

## 第七章 模板与泛型编程

## 是什么在影响我们的开发效率？

只谈程序员自身因素，06年写过的一段，供参考

熟练人员经过多年的积累加上自己的CodeSnip的总结，基本不用额外再查找资料。而一般的开发人员在开发过程中会花掉10-20%时间去查找资料。

熟练人员注意代码复用，并且时刻注意重构和抽取公用代码。一般开发人员是代码拷来拷去完成功能。

熟练人员非常注意查找，定位，标签等各种快捷键的使用，定位查找方便快捷，IDE环境也根据习惯定义到最方便状态。

熟练人员编码前先思考清楚整个流程，在头脑或纸张上规划好整个实现方式和方法函数的划分。一般人员想到哪里写到哪里。

熟练人员写了50行以上或更多代码才Debug一两次，一般人员写了几行代码就要Debug多次，完全通过Debug来验证代码正确性。

熟练人员注重代码的质量，单元测试和可维护性，注重各种业务逻辑的验证和边界条件的校验。一般人员只注重简单功能的简单完成。

## 是什么在影响我们的开发效率？

只谈程序员自身因素，06年写过的一段，供参考

熟练人员经过多年的积累加上自己的CodeSnip的总结，基本不用额外再查找资料。而一般的开发人员在开发过程中会花掉10-20%时间去查找资料。

熟练人员注意代码复用，并且时刻注意重构和抽取公用代码。一般开发人员是代码拷来拷去完成功能。

熟练人员非常注意查找，定位，标签等各种快捷键的使用，定位查找方便快捷，IDE环境也根据习惯定义到最方便状态。

熟练人员编码前先思考清楚整个流程，在头脑或纸张上规划好整个实现方式和方法函数的划分。一般人员想到哪里写到哪里。

熟练人员写了50行以上或更多代码才Debug一两次，一般人员写了几行代码就要Debug多次，完全通过Debug来验证代码正确性。

熟练人员注重代码的质量，单元测试和可维护性，注重各种业务逻辑的验证和边界条件的校验。一般人员只注重简单功能的简单完成。

## 是什么在影响我们的开发效率？

只谈程序员自身因素，06年写过的一段，供参考

熟练人员经过多年的积累加上自己的CodeSnip的总结，基本不用额外再查找资料。而一般的开发人员在开发过程中会花掉10-20%时间去查找资料。

熟练人员注意代码复用，并且时刻注意重构和抽取公用代码。一般开发人员是代码拷来拷去完成功能。

熟练人员非常注意查找，定位，标签等各种快捷键的使用，定位查找方便快捷，IDE环境也根据习惯定义到最方便状态。

熟练人员编码前先思考清楚整个流程，在头脑或纸张上规划好整个实现方式和方法函数的划分。一般人员想到哪里写到哪里。

熟练人员写了50行以上或更多代码才Debug一两次，一般人员写了几行代码就要Debug多次，完全通过Debug来验证代码正确性。

熟练人员注重代码的质量，单元测试和可维护性，注重各种业务逻辑的验证和边界条件的校验。一般人员只注重简单功能的简单完成。

## 1 函数模板

- 定义函数模板
- 实例化函数模板
- 模板参数类型
- 类成员模板
- 可变参函数模板

## 学习目标

- ① 掌握模板的定义和基本使用方法，包括函数模板和类模板；
- ② 学会运用模板实现泛型编程；
- ③ 掌握常用排序算法和二分查找算法。

## 7.1 函数模板

### 泛型编程

- 一种采用与数据类型无关的方式编写代码的方法，是代码重用的重要手段。

## 7.1 函数模板

### 泛型编程

- 一种采用与数据类型无关的方式编写代码的方法，是**代码重用**的重要手段。如何设计一个通用的排序算法？



## 7.1 函数模板

### 泛型编程

- 一种采用与数据类型无关的方式编写代码的方法，是**代码重用**的重要手段。如何设计一个通用的排序算法？
- **模板**是泛型编程的基础，它将数据类型参数化，为数据结构和算法的**抽象**提供**通用的代码**解决方案

