

Lecture Notes on C++ Multi-Paradigm Programming

Bachelor of Software Engineering, Spring 2014

Wan Hai

whwanhai@163.com

13512768378

Software School, Sun Yat-sen University, GZ

Template, STL and Generic Programming

Agenda

- **Generic programming**(泛型编程)
- **Function template**（函数模板）
- **Class template**（类模板）
- **Non-type parameters**(非类型模板形参)



Generic programming(泛型编程)

- **Generic programming:** programming independent of any particular data types by which data(objects) of different types can be manipulated by the same codes.
- Particular data type is determined when the instance code is created from generic code.
- Generic programming is one kind of compile-time polymorphism in which data type itself is **parameterized** so that the program has polymorphism characteristics.
- **Instantiation**(实例化) is the process in which different instance codes are created from the same generic code by the compiler.

Generic programming

- In C++, **template** is used for generic programming.
 - Function template
 - Class template

Function template

- **Same processing on data of different types.**
 - **Code reduction**
 - **Can be used when the data type is not known when programming.**

Example

```
void swap(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

```
void swap(double& v1, double& v2)
{
    double temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

```
void swap(string& v1, string& v2)
{
    string temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

三个函数除了所处理对象的类型不同之外，代码几乎完全相同。即：三个函数功能类似。

Function template

//此函数模板可取代上述三函数
//的作用

```
template <typename T>
void swap( T& v1, T& v2)
{
    T temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

函数模板的一般形式

不能为空

```
template < 模板形参表 >
返回值类型 函数名 (形式参数列表)
{
    函数体语句
}
```

必须包含模板
形参表中出现
的所有形参

模板形参表形式如下：

typename 模板形参1, **typename** 模板形参2, ...

Function template

//此函数模板可取代上述三函数
//的作用

```
template <class T>
void swap( T& v1, T& v2)
{
    T temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

函数模板的一般形式

```
class < 模板形参表 >
返回值类型 函数名 (形式参数列表)
{
    函数体语句
}
```

不能为空

必须包含模板
形参表中出现
的所有形参

模板形参表形式如下：

class 模板形参1, **class** 模板形参2, ...

Function template

//由于标准库中有**swap**函数，我们所编的**swap**模板应该定义在自己的名字空间中或者
//改成另外的名称，否则会有重定义错误。我们把**swap**模板定义在**myNamespace**这
//个名字空间中，具体请参考文件夹“代码”。

```
int main()
{
    std::string s1("rabbit"), s2("bear");
    int iv1 = 3, iv2 = 5;
    double dv1 = 2.8, dv2 = 8.5;

    // 调用函数模板的实例swap(string&, string&)
    myNamespace::swap(s1, s2);

    // 调用函数模板的实例swap(int&, int&)
    myNamespace::swap(iv1, iv2);

    // 调用swap的实例swap(double&, double&)
    myNamespace::swap(dv1, dv2);
}
```

函数模板的**使用形式**
与普通函数调用相同。
但实际过程不同。

Function template

- 调用函数模板的实际过程：
 1. 模板实参推断（**template argument deduction**）：编译器根据函数调用中所给出的实参的类型，确定相应的模板实参。
 2. 函数模板的实例化（**instantiation**）：模板实参确定之后，编译器就使用模板实参代替相应的模板形参，产生并编译函数模板的一个特定版本（称为函数模板的一个实例（**instance**））（注意：此过程中不进行常规隐式类型转换）。

Function template

- 对函数模板进行重载：定义名字相同而函数形参表不同的函数模板，或者定义与函数模板同名的非模板函数，在其函数体中完成不同的行为。

如何确定调用哪个函数？

- **静态绑定**：编译时将一个函数调用关联到特定的函数体代码的过程。
- 函数调用的**静态绑定规则**
 1. 如果某一同名非模板函数的形参类型正好与函数调用的实参类型匹配（完全一致），则调用该函数。否则，进入第**2**步。
 2. 如果能从同名的函数模板实例化一个函数实例，而该函数实例的形参类型正好与函数调用的实参类型匹配（完全一致），则调用该函数实例。否则，进入第**3**步。
 3. 对函数调用的实参作隐式类型转换后与非模板函数再次进行匹配，若能找到匹配的函数则调用该函数。否则，进入第**4**步。
 4. 提示编译错误。

例子

