为类模板，其基类 CB为一般类
    T t; // 私有数据为 T 类型的
public:
};
```

类模板的继承与派生

- 一般类作基类，派生出类模板：
- 类模板作基类，派生出新的类模板，但仅基类中用到类型参数 T：

```
template <class T>
class CB
{         // CB为类模板，它将作为类模板 CA的基类
    T t; // 私有数据为 T 类型的
public:
    T gett(){ // 用到类型参数 T
        return t;}
};
template <class T>
class CA : public CB<T>
{   // CA为类模板，其基类 CB也为类模板。注意，类型参数 T
    // 将被“传递”给基类 CB，本派生类中并不使用该类型参数 T
    double t1; // 私有数据成员
public:
};
```

类模板的继承与派生

- 一般类作基类，派生出类模板：
- 类模板作基类，派生出新的类模板，但仅基类中用到类型参数 T：
- 类模板作基类，派生出新的类模板，且基类与派生类中均使用同一个类型参数 T：

```
template <class T>
class CB
{// CB为类模板（使用类型参数 T），它
将作为类模板 CA的基类
    T t; // 数据成员为 T 类型的
public:
    T gett(){ // 用到类型参数 T
        return t;}
};
```

```
template <class T>
class CA : public CB<T>
{// CA为类模板，其基类 CB也为类模板。注意，
类型参数 T 将被 “传递”给基类 CB；本派生类中也
将使用同一个类型参数 T
    T t1; // 数据为 T 类型的
public: };
```

类模板的继承与派生

- 一般类作基类，派生出类模板：
- 类模板作基类，派生出新的类模板，但仅基类中用到类型参数 T：
- 类模板作基类，派生出新的类模板，且基类与派生类中均使用同一个类型参数 T：
- 类模板作基类，派生出新的类模板，但基类中使用类型参数T2，而派生类中使用另一个类型参数 T1：

```
template <class T2>
class CB
{// CB为类模板（使用类型参数 T2），
它将作为类模板 CA的基类
    T2 t2; // 数据为 T2 类型的
public:};
```

```
template <class T1, class T2>
class CA : public CB<T2>
{// CA为类模板，其基类 CB也为类模板。类型参数 T2 将
被 “传递”给基类 CB；本派生类中还将使用另一个类型参数
T1
    T1 t1; // 数据为 T1 类型的
public:};
```

类模板的继承与派生

- 一般类作基类，派生出类模板：
- 类模板作基类，派生出新的类模板，但仅基类中用到类型参数 T：
- 类模板作基类，派生出新的类模板，且基类与派生类中均使用同一个类型参数 T：
- 类模板作基类，派生出新的类模板，但基类中使用类型参数T2，而派生类中使用另一个类型参数 T1：
- **记住一个原则：其基类一定是一个实例化后的模板类，而不能是未实例化的类模板！**

特例版本

模板的特例版本

- 大多数类模板不能任意进行实例化。也就是说类模板的类型参数往往在实例化时不允许用任意的类（类型）作为“实参”。比如double和char类型的运算符号不一致。
- C++语言中没有对模板的“实参”类型进行检查的机制，它仅仅是通过实际操作中发生语法错误时，才能指出实例化的错误。

```
template <class T> class stack
{                    // 栈中元素类型为 T 的类模板
    T num[MAX]; // num 中存放栈的实际数据
    int top;     // top 为栈顶位置
public:  stack() { top = 0; }                    // 构造函数
    void push(T a) { num[top++] = a; } // 将数据 a“压入”栈顶
    void showtop()
    { // 显示栈顶数据
        if (top == 0) cout << "stack is empty!" << endl;
        else cout << "Top_Member:" << num[top - 1] << endl; }
};
```

该类模板要求类型T必须支持<<重载输出，否则就报错。

与之类似的还有很多，比如float、double类型不支持位移运算符，而整型类型支持，此时使用模板函数也会出错。

模板的特例版本

- 为了应对模板中对特殊类型的特殊支持，C++支持为任意模板添加特例版本，这是由C++编译器规则确定的（哪一条规则？）。