请观察下面两组代码：

#### getMax 函数定义一

```
const int& getMax(const int &a, const int &b){  
    return a>b ? a : b;  
}
```

#### getMax 函数定义二

```
const string& getMax(const string &a, const string &b){  
    return a>b ? a : b;  
}
```

## 7.1 函数模板

### 泛型编程

- 一种采用与数据类型无关的方式编写代码的方法，是**代码重用**的重要手段。如何设计一个通用的排序算法？
- **模板**是泛型编程的基础，它将数据类型参数化，为数据结构和算法的**抽象**提供**通用的代码**解决方案

请观察下面两组代码：

#### getMax 函数定义一

```
const int& getMax(const int &a, const int &b){  
    return a>b ? a : b;  
}
```

#### getMax 函数定义二

```
const string& getMax(const string &a, const string &b){  
    return a>b ? a : b;  
}
```

#### 问题

- 两个函数定义有什么异同？

## 7.1 函数模板

### 泛型编程

- 一种采用与数据类型无关的方式编写代码的方法，是**代码重用**的重要手段。如何设计一个通用的排序算法？
- **模板**是泛型编程的基础，它将数据类型参数化，为数据结构和算法的**抽象**提供**通用的代码**解决方案

请观察下面两组代码：

#### getMax 函数定义一

```
const int& getMax(const int &a, const int &b){  
    return a>b ? a : b;  
}
```

#### getMax 函数定义二

```
const string& getMax(const string &a, const string &b){  
    return a>b ? a : b;  
}
```

#### 问题

- 两个函数定义有什么异同？
- 有什么办法可以简化？

## 7.1.1 定义函数模板

定义函数模板来实现一类函数的通用代码解决方案，即实现某种功能的一类函数的抽象：

### getMax 函数模板定义

```
const int& getMax(const int &a, const int &b){  
    return a>b ? a : b;  
}
```

类型参数化

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a>b ? a : b;  
}
```

## 7.1.1 定义函数模板

定义函数模板来实现一类函数的通用代码解决方案，即实现某种功能的一类函数的抽象：

### getMax 函数模板定义

```
const int& getMax(const int &a, const int &b){  
    return a>b ? a : b;  
}
```

类型参数化

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a>b ? a : b;  
}
```

### 说明

- 模板的定义以关键字 `template` 开始
- 模板参数列表放在一对尖括号里面
- 每一个参数前面用 `typename` 或 `class` 声明
- 列表含有多个模板参数则参数之间用逗号分开

## 7.1.1 定义函数模板

定义函数模板来实现一类函数的通用代码解决方案，即实现某种功能的一类函数的抽象：

### getMax 函数模板定义

```
const int& getMax(const int &a, const int &b){  
    return a>b ? a : b;  
}
```

类型参数化

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a>b ? a : b;  
}
```

### 说明

- 模板的定义以关键字 `template` 开始
- 模板参数列表放在一对尖括号里面
- 每一个参数前面用 `typename` 或 `class` 声明
- 列表含有多个模板参数则参数之间用逗号分开

### 注意

- 注意不要混淆模板参数和函数形参的概念
- 模板的声明和定义应放在同一个头文件里

## 7.1.2 实例化函数模板 — 模板参数推断

如何使用函数模板？

## 7.1.2 实例化函数模板 — 模板参数推断

如何使用函数模板？实例化模板函数，需要提供具体的数据类型或值



## 7.1.2 实例化函数模板 — 模板参数推断

如何使用函数模板？实例化模板函数，需要提供具体的数据类型或值

### 实例化方法一：隐式推断类型

```
cout << getMax(1.0, 2.5) << endl; // T被推断为double
```

生成如下函数实例

```
const double& getMax(const double &a, const double &b){  
    return a>b ? a : b;  
}
```

### 说明

编译器在编译的过程中，利用实参来推断模板参数的类型

## 7.1.2 实例化函数模板 — 模板参数推断

如何使用函数模板？实例化模板函数，需要提供具体的数据类型或值

### 实例化方法一：隐式推断类型

```
cout << getMax(1.0, 2.5) << endl; // T被推断为double
```

生成如下函数实例

```
const double& getMax(const double &a, const double &b){  
    return a>b ? a : b;  
}
```

### 说明

编译器在编译的过程中，利用实参来推断模板参数的类型

### 实例化方法二：显式指定类型

```
cout << getMax<double>(1.0, 2.5) << endl; //显式指明T为double  
cout << getMax<string>("Hi ", "C++") << endl; //显式指明T为string
```

### 说明

用户显式地指明模板参数的类型

## 7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为**类类型**时，需要为类对象添加模板使用到的相关操作：

## 7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为**类类型**时，需要为类对象添加模板使用到的相关操作：

### 示例代码