```
template <typename T>
void demoFunc(const T v1, const T v2)
{
    cout << "the first generic version of demoFunc()" << endl;
    cout << "the arguments: " << v1 << " " << v2 << endl;
}

// 定义函数模板demoFunc的重载版本
template <typename T>
void demoFunc(const T v)
{
    cout << "the second generic version of demoFunc()" << endl;
    cout << "the argument: " << v << endl;
}

// 定义重载函数模板demoFunc的非模板函数
void demoFunc(const int v1, const int v2)
{
    cout << "the ordinary version of demoFunc()" << endl;
    cout << "the arguments: " << v1 << " " << v2 << endl;
}
```

例子

```
int main()
{
    char ch1 = 'A', ch2 = 'B';
    int iv1 = 3, iv2 = 5;
    double dv1 = 2.8, dv2 = 8.5;

    // 调用第一个函数模板的实例
    demoFunc(dv1, dv2);

    // 调用第二个函数模板的实例
    demoFunc(iv1);

    // 调用非模板函数demoFunc(int, int)
    demoFunc(iv1, iv2);

    // 调用非模板函数demoFunc(int, int)（进行隐式类型转换）
    demoFunc(ch1, iv2);

    return 0;
}
```

类模板

- 使用情景：定义可以存放任意类型对象的通用容器类
 - 例如，定义一个栈（**stack**）类，即可用于存放**int**型对象，又可用于存放**float**、**double**、**string**...甚至任意未知类型的元素
- 实现方式：为类声明一种模式，使得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，能取任意类型（包括基本类型和用户自定义类型）

类模板

- 定义形式:

template < 模板形参表 >

class 类模板名

{ 类成员声明 };

其中，模板形参表形式如下：

typename 模板形参1, **typename** 模板形参2, ...

- 在类模板以外定义其成员函数，则**函数首部**形式如下：

template <模板形参表>

返回值类型 类模板名 <模板形参名列表>::函数名(函数形参表)

例：以指针方式实现的栈类模板

