# 吴秦（Tyler）



## C++的函数重载

2010-09-05 21:26 by 吴秦, 118619 阅读, 19 评论, 收藏, 编辑

**——每个现象后面都隐藏一个本质，关键在于我们是否去挖掘**

**写在前面：**

函数重载的重要性不言而明，但是你知道C++中函数重载是如何实现的呢（虽然本文谈的是C++中函数重载的实现，但我想其它语言也是类似的）？这个可以分解为下面两个问题

- 1、声明/定义重载函数时，是如何解决命名冲突的？（抛开函数重载不谈，using就是一种解决命名冲突的方法，解决命名冲突还有很多其它的方法，这里就不论述了）
- 2、当我们调用一个重载的函数时，又是如何去解析的？（即怎么知道调用的是哪个函数呢）

这两个问题是任何支持函数重载的语言都必须要解决的问题！带着这两个问题，我们开始本文的探讨。本文的主要内容如下：

- **1、例子引入（现象）**
  - 什么是函数重载（what）？
  - 为什么需要函数重载（why）？
- **2、编译器如何解决命名冲突的?**
  - 函数重载为什么不考虑返回值类型
- **3、重载函数的调用匹配**
  - 模凌两可的情况
- **4、编译器是如何解析重载函数调用的?**
  - 根据函数名确定候选函数集
  - 确定可用函数
  - 确定最佳匹配函数
- **5、总结**

### 1、例子引入（现象）

#### 1.1、什么是函数重载（what）？

函数重载是指在同一作用域内，可以有一组具有相同函数名，不同参数列表的函数，这组函数被称为重载函数。重载函数通常用来命名一组功能相似的函数，这样做减少了函数名的数量，避免了名字空间的污染，对于程序的可读性有很大的好处。

> When two or more different declarations are specified for a single name in the same scope, that name is said to *overloaded*. By extension, two declarations in the same scope that declare the same name but with different types are called *overloaded declarations*. Only function declarations can be overloaded; object and type declarations cannot be overloaded. ——摘自《ANSI C++ Standard. P290》

看下面的一个例子，来体会一下：实现一个打印函数，既可以打印int型、也可以打印字符串型。在C++中，我们可以这样做：

```
#include<iostream>
using namespace std;
```

```
void print(int i)
{
        cout<<"print a integer :"<<i<<endl;
}

void print(string str)
{
        cout<<"print a string :"<<str<<endl;
}

int main()
{
        print(12);
        print("hello world!");
        return 0;
}
```

通过上面代码的实现，可以根据具体的print()的参数去调用print(int)还是print(string)。上面print(12)会去调用print(int)，print("hello world")会去调用print(string)，如下面的结果：（先用g++ test.c编译，然后执行）


**1.2、为什么需要函数重载（why）？**

- 试想如果没有函数重载机制，如在C中，你必须要这样去做：为这个print函数取不同的名字，如print_int、print_string。这里还只是两个的情况，如果是很多个的话，就需要为实现同一个功能的函数取很多个名字，如加入打印long型、char*、各种类型的数组等等。这样做很不友好！
- 类的构造函数跟类名相同，也就是说：构造函数都同名。如果没有函数重载机制，要想实例化不同的对象，那是相当的麻烦！
- 操作符重载，本质上就是函数重载，它大大丰富了已有操作符的含义，方便使用，如+可用于连接字符串等！

通过上面的介绍我们对函数重载，应该唤醒了我们对函数重载的大概记忆。下面我们就来分析，C++是如何实现函数重载机制的。

**2、编译器如何解决命名冲突的?**

为了了解编译器是如何处理这些重载函数的，我们反编译下上面我们生成的执行文件，看下汇编代码（全文都是在Linux下面做的实验，Windows类似，你也可以参考《一道简单的题目引发的思考》一文，那里既用到Linux下面的反汇编和Windows下面的反汇编，并注明了Linux和Windows汇编语言的区别）。我们执行命令objdump -d a.out >log.txt反汇编并将结果重定向到log.txt文件中，然后分析log.txt文件。

发现函数void print(int i) 编译之后为：（注意它的函数签名变为——**_Z5printi**）


| | | | | | | |
|---|---|---|---|---|---|---|
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |

随笔分类

.NET 2.0配置解谜系列(9)
.NET(C#) Internals (10)
【日常小记】(3)
【转载】(2)
Android开发之旅(18)
as3(1)
C/C++ Internals(15)
cocos2d-x(1)
JavaScript(1)
nginx(2)
PureMVC（AS3）剖析(5)
Python(5)
Unity3D(2)
Unix/Linux下编程(8)
服务器开发(3)
基于AIR Android应用开发(1)
客户端开发(1)
数据库(4)
网页游戏开发(23)
源码剖析：DotText源码学习(2)
源码剖析：Mongoose(5)

推荐排行榜

1. 字符集和字符编码（Charset & Encoding）(176)
2. HTTP协议