```
Fraction a(3,4),b(2,5);  
cout << getMax(a, b) << endl; // T为Fraction类型
```

## 7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为**类类型**时，需要为类对象添加模板使用到的相关操作：

### 示例代码

```
Fraction a(3,4),b(2,5);  
cout << getMax(a, b) << endl; // T为Fraction类型
```

### 函数模板实例化

```
const Fraction& getMax(const Fraction &a, const Fraction &b  
) {  
    return a>b ? a : b;  
}
```

## 7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为**类类型**时，需要为类对象添加模板使用到的相关操作：

### 示例代码

```
Fraction a(3,4),b(2,5);  
cout << getMax(a, b) << endl; // T为Fraction类型
```

### 问题

在编译上面代码时提示编译错误，原因可能是什么？

### 函数模板实例化

```
const Fraction& getMax(const Fraction &a, const Fraction &b  
) {  
    return a>b ? a : b;  
}
```

## 7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为**类类型**时，需要为类对象添加模板使用到的相关操作：

### 示例代码

```
Fraction a(3,4),b(2,5);  
cout << getMax(a, b) << endl; // T为Fraction类型
```

### 函数模板实例化

```
const Fraction& getMax(const Fraction &a, const Fraction &b  
{  
    return a>b ? a : b;  
}
```

### 问题

在编译上面代码时提示编译错误，原因可能是什么？

### 答案

在 getMax 模板内部用到了关系 > 运算，但 Fraction 类不支持关系 > 运算

## 7.1.2 实例化函数模板 — 为类类型添加模板支持

给 Fraction 类型添加关系 > 运算支持:

### Fraction 类 关系 > 运算 声明及定义

```
class Fraction{
    // 将关系>运算声明为Fraction类的友元
    friend bool operator>(const Fraction &lhs, const
        Fraction &rhs);
    // 其它成员与之前一致
    ...
};

bool operator>(const Fraction &lhs, const Fraction &rhs){
    return lhs.m_numerator*rhs.m_denominator > lhs.
        m_denominator*rhs.m_numerator;
}
```

### 说明

根据运算符重载的原则将关系运算符函数 `operator>` 作为 Fraction 类的辅助函数, 并将其声明为 Fraction 类的友元



## 7.1.3 模板参数类型

以下两组代码中，函数模板参数有什么异同？

### foo 函数定义

```
template <typename T, typename U>
T foo(const T &t, const U &u) {
    return T(t);
}
```

### maxElem 函数定义

```
template<typename T, int size>
const T& maxElem(T (&arr)[size]) {
    T *p = &arr[0];
    for (auto i = 0; i < size; ++i)
        if (*p < arr[i])
            p = &arr[i];
    return *p;
}
```

## 7.1.3 模板参数类型

以下两组代码中，函数模板参数有什么异同？

### foo 函数定义

```
template <typename T, typename U>
T foo(const T &t, const U &u) {
    return T(t);
}
```

### maxElem 函数定义

```
template<typename T, int size>
const T& maxElem(T (&arr)[size]) {
    T *p = &arr[0];
    for (auto i = 0; i < size; ++i)
        if (*p < arr[i])
            p = &arr[i];
    return *p;
}
```

### 类型参数

作为类型说明符，指定函数的返回值类型、形参类型以及函数体内对象的类型等

### 非类型参数

代表一个值，当编译器实例化该模板时必须为其提供一个常量表达式

## 7.1.3 模板参数类型

以下两组代码中，函数模板参数有什么异同？

### foo 函数定义

```
template <typename T, typename U>
T foo(const T &t, const U &u) {
    return T(t);
}
```

### maxElem 函数定义

```
template<typename T, int size>
const T& maxElem(T (&arr)[size]) {
    T *p = &arr[0];
    for (auto i = 0; i < size; ++i)
        if (*p < arr[i])
            p = &arr[i];
    return *p;
}
```

### 类型参数

作为类型说明符，指定函数的返回值类型、形参类型以及函数体内对象的类型等

### 非类型参数

代表一个值，当编译器实例化该模板时必须为其提供一个常量表达式

### 说明

maxElem 函数模板中的函数形参 arr 为一个指向含有 size 个 T 类型数据元素数组的引用

## 7.1.3 模板参数类型

调用 `maxElem` 函数：

### maxElem 函数模板实例化