```
template <typename ElementType>
```

```
//genericStack.h
```

```
class Stack {
```

```
public:
```

```
Stack();
```

```
~Stack();
```

```
void push(ElementType obj)
```

```
void pop()
```

```
ElementType getTop() const
```

```
bool isEmpty() const;
```

```
private:
```

```
struct Node // 栈结点类型
```

```
{
```

```
    ElementType element;
```

```
    Node* next;
```

```
};
```

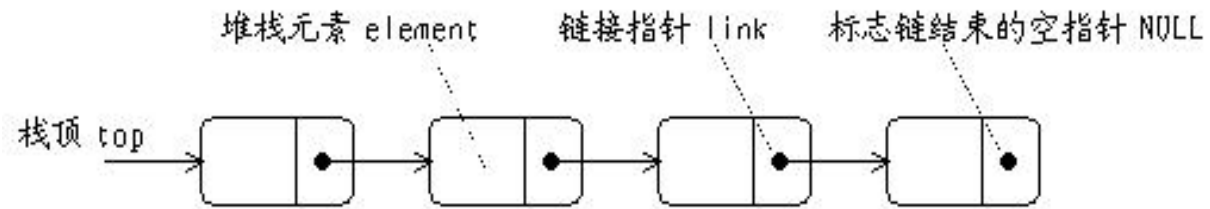
```
Node* top;
```

```
};
```

```
throw(std::bad_alloc);
```

```
throw(std::logic_error);
```

```
throw(std::logic_error);
```



```
// 结点中存放的元素
```

```
// 指向下一结点的指针
```

```
// 栈顶
```

类模板成员函数的实现

```

: //接上
template <typename ElementType>
void Stack<ElementType>::push( ElementType obj ) throw(std::bad_alloc)
{
    Node* temp;
    try {
        temp = new Node;
        temp -> element = obj;
        temp -> next = top;
        top = temp;
    }
    catch (std::bad_alloc e) { // 内存分配失败时进行异常处理
        throw;                // 重新抛出异常
    }
}

```

堆栈元素 element 链接指针 link 标志链结束的空指针 NULL

栈顶 top

temp

类模板成员函数的实现

```

:
template <typename ElementType>
void Stack<ElementType>::pop() throw(std::logic_error)
{
    Node* temp;
    if (top != NULL) {
        temp = top;
        top = top -> next;
        delete temp;
    }
    else { // 栈为空时抛出异常
        throw std::logic_error("pop from empty Stack");
    }
}

```

类模板的使用

```
#include "genericStack.h"
:
int main()
{
    Stack<int> stack; // 实例化一个保存int型元素的栈

    for (int i = 1; i < 9; i++) // 向栈中压入8个元素
        stack.push(i);

    while (!stack.isEmpty()) { // 栈不为空时循环
        cout << stack.getTop() << " "; // 显示栈顶元素
        stack.pop(); // 弹出栈顶元素
    }
    :
```

类模板的实例化

- 类模板是一个通用类模型，而不是具体类，不能用于创建对象，只有经过实例化后才得到具体类，才能用于创建对象。
- 实例化的一般形式：

类模板名 < 模板实参表 >

- 模板实参是一个实际类型。
- 一个类模板可以实例化为多个不同的具体类。

Stack<int> stack_int;

Stack<double> stack_double;

Stack<string> stack_string;

两种模板编译模式

- 包含编译模式 (**inclusion compilation model**)
 - 要求：在函数模板或类模板成员函数的调用点，相应函数的定义对编译器必须是可见的。
 - 实现方式：在头文件中用**#include**包含实现文件（也可将模板的实现代码直接放在头文件中）。
- 分离编译模式 (**separate compilation model**)
 - 要求：程序员在实现文件中使用保留字**export**告诉编译器，需要记住哪些模板定义。
 - 不是所有编译器都支持该模式。

类模板的文件组织形式(1)

//genericStack.h

#ifndef GSTACK_H

#define GSTACK_H

类模板的定义和实现代码

#endif

//client.cpp客户代码

#include "genericStack.h"

int main()

{

Stack<int> stack;

for (int i = 1; i < 9; i++)

stack.push(i);

:

}

- 模板的定义和实现都写在.h文件中。属于包含编译模式。

类模板的文件组织形式(2)

