Practice of Programming 7
template I

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

内容提纲

- 模板定义
  - 定义函数模板
  - 定义类模板
  - 模板形参
  - 模板类型形参
  - 非类型模板形参
  - 编写泛型程序
- 实例化
  - 模板实参推断
  - 函数模板的显式实参指定

模板

- 模板是创建类或函数的蓝图
  - 标准库定义了一个类模板，该模板定义了vector的含义
  - 它可以用于产生任意数量的特定类型的vector
  - vector<int>
  - vector<string>

定义函数模板

- 编写比较两个值的函数

```
int compare(const string &v1, const string &v2)
{
        if (v1 < v2) return -1;
        if (v2 < v1) return 1;
    return 0;
}

int compare(const double &v1, const double &v2)
{
        if (v1 < v2) return -1;
        if (v2 < v1) return 1;
    return 0;
}
```

定义函数模板

- 除了形参类型不同，两个函数完全一样
  - 为每种需要比较的类型都编写重复的函数体会显得很麻烦
- 而且，如果事先不知道函数要作用于哪些类型
  - 函数有可能会作用于未知类型
  - 重复函数体的策略就不适合了

- 此时，可以定义函数模板 function template

定义函数模板

- 函数模板是独立于类型的函数
  - 可以产生函数的特定类型版本

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
  return 0;
}
```

- 模板定义以template开始，后接模板形参表
  - 多个模板形参用逗号分隔

定义函数模板

- 模板形参表
  - 表示在函数中可以使用的类型或值
  - compare函数声明了名为T的类型形参
  - compare内部可以使用T引用一个类型
  - T表示哪个实际类型由编译器根据所用的函数确定

- 使用函数模板时
  - 编译器会推断哪个模板实参绑定到模板形参
  - 一旦编译器确定了实际的模板实参，就称它实例化了函数模板的一个实例
  - 编译器使用实参代替形参产生并编译该版本的函数
  - 编译器为使用的每种类型编写函数

定义类模板

- 可以像定义函数模板一样定义类模板

- Queue类
  - 能够支持不同类型的对象，将其定义成类模板
  - 包含的操作：
  - push，在队尾增加一项
  - pop，在队头删除一项
  - front，返回队头元素的引用
  - empty，之处队列中是否有元素

定义类模板

- Queue类

```
template <class Type> class Queue {
 public:
      Queue();
      Type &front();
      const Type &front () const;
      void push(const Type &);
      bool empty() const;
  private:
    // ....
};
```

Const的用法

- “Const，Const函数，Const变量，函数后面的Const”
- 文章出处：飞诺网(www.firnow.com)
- http://dev.firnow.com/course/3_program/c++/cppjs/20090302/156703.htmlhttp://dev.firnow.com/course/3_program/c++/cppjs/20090302/156703.html

- 1.用const 修饰函数的参数
- 如果输入参数采用“指针传递”，那么加const 修饰可以防止意外地改动该指针，起到保护作用。
- 例如StringCopy 函数：
- void StringCopy(char *strDestination, const char *strSource);

- 其中strSource 是输入参数，strDestination 是输出参数。给strSource 加上const修饰后，如果函数体内的语句试图改动strSource 的内容，编译器将指出错误。

Const的用法

- 2. 用const 修饰函数的返回值

- 该返回值只能被赋给加const 修饰的同类型指针
- 例如函数
- const char * GetString(void);
- 如下语句将出现编译错误：
- char *str = GetString(); //error
- 正确的用法是
- const char *str = GetString();

- 如果函数返回值采用“值传递方式”，由于函数会把返回值复制到外部临时的存储单元中，加const 修饰没有任何价值

Const的用法

- 3. const 成员函数
- 任何不会修改数据成员的函数都应该声明为const 类型
- 如果在编写const 成员函数时，不慎修改了数据成员，或者调用了其它非const 成员函数，编译器将指出错误，这无疑会提高程序的健壮性
- const 成员函数的声明看起来怪怪的：const 关键字只能放在函数声明的尾部，大概是因为其它地方都已经被占用了

定义类模板

- 类模板也是模板
  - 以template开头
  - 后接模板形参表

- 在使用类模板时，必须为模板形参显式指定实参