```
int arr[10] = {1,8,5,3};  
int x = maxElem(arr);  
// 或者显式调用 maxElem<int,10>(arr);
```

编译器将会生成如下版本的函数：

```
const int& maxElem(int (&arr)[10]);
```

## 7.1.3 模板参数类型

调用 `maxElem` 函数：

### maxElem 函数模板实例化

```
int arr[10] = {1,8,5,3};  
int x = maxElem(arr);  
// 或者显式调用 maxElem<int,10>(arr);
```

编译器将会生成如下版本的函数：

```
const int& maxElem(int (&arr)[10]);
```

### 问题

传递数组参数还有什么方式？

## 7.1.3 模板参数类型

调用 `maxElem` 函数：

### maxElem 函数模板实例化

```
int arr[10] = {1,8,5,3};  
int x = maxElem(arr);  
// 或者显式调用 maxElem<int,10>(arr);
```

编译器将会生成如下版本的函数：

```
const int& maxElem(int (&arr)[10]);
```

### 问题

传递数组参数还有什么方式？

### 答案

还可以通过指针传递数组首地址的方式

## 7.1.3 模板参数类型 — 模板重载与特化

如果前面定义的 `getMax` 函数模板在调用过程中的实参为指针类型：

### getMax 函数调用一

```
int a = 1, b = 2;  
getMax(&a, &b);
```

### getMax 定义一

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}
```

### 说明

需要返回两个指针所指向的对象的比较结果

### 问题

`getMax` 函数模板的定义还能否满足要求？

## 7.1.3 模板参数类型 — 模板重载与特化

如果前面定义的 `getMax` 函数模板在调用过程中的实参为指针类型：

### getMax 函数调用一

```
int a = 1, b = 2;  
getMax(&a, &b);
```

### getMax 定义一

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}
```

### 说明

需要返回两个指针所指向的对象的比较结果

### 问题

`getMax` 函数模板的定义还能否满足要求？

### 答案

不能。编译器推演出的参数 `T` 为 `int*`，函数体里面的操作变成了两个指针对象的比较



## 7.1.3 模板参数类型 — 模板重载与特化

如果前面定义的 `getMax` 函数模板在调用过程中的实参为指针类型：

### getMax 函数调用一

```
int a = 1, b = 2;  
getMax(&a, &b);
```

### getMax 定义一

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}
```



```
const int* & getMax(const int* &a, const int* &b){  
    return a > b ? a : b;  
}
```

### 说明

需要返回两个指针所指向的对象的比较结果

### 问题

`getMax` 函数模板的定义还能否满足要求？

### 答案

不能。编译器推演出的参数 `T` 为 `int*`，函数体里面的操作变成了两个指针对象的比较

## 7.1.3 模板参数类型 — 模板重载与特化

为此，需要**重载**一个 `getMax` 模板函数：

### getMax 函数调用二

```
int a = 1, b = 2;  
getMax(&a, &b);
```

### getMax 函数模板重载

```
template <typename T>  
T* const & getMax( T* const &a, T* const &b){  
    return *a>*b ? a : b;  
}
```

### 说明

模板实参 `T` 的类型为 `int`，  
`*a` 和 `*b` 指向的是 `int` 对象，  
函数体里面的操作是两个  
`int` 对象的比较

## 7.1.3 模板参数类型 — 模板重载与特化

进一步，如果调用的实参是指向字符串常量的指针：

### getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

### 说明

需要返回指向字符串值较大的  
字符指针

### getMax 函数定义二

```
template <typename T>  
const T & getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}  
  
template <typename T>  
T* const & getMax(T* const &a, T* const &b){  
    return *a > *b ? a : b;  
}
```

## 7.1.3 模板参数类型 — 模板重载与特化

进一步，如果调用的实参是指向字符串常量的指针：

### getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

### 说明

需要返回指向字符串值较大的字符指针

### getMax 函数定义二

```
template <typename T>  
const T & getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}  
  
template <typename T>  
T* const & getMax(T* const &a, T* const &b){  
    return *a > *b ? a : b;  
}
```