```
template <typename ElementType>
```

```
//genericStack.h
```

```
class Stack {
```

```
public:
```

```
    Stack();
```

```
    ~Stack();
```

```
    void push(ElementType obj)
```

```
    void pop()
```

```
    ElementType getTop() const
```

```
    bool isEmpty() const;
```

```
private:
```

```
    struct Node // 栈结点类型
```

```
{
```

```
        ElementType element;
```

```
        Node* next;
```

```
};
```

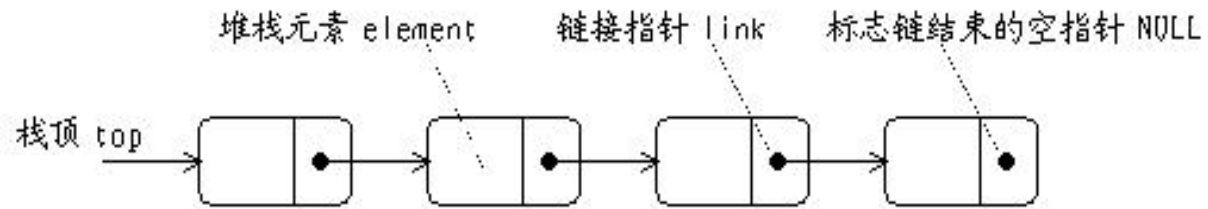
```
    Node* top;
```

```
};
```

```
    throw(std::bad_alloc);
```

```
    throw(std::logic_error);
```

```
    throw(std::logic_error);
```



```
    // 结点中存放的元素
```

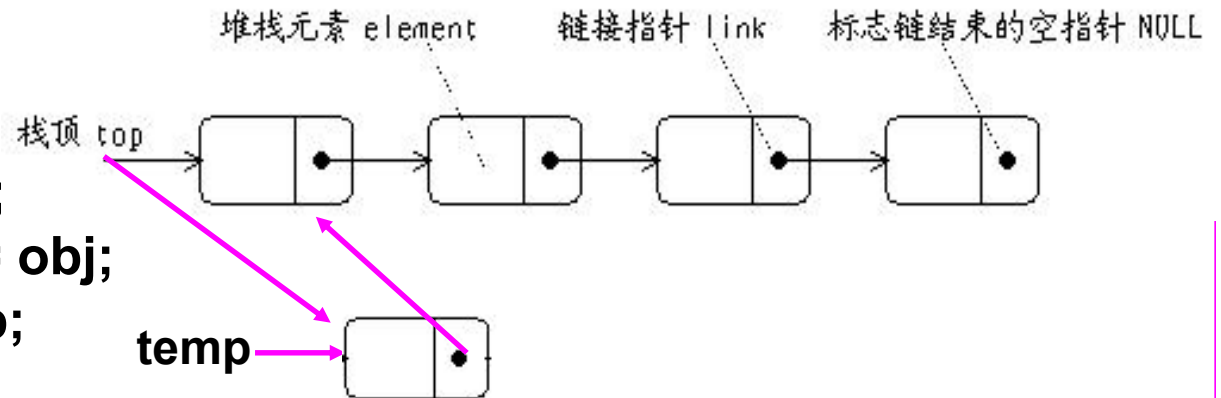
```
    // 指向下一结点的指针
```

```
    // 栈顶
```

类模板的文件组织形式(2)

//genericStack.cpp

```
template <typename ElementType>
void Stack<ElementType>::push( ElementType obj ) throw(std::bad_alloc)
{
    Node* temp;
    try {
        temp = new Node;
        temp -> element = obj;
        temp -> next = top;
        top = temp;
    }
    catch (std::bad_alloc e) { // 内存分配失败时进行异常处理
        throw;                // 重新抛出异常
    }
}
```



:

类模板的文件组织形式(2)

//genericStack.h

#ifndef GSTACK_H

#define GSTACK_H

类模板的声明

#include "generic.cpp"

#endif

//client.cpp客户代码

#include "genericStack.h"

int main()

{

:

}

//genericStack.cpp

template <typename ElementType>

void Stack<ElementType>::push(ElementType obj)

{ : }

:

类模板声明体放在头文件中，将类模板成员函数的定义和类的静态数据成员的定义放在实现文件中，与形式（1）实质一样。属于包含编译模式。

非类型模板形参

- 两类模板形参：类型形参和非类型形参
- 非类型形参
 - 相当于模板内部的常量。
 - 形式上类似于普通的函数形参。
 - 对模板进行实例化时，非类型形参由相应模板实参的值代替。
 - 对应的模板实参必须是编译时常量表达式。

类模板的非类型形参

// array-basedGStack.h 功能：定义基于数组的堆栈类模板Stack

// 以数组方式实现的堆栈类模板的定义

```

                                :
template <typename ElementType, std::size_t N>
class Stack {
public:
    Stack();
    void push(ElementType obj) throw(std::logic_error);
    void pop() throw(std::logic_error);
    ElementType getTop() const throw(std::logic_error);
    bool isEmpty() const;

private:
    ElementType elements[N];    // 堆栈中存放的元素
    std::size_t count;          // 堆栈中现有元素的数目
};