```
void stack<complex>::showtop()
{
    // 专用于 complex 类型的 showtop（专门补充的“特例版本”），显示栈顶的
    // 那一个 complex 型数据。其中的 stack<complex>为一个实例化后的模板类。
    if (top == 0)
        cout << "stack is empty!" << endl;
    else
    {
        cout << "Top_Member:" << num[top - 1].get_r();
        cout << "," << num[top - 1].get_i() << endl;
    }
}
```

假设自定义的复数类型 complex 中具有公有的成员函数 get_r()以及 get_i()，用于获取复数的实部和虚部。如此，当实例化 stack<complex>后，若使用其 showtop 函数时，将按此处的定义进行。

模板的特例版本

- 当处理某一类模板中的可变类型T型数据时，如果处理算法并不能对所有的T 类型取值做统一的处理，此时可通过**使用专门补充的所谓特例版本**来对具有特殊性的那些T 类型取值做特殊处理。**这是因为C++编译器会优先搜索类型匹配的函数或者类！**

```
void stack<complex>::showtop()
{
    // 专用于 complex 类型的 showtop（专门补充的“特例版本”），显示栈顶的
    // 那一个 complex 型数据。其中的 stack<complex>为一个实例化后的模板类。
    if (top == 0)
        cout << "stack is empty!" << endl;
    else
    {
        cout << "Top_Member:" << num[top - 1].get_r();
        cout << "," << num[top - 1].get_i() << endl;
    }
}
```

假设自定义的复数类型 complex 中具有公有的成员函数 get_r()以及 get_i()，用于获取复数的实部和虚部。如此，当实例化 stack<complex>后，若使用其 showtop 函数时，将按此处的定义进行。

应用与思考

泛用链表

泛用栈与队列

泛用动态数组

…

泛型链表

```
4   template <class T>
5   class list
6   { // 类模板 list，使用类型形参 T
7       struct node…
11      } *head, *tail; // 两个私有数据成员 head
12  public:
13      list()…
17      void Insert(T *item)…
27      void Append(T *item)…
41      T Get()…
54  }; // 类模板 list 定义结束
55  class person
56  {
57  public:
58      char name[20]; // 姓名
59      int age;       // 年龄
60      float hight;   // 身高
61  };
```

```
62  int main()
63  { // 创建类模板 list 的实例对象并对其进行使用
64      person ps;
65      list<int> link1;    // int 型空链表 link1
66      list<person> link2; // person 型空链表 link2
67      cout << " --- Input 5 person's information ---" << endl;
68      for (int i = 0; i < 5; i++)
69      { // 输入 5 个人的有关信息并进行处理
70          cout << "input " << i << " inf(name,age,hight):";
71          cin >> ps.name >> ps.age >> ps.hight;
72          link2.Insert(&ps); // 将 ps 对象插入到 link2 链表的链首
73          link1.Append(&i);  // 将当前对象的顺序号加到 link1 的链尾
74      }
75      cout << " ----- The result ----" << endl;
76      for (int i = 0; i < 5; i++)
77      {
78          ps = link2.Get();  // 取出 link2 链表首项的人员信息
79          link2.Append(&ps); // 将刚从 link2 首取来的人员信息，
80          // 再一次附加到 link2 链表的链尾
81          cout << ps.name << " " << link1.Get() << endl; // 输出 link2 人员的 name
82          // 以及 link1 链表表项中的对象顺序号
83      }
84  }
85
```

泛型栈与队列

- 基于之前的栈与队列类型，实现栈与队列的类模板；
- 为特殊类型的栈与队列提供特例函数支撑。

泛型动态数组

- 基于之前的动态数组类型，实现动态数组的类模板；
- 为特殊类型的动态数组提供特例函数支撑。
- 与C++标准库中的vector容器进行对比。

思考题

- 什么是模板？为什么要使用模板？模板可以分为几种类型？
- 函数模板只允许使用类型参数吗？定义模板的参数时应该注意什么？
- 如何对函数模板进行使用（调用）？模板函数调用时，系统是否进行实参到形参类型的自动转换？
- 是否允许函数模板与另一个函数取相同的名字？允许两个函数模板取相同的名字吗？使用这种重载 方法时需要注意些什么？

思考题

- 什么是类模板？怎样定义类模板？
- 对类模板也只允许使用类型参数吗？与函数模板是否有所不同？
- 为什么要对类模板进行实例化？怎样进行实例化？函数模板是否也需要以相同的方式进行实例化？
- 类模板的成员函数在类体外进行定义时，是否相当于一个函数模板？定义时需要注意些什么？

思考题

- 什么是类模板的静态成员？如何定义和使用一个类模板的静态成员？
- 什么是类模板的友元？如何定义和使用一个类模板的友元？
- 什么是“特例版本”？如何对类模板增加“特例版本”？如何为类模板的个别函数成员补充其“特例版本”？
- 可以使用哪些不同的方式来派生类模板？可以使用类模板作为基类来派生新的类模板吗？