### 问题

现有的两个 getMax 函数的定义还能否满足要求？

## 7.1.3 模板参数类型 — 模板重载与特化

进一步，如果调用的实参是指向字符串常量的指针：

### getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

### getMax 函数定义二

```
template <typename T>  
const T & getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}  
  
template <typename T>  
T* const & getMax(T* const &a, T* const &b){  
    return *a > *b ? a : b;  
}
```

### 说明

需要返回指向字符串值较大的字符指针

### 问题

现有的两个 getMax 函数的定义还能否满足要求？

### 答案

不能。\*a 和 \*b 指向的是单个字符，函数体里面的操作变成了两个字符的比较

## 7.1.3 模板参数类型 — 模板重载与特化

为此，需要特例化一个 getMax 模板函数：

### getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

### getMax 函数模板特化

```
template <>  
const char* const & getMax(const char* const &a, const char  
    * const &b){  
    return strcmp(a,b) > 0 ? a : b;  
}
```

### 说明

模板参数列表为空，表明将显式提供所有模板实参

### 说明

T 被推断为 const char\*, a 和 b 分别为一个指向 const char 对象的 const 指针的引用，函数是对两个字符串值的比较

## 7.1.3 模板参数类型 — 模板重载与特化

为此，需要特例化一个 getMax 模板函数：

### getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

### getMax 函数模板特化

```
template <>  
const char* const & getMax(const char* const &a, const char  
* const &b){  
    return strcmp(a,b) > 0 ? a : b;  
}
```

### 说明

模板参数列表为空，表明将显式提供所有模板实参

### 说明

T 被推断为 const char\*, a 和 b 分别为一个指向 const char 对象的 const 指针的引用，函数是对两个字符串值的比较

### 问题

为什么会选择特例化的版本？

### 7.1.3 模板参数类型 — 模板重载与特化

还可以通过模板特化改善算法：



## 7.1.3 模板参数类型 — 模板重载与特化

还可以通过模板特化改善算法：

### Swap 函数模板定义

```
template<typename T>
void Swap(T &a, T &b) {
    T c(a); // 复制构造对象 c
    a = b;
    b = c;
}
```

### 说明

需要构造一个辅助的局部对象 c，才能完成 a 和 b 的交换

## 7.1.3 模板参数类型 — 模板重载与特化

还可以通过模板特化改善算法：

### Swap 函数模板定义

```
template<typename T>
void Swap(T &a, T &b) {
    T c(a); // 复制构造对象 c
    a = b;
    b = c;
}
```

如果 T 是 int，可以利用模板特化做出优化：

### 说明

需要构造一个辅助的局部对象 c，才能完成 a 和 b 的交换

## 7.1.3 模板参数类型 — 模板重载与特化

还可以通过模板特化改善算法：

### Swap 函数模板定义

```
template<typename T>
void Swap(T &a, T &b) {
    T c(a); // 复制构造对象 c
    a = b;
    b = c;
}
```

如果 T 是 int，可以利用模板特化做出优化：

### Swap 函数模板特化

```
template<>
void Swap(int &a, int &b)
{
    a ^= b;
    b ^= a;
    a ^= b;
}
```

#### 说明

需要构造一个辅助的局部对象 c，才能完成 a 和 b 的交换

#### 说明

利用异或操作完成两个整数的交换，没有创建辅助对象，没有产生构造和析构行为，提高了执行效率

## 7.1.4 类成员模板

类的成员函数也可以定义为函数模板：

### 类 X 定义

```
class X{  
    void * m_p = nullptr;  
public:  
    template <typename T>  
    void reset(T *t) { m_p = t; }  
};
```

### 说明

成员函数 `reset` 定义为一个函数模板，接受不同类型的指针实参

## 7.1.4 类成员模板

类的成员函数也可以定义为函数模板：

### 类 X 定义

```
class X{  
    void * m_p = nullptr;  
public:  
    template <typename T>  
    void reset(T *t) { m_p = t; }  
};
```

### reset 函数调用

```
int i = 0;  
double d = 0;  
X x;  
x.reset(&i); // 或者 x.reset<int>(&i);  
x.reset(&d); // 或者 x.reset<double>(&d);
```

### 说明

成员函数 `reset` 定义为一个函数模板，接受不同类型的指针实参

### 说明

- 第一条 `reset` 函数调用中 `T` 被推断为 `int` 类型，`m_p` 存放整型对象 `i` 的地址
- 第二条 `reset` 函数调用中 `T` 被推断为 `double` 类型，`m_p` 存放 `double` 类对象 `d` 的地址

