作者：许富博
泛型编程打算从三个方面来学习，首先学习C++泛型编程需要的基础知识--模版，接着以C++的STL的实现为泛型编程的典型示例来学习C++的泛型编程，最后以威力更大的loki库为泛型编程的示例，进一步提高泛型编程技术。
第一部分 C++模版编程基础知识

第一章 函数模版

函数模版即类型被参数化的函数，它代表的是一个函数族。
例如：

```
1.  template <typename T>
2.  inline T const& max (T const& a, T const& b)
3.  {
4.      return a < b ? b : a;
5.  }
```

该模版定义了返回两个值中较大者的函数族。并且该函数模版要求类型T支持<操作符。
使用如下：

```
1.  int main()
2.  {
3.      int i = 42;
4.      std::cout << "max(7,i): " << ::max(7,i) << std::endl;
5.
6.      double f1 = 3.4;
7.      double f2 = -6.7;
8.      std::cout << "max(f1,f2): " << ::max(f1,f2) << std::endl;
9.
10.     std::string s1 = "mathematics";
11.     std::string s2 = "math";
12.     std::cout << "max(s1,s2): " << ::max(s1,s2) << std::endl;
13. }
```

对于每个对函数模版的调用，编译器都会生成一个对应的函数，对于上述三个调用，编译器将产生三个返回两个值的较大者的函数。
由此可见：
对于模版：编译器将进行两次编译。
1. 实例化之前：先检查模版代码本身，查看语法是否正确，例如少了分号等；
2. 实例化期间：检查模版定义代码，查看是否为有效调用，例如类型不支持的调用将是错误的。
可见，对于模版，在使用时编译器需要查看模版的定义，这不同于函数，只要声明函数原型就能通过编译，因此，模版的定义要在头文件中实现。
如上，实例化期间，编译器要推演出参数的类型，并且看到，该函数的形参类型都是一致的，因此要求两个参数类型一致，否则编译器无法推演（这里不允许类型转换），例如，如下使用是错误的。

```
1.  max(4,4.2)   // ERROR: first T is int, second T is double
```

处理如下：

1、对实参强制类型转化；

```
max(static_cast<double>(4),4.2)    // OK
```

2、显式指定类型参数的值；

```
max<double>(4,4.2)                 // OK
```

3、修改模版，使其形参为不同的类型

```
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b)
{
    return a < b ? b : a;
}
…
max(4,4.2)   // OK, but type of first argument defines return type
```

也可明确定义返回类型：

```
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b);
//需要如下使用
max<int,double,double>(4,4.2)    // OK, 但是比较麻烦
//如下定义则使用简单，但是效果一样
template <typename RT, typename T1, typename T2>
inline RT max (T1 const& a, T2 const& b);
…
max<double>(4,4.2)    // OK: return type is double
//在下一种定义中，RT被实例化为double，T1和T2分别被实例化为int和
```

函数模版也可以重载

```
inline int const& max (int const& a, int const& b)
{
    return a<b?b:a;
}

// maximum of two values of any type
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a<b?b:a;
}

// maximum of three values of any type
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return max (max(a,b), c);
```

```
}

int main()
{
    ::max(7, 42, 68);     // calls the template for three arguments
    ::max(7.0, 42.0);     // calls max<double> (by argument deduction)
    ::max('a', 'b');      // calls max<char> (by argument deduction)
    ::max(7, 42);         // calls the nontemplate for two ints
    ::max<>(7, 42);       // calls max<int> (by argument deduction)
    ::max<double>(7, 42); // calls max<double> (no argument deduction)
    ::max('a', 42.7);     // calls the nontemplate for two ints
}
```

第二章 类模版

类也可以被一种或者多种类型参数化，容器类就是一个典型例子。

```
#include <vector>
#include <stdexcept>

template <typename T>
class Stack {
  private:
    std::vector<T> elems;     // elements

  public:
    void push(T const&);      // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const {      // return whether the stack is empty
        return elems.empty();
    }
};
template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem);    // append copy of passed elem
}

template<typename T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back();         // remove last element
}

template <typename T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
```

```
    }
    return elems.back();      // return copy of last element
}
```

当在声明中需要使用该类的类型时，必须使用Stack<T>，例如，你要声明自己实现的拷贝构造函数和赋值运算符，那么该这么写：

```
template <typename T>
class Stack {
    …
    Stack (Stack<T> const&);                // copy constructor
    Stack<T>& operator= (Stack<T> const&);  // assignment operator
    …
};
```