#include "array-basedGStack.cpp" // 包含源文件

```

类模板的非类型形参

// 实现文件array-basedGStack.cpp。功能：实现基于数组的堆栈类模板**Stack**

```
template <typename ElementType, std::size_t N>
```

```
Stack<ElementType, N>::Stack()
```

```
// 将堆栈初始化为空栈
```

```
{
```

```
    count = 0;
```

```
// 将元素数目置为0
```

```
}
```

```
template <typename ElementType, std::size_t N>
```

```
void Stack<ElementType, N>::push(ElementType obj) throw(std::logic_error)
```

```
// 将元素obj压入堆栈
```

```
{
```

```
    if (count < N) {
```

```
// 堆栈未滿
```

```
        elements[count] = obj;
```

```
        count++; }
```

```
    else {
```

```
// 堆栈已滿
```

```
        throw std::logic_error("push onto full stack"); }
```

```
}
```

类模板的非类型形参

// aGStackDemo.cpp 功能：演示基于数组的堆栈类模板**Stack**及非类型模板形参的使用

```
#include "array-basedGStack.h"
```

```
int main()
```

```
{
```

```
    Stack<int, 10> stack;           // 声明一个保存10个int型元素的堆栈
```

```
    :
```

```
    return 0;
```

```
}
```

函数模板的非类型形参

// printArray.cpp 功能： 演示非类型模板形参在函数模板中的使用
#include <iostream>

```
template <typename T, std::size_t N>  
void printValues(T (&arr)[N])  
{  
    for (std::size_t i =0; i != N; ++i)  
        std::cout<< arr[i] << std::endl;  
}
```

```
int main()  
{  
    int intArr[6] = {1, 2, 3, 4, 5, 6};  
    double dblArr[4] = {1.2, 2.3, 3.4, 4.5};  
  
    printValues(intArr);// 生成函数实例printValues(int (&) [6])  
    printValues(dblArr);    // 生成函数实例printValues(double (&) [4])  
}
```

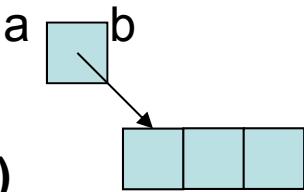

编译器会检查实参数组和形参数组的长度，两长度必须相等。所以这里不仅需要给出长度，而且必须与实参数组长度相等。

题外话

编译器不会检查实参数组和形参数组的长度，所以这里不需要给出长度。如果传递长度，要另外增加形参。

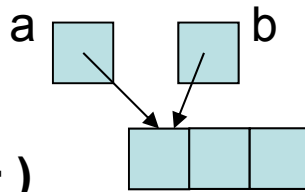
```
void f( int (&b)[3] )
{
    int i;
    for( i = 0; i < 3; i++ )
    {
        b[i]++;
        cout << b[i] << endl;
    }
    cout << "address of b in f: " << &b;

    //b++; //错误，b是指针常量
}
```



```
void f( int b[ ] )
{
    int i;
    for( i = 0; i < 3; i++ )
    {
        b[i]++;
        cout << b[i] << endl;
    }
    cout << "address of b in f: " << &b;

    b++; //正确，b是指针变量
}
```



```
address of a in main: 0012FF74
2
3
4
address of b in f: 0012FF74
```

```
int main()
{
    int a[3] = {1,2,3};
    cout << "address of a in main: " << &a ;
    f(a);
}
```

```
address of a in main: 0012FF74
2
3
4
address of b in f: 0012FF24
```

不使用非类型形参

```
template <typename T>
void printValues(T* arr, int N)
{
    for (int i =0; i != N; ++i)
        std::cout<< arr[i] << std::endl;
}

int main()
{
    int intArr[6] = {1, 2, 3, 4, 5, 6};
    double dblArr[4] = {1.2, 2.3, 3.4, 4.5};

    printValues(intArr,6);
    printValues(dblArr,4);

    :

}
```