## 7.1.5 可变参函数模板

C++11 新标准允许我们使用**数目可变**的模板参数：

### foo 函数定义

```
template<typename... Args >
void foo(Args... args) {
    // 打印参数包args中参数的个数
    cout << sizeof...(args) << endl;
}
```

### 说明

- 可变数目的参数称为**参数包**，用省略号 “...” 表示，可包含 0 到任意个模板参数
- foo 函数的形参 args 为模板参数包类型，接受可变数目的实参

## 7.1.5 可变参函数模板

C++11 新标准允许我们使用**数目可变**的模板参数：

### foo 函数定义

```
template<typename... Args >
    void foo(Args... args) {
        // 打印参数包args中参数的个数
        cout << sizeof...(args) << endl;
    }
```

### foo 函数调用

```
foo(); // 输出: 0
foo(1,1.5); // 输出: 2
foo(1,1.5,"C++"); // 输出: 3
```

### 说明

- 可变数目的参数称为**参数包**，用省略号 “...” 表示，可包含 0 到任意个模板参数
- foo 函数的形参 args 为模板参数包类型，接受可变数目的实参

## 7.1.5 可变参函数模板 — 包展开

可以通过**递归**的方式展开函数模板参数包：

### print 函数定义

```
template<typename T, typename... Args>
ostream& print(ostream &os, const T &t, const Args&... rest
) {
    os << t << " "; // 打印第一个参数
    return print(os, rest...); // 递归调用
}

template<typename T>
ostream& print(ostream &os, const T &t) {
    return os << t; // 打印最后一个参数
}
```

### 说明

- 第一次处理参数包中的第一个参数，然后用剩余参数调用自身
- 当参数包里面只剩下一个参数时，非可变参模板与可变参模板都匹配，但是非可变参模板更特例化，编译器首选非可变参数版本



## 7.1.5 可变参函数模板 — 包展开

可以通过**递归**的方式展开函数模板参数包：

### print 函数定义

```
template<typename T, typename... Args>
ostream& print(ostream &os, const T &t, const Args&... rest
) {
    os << t << " "; // 打印第一个参数
    return print(os, rest...); // 递归调用
}

template<typename T>
ostream& print(ostream &os, const T &t) {
    return os << t; // 打印最后一个参数
}
```

### print 函数调用

```
print(cout, 1, 2.5, "C++") << endl; // 输出1 2.5 C++
```

### 说明

- 第一次处理参数包中的第一个参数，然后用剩余参数调用自身
- 当参数包里面只剩下一个参数时，非可变参模板与可变参模板都匹配，但是非可变参模板更特例化，编译器首选非可变参数版本

## 7.1.5 可变参函数模板 — 转发参数包

在泛型编程中，常常需要将参数原封不动的转发给另外一个函数

```
template<typename TYPE, typename ARG
>
TYPE* get_instance(const ARG arg){

    TYPE *ret;
    ret = new TYPE(arg);
    return ret;
}
```

## 7.1.5 可变参函数模板 — 转发参数包

在泛型编程中，常常需要将参数原封不动的转发给另外一个函数

```
template<typename TYPE, typename ARG
>
TYPE* get_instance(const ARG arg){
    TYPE *ret;
    ret = new TYPE(arg);
    return ret;
}
```

改进效率



```
template<typename TYPE, typename ARG
>
TYPE* get_instance(const ARG &arg){
    TYPE *ret;
    ret = new TYPE(arg);
    return ret;
}
```

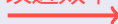
## 7.1.5 可变参函数模板 — 转发参数包

在泛型编程中，常常需要将参数原封不动的转发给另外一个函数

```
template<typename TYPE, typename ARG
>
TYPE* get_instance(const ARG arg){

    TYPE *ret;
    ret = new TYPE(arg);
    return ret;
}
```

改进效率



```
template<typename TYPE, typename ARG
>
TYPE* get_instance(const ARG &arg){

    TYPE *ret;
    ret = new TYPE(arg);
    return ret;
}
```

### 问题

如果形参类型是右值，如何处理？

## 7.1.5 可变参函数模板 — 转发参数包

两个函数的形参都是**右值引用**，forwardValue 函数调用报错，为什么？

### forwardValue 函数及 rvalue 函数定义

```
void rvalue(int &&val){}

template<typename T>
void forwardValue(T &&val) {
    rvalue(val);
}
```