然而，当使用类名，而不是类的类型时，就应该只用Stack，如：当你指定类的名称、构造函数、析构函数时就应该使用Stack。

类模版特化：

和函数模版的重载类似，你可以特化类模版，从而优化基于某种特定类型的实现。要特化一个类模版，你还要特化该类模版的所有成员函数。虽然也可以特化某个成员函数，但是这个做法并没有特化整个类，也就没有特化整个类模版。

例如用std::string特化Stack：

```
#include <deque>
#include <string>
#include <stdexcept>
#include "stack1.hpp"

template<>
class Stack<std::string> {
  private:
    std::deque<std::string> elems;  // elements

  public:
    void push(std::string const&);  // push element
    void pop();                     // pop element
    std::string top() const;        // return top element
    bool empty() const {            // return whether the stack is empty
        return elems.empty();
    }
};

void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem);    // append copy of passed elem
}

void Stack<std::string>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range
```

```
                ("Stack<std::string>::pop(): empty stack");
    }
    elems.pop_back();         // remove last element
}

std::string Stack<std::string>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range
                ("Stack<std::string>::top(): empty stack");
    }
    return elems.back();      // return copy of last element
}
```

局部特化：

例如类模版

```
template <typename T1, typename T2>
class MyClass {
  …
};
```

就可以有以下几种局部特化：

```
// partial specialization: both template parameters have same type
template <typename T> class MyClass<T,T> {   … };
// partial specialization: second type is int template <typename T>
class MyClass<T,int> {   … };
// partial specialization: both template parameters are pointer types
template <typename T1, typename T2> class MyClass<T1*,T2*> {   … };
```

缺省模版实参：

```
include <vector>
#include <stdexcept>

template <typename T, typename CONT = std::vector<T> >
class Stack {
  private:
    CONT elems;               // elements

  public:
    void push(T const&);      // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const {      // return whether the stack is empty
        return elems.empty();
    }
};

template <typename T, typename CONT>
void Stack<T,CONT>::push (T const& elem)
```

```
{
    elems.push_back(elem);    // append copy of passed elem
}

template <typename T, typename CONT>
void Stack<T,CONT>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back();         // remove last element
}

template <typename T, typename CONT>
T Stack<T,CONT>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems.back();      // return copy of last element
}
/*
Note that we now have two template parameters, so each definition of a member
function must be defined with these two parameters:
*/

template <typename T, typename CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem);    // append copy of passed elem
}
/*
You can use this stack the same way it was used before. Thus, if you pass a first a
only argument as an element type, a vector is used to manage the elements of this t
*/

template <typename T, typename CONT = std::vector<T> >
class Stack {
  private:
    CONT elems;    // elements
  …
};
```

第三章 非类型模版参数

一般而言，模版的参数都为类型，但是模版参数并不局限于类型，普通值也可以作为模版参数。

```
#include <stdexcept>

template <typename T, int MAXSIZE>
class Stack {
  private:
```

```
    T elems[MAXSIZE];       // elements
    int numElems;           // current number of elements
  public:
    Stack();                // constructor
    void push(T const&);    // push element
    void pop();             // pop element
    T top() const;          // return top element
    bool empty() const {    // return whether the stack is empty
        return numElems == 0;
    }
    bool full() const {     // return whether the stack is full
        return numElems == MAXSIZE;
    }
};

// constructor
template <typename T, int MAXSIZE>
Stack<T,MAXSIZE>::Stack ()
  : numElems(0)             // start with no elements
{
    // nothing else to do
}

template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::push (T const& elem)
{
    if (numElems == MAXSIZE) {
        throw std::out_of_range("Stack<>::push(): stack is full");
    }
    elems[numElems] = elem;  // append element
    ++numElems;              // increment number of elements
}

template<typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::pop ()
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    --numElems;              // decrement number of elements
}

template <typename T, int MAXSIZE>
T Stack<T,MAXSIZE>::top () const
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems[numElems-1];  // return last element
}
```

也可以为函数模版定义非类型模版参数

```
template <typename T, int VAL>
```

```
T addValue (T const& x)
{
    return x + VAL;
}
```

非类型模版参数的限制：

非类型模版参数可以为整数（包括枚举）或者指向外部链接对象的指针。浮点数和类对象是不允许为非类型模版参数的。

第四章 技巧性基础知识

1、关键字typename

例如：

```
template <typename T>
class MyClass {
   typename T::SubType * ptr;
   …
};
```

上例中，第二个typename用来说明：SubType是定义于类型T内部的一种类型，因此ptr为一个指向T::SubType类型的指针。如果不使用typename，SubType就会被认为是一个静态成员，表达式T::SubType * ptr 会被看作静态成员SubType和ptr的乘积。