```
Queue<int> qi;
Queue<vector<double>> qc;
Queue<string> qs;
```

  - 编译器使用实参来实例化这个类的特定类型版本
  - 编译器用用户提供的实际特定类型代替Type，重新编写Queue类

模板形参

- 模板形参的名字没有本质含义

```
template <typename Glorp>
int compare(const Glorp &v1, const Glorp &v2)
{
        if (v1 < v2) return -1;
        if (v2 < v1) return 1;
      return 0;
}
```

- 如果是类型形参，则表示未知类型
- 如果是非类型形参，则表示是未知值

模板形参

- 模板形参的名字可以
  - 在声明为模板形参之后
  - 直到模板声明或定义的末尾处使用
- 模板形参遵循常规的名字屏蔽规则

```
typedef double T;
template <class T> T calc(const T &a, const T &b)
{
  T tmp = a; // which T ?!
   return tmp
}
```

- T定义为double的全局类型别名被名为T的类型形参所屏蔽

模板形参

- 用作模板形参的名字不能在模板内部重用：

```
template <class T> T calc(const T &a, const T &b)
{
    typedef double T;
    T tmp = a;
    return tmp
}
```

- 模板形参的名字只能在同一模板形参中使用一次

```
template <class V, class V> V calc(const V&, const V&)
```

- 模板形参的名字可以在不同的模板中重用

```
template <class T> T calc(const T&, const T&)
template <class T> int compare(const T&, const T&)
```

模板形参

- 模板可以只声明不定义

  ```
  template <class T> int compare(const T&, const T&);
  ```

- 同一模板的声明和定义中，模板形参的名字不必相同

  ```
  template <class T> T calc(const T&, const T&);
  template <class U> U calc(const U&, const U&){ … };
  ```

- 每个模板类型形参前面必须带上关键字class或typename
- 每个非类型形参前面必须带上类型名字
- 省略关键字或类型说明符都是错误的

  ```
  template <class T, U> T calc(const T&, const U&)
  ```

模板形参

- 模板类型形参
  - 由关键字class或typename后接说明符构成
  - 模板类型形参可以作为类型说明符用在模板中任何地方

- typename和class具有相同含义，可以互换使用

```
template <typename T, class U> T calc(const T&, const U&)
```

- typename看起来更加直观
  - 内置类型，即非类类型，也可以作为实际的类型形参

模板形参

- 模板非类型形参
  - 在调用函数时，非类型形参将用值代替
  - 值的类型在模板形参表中指定

```
template <class T, size_t N> void array_init(T (&parm) [N])
{
    for (size_t i = 0; i != N; ++i)
        parm[i] = 0;
}
int x[42];
double y[10];
array_init(x);
array_init(y);
```

- size_t 是无符号的整数类型，并且和操作系统无关

模板形参

- 模板非类型形参
  - 类型等价性
  - 求值结果相同的表达式被认为是等价的

```
int x[42];
const int sz = 40;
int y[sz+2];
array_init(x);
array_init(y);
```

编写泛型程序

- 编写模板程序可以不针对特定类型
  - 但是对所使用的类型还是要做一些假设

```
if (v1 < v2) return -1;
if (v2 < v1) return 1;
return 0;
```

- 隐含条件即被测试的对象支持 < 操作
- 如果被调用的对象是不支持 < 操作的，上述操作将无效

- 泛型程序要保证在模板中使用的操作作用于实参时可以正常运行

编写泛型程序

- 编写模板代码时，应该对实参类型的要求尽可能少

```
if (v1 < v2) return -1;
if (v2 < v1) return 1;
return 0;

if (v1 < v2) return -1;
if (v1 > v2) return 1;
return 0;
```

- 函数体只使用 < 操作显然对实参的要求更少

实例化

- 模板本身不是类或函数
  - 编译器用模板产生指定的类或函数的特定类型版本
  - 产生模板的特定类型实例的过程称为实例化
- 模板在使用时实例化
  - 类模板在引用实际模板类类型时实例化
  - 函数模板在调用它或用它对函数指针进行初始化或赋值时实例化

类的实例化

- 当编写Queue<int> qi时
  - 编译器自动创建名为Queue<int>的类
  - 用类型int替换模板形参的每次出现

```
template <class Type> class Queue<int> {
 public:
      Queue();
      int &front();
      const int &front () const;
      void push(const int &);
      bool empty() const;
  private:
    // ....
};
```

类的实例化

- 要使用类模板，就必须显式指定模板实参：

  ```
  Queue qs; // error
  ```

- 类模板不定义类型，特定的实例才定义类型

- 用模板类定义的类型总是包含模板实参
  - Queue不是类型
  - Queue<int> 或 Queue<string>是类型

函数模板的实例化

- 使用函数模板时，编译器通常会为我们推断模板实参：

```
int main(){
        compare(1,0);
        compare(3.14,2.7);
}
```

- 编译器会实例化两个compare版本，分别用int和double替代模板中的T

```
int compare(const int &v1, const int &v2)
int compare(const double &v1, const double &v2)
```

模板实参推断

- 从函数实参确定模板实参的类型和值的过程叫做模板实参推断(template argument deduction)
- 模板类型形参可以用于一个以上函数形参的类型

```
template <typename T>
int compare(const T &v1, const T &v2)
 {
        if (v1 < v2) return -1;
        if (v2 < v1) return 1;
      return 0;
}
```

- 必须为每个对应的函数实参产生相同的模板实参类型

```
short si;
compare(si, 1024)  // error
```

模板实参推断


```
template <typename A, typename B>
int compare(const A &v1, const B &v2)
 {
        if (v1 < v2) return -1;
        if (v2 < v1) return 1;
      return 0;
 }
```

- 函数必须用两个类型形参来定义可转换的类型

```
short si;
compare(si, 1024)
```

类型形参的实参受限转换


```
short s1, s2;
int i1, i2;
compare(i1, i2);
compare(s1,s2);
```

- 第一个调用将T绑定到int的实例
- 第二个调用将T绑定到short的实例

- 如果compare(int,int)是普通的非模板函数，第二个调用会匹配这个函数，short被转型为int

- compare是一个模板，因此将实例化一个新函数，将类型形参绑定到short上

应用于非模板实参的类型转换

- 用普通类型定义的形参可以使用常规转换

```
template <class Type> Type sum(const Type &op1, int op2)
{
    return op1 + op2;
}
```

- 由于op2的类型是固定的，在调用sum的时候，可以对传递给op2的实参应用常规转换

```
double d = 3.14;
string s1("hiya"), s2("world");
sum(1024, d); //OK:instantiates sum(int, int), converts d to int
sum(1.4, d);   //OK: instantiates sum(double, int), converts d to int
sum(s1, s2);   //error: s2 cannot be converted to int
```

函数模板的显式实参

- 有时无法确定返回值的形参类型

```
template <class T, class U> ??? sum(T, U);
sum(3, 4L);
sum(3L, 4);
```

- 指定返回类型的一种方法是引入第三个模板形参

```
template <class T1, class T2, class T3>
T1 sum(T2 &a , T3 &b){
        return a + b;
 }
int i = 1;
long lng = 2;
long val = sum<long>(i, lng);
long val1 = sum<long, int, long>(i, long);
```

Reference

- 《C++ Primer（第四版）》，Stanley B. Lippman、Barbara E. Moo、Josée LaJoie著，李师贤等人译，人民邮电出版社

Thank You!