### forwardValue 函数调用

```
forwardValue(42); // 错误
```

## 7.1.5 可变参函数模板 — 转发参数包

两个函数的形参都是**右值引用**，forwardValue 函数调用报错，为什么？

### forwardValue 函数及 rvalue 函数定义

```
void rvalue(int &&val){}

template<typename T>
void forwardValue(T &&val) {
    rvalue(val);
}
```

### forwardValue 函数调用

```
forwardValue(42); // 错误
```

### forwardValue 函数调用细节

```
T &&val = 42;    //等同于 auto &&val = 42
rvalue(val);
```

### 答案

- 右值引用声明 && 与类型推导结合可以与右值绑定，所以 val 为右值引用
- 右值引用 val 引用右值 42，但 val 本身是左值
- rvalue 函数只接受右值形参，但 val 是左值

## 7.1.5 可变参函数模板 — 转发参数包

两个函数的形参都是**右值引用**，forwardValue 函数调用报错，为什么？

### forwardValue 函数及 rvalue 函数定义

```
void rvalue(int &&val){}

template<typename T>
void forwardValue(T &&val) {
    rvalue(val);
}
```

### forwardValue 函数调用

```
forwardValue(42); // 错误
```

### forwardValue 函数调用细节

```
T &&val = 42;    //等同于 auto &&val = 42
rvalue(val);
```

### 答案

- 右值引用声明 && 与类型推导结合可以与右值绑定，所以 val 为右值引用
- 右值引用 val 引用右值 42，但 val 本身是左值
- rvalue 函数只接受右值形参，但 val 是左值

### 问题

可以让传入 rvalue 函数的**实参**保持原属性吗？

## 7.1.5 可变参函数模板 — 转发参数包

在 C++11 新标准下可以利用 `std::forward` 函数实现：

`std::forward` 转发左值描述性声明

```
template<typename T>
T&& forward( typename std::remove_reference<T>::type& t );
```



## 7.1.5 可变参函数模板 — 转发参数包

在 C++11 新标准下可以利用 `std::forward` 函数实现：

`std::forward` 转发左值描述性声明

```
template<typename T>
T&& forward( typename std::remove_reference<T>::type& t );
```

`forwardValue` 函数定义二

```
template<typename T>
void forwardValue(T &&val) {
    rvalue(std::forward<T>(val));
}
```

`forwardValue` 函数调用二

```
forwardValue(42);    //正确
int a = 42;
forwardValue(a);     //正确
```

## 7.1.5 可变参函数模板 — 转发参数包

在 C++11 新标准下可以利用 `std::forward` 函数实现：

### `std::forward` 转发左值描述性声明

```
template<typename T>
T&& forward( typename std::remove_reference<T>::type& t );
```

### `forwardValue` 函数定义二

```
template<typename T>
void forwardValue(T &&val) {
    rvalue(std::forward<T>(val));
}
```

### `forwardValue` 函数调用二

```
forwardValue(42);    //正确
int a = 42;
forwardValue(a);     //正确
```

### 说明

- 当传入 `forwardValue` 的实参为右值，`T` 被推断为非引用类型，`forward<T>` 返回右值引用
- 当传入 `forwardValue` 的实参为左值，`T` 被推断为左值引用类型，此时 `forward<T>` 将返回左值引用

## 7.1.5 可变参函数模板 — 转发参数包

在 C++11 新标准下可以利用 `std::forward` 函数实现：

### std::forward 转发左值描述性声明

```
template<typename T>
T&& forward( typename std::remove_reference<T>::type& t );
```

### forwardValue 函数定义二

```
template<typename T>
void forwardValue(T &&val) {
    rvalue(std::forward<T>(val));
}
```

### forwardValue 函数调用二

```
forwardValue(42);    //正确
int a = 42;
forwardValue(a);     //正确
```

### 说明

- 当传入 forwardValue 的实参为右值，T 被推断为非引用类型，forward<T> 返回右值引用
- 当传入 forwardValue 的实参为左值，T 被推断为左值引用类型，此时 forward<T> 将返回左值引用

### 注意

在 C++11 新标准下，  
&& & 和 & && 折叠为 &

# 本章结束

## 作业

- ① 在线提交系统：第七章作业
- ② 验收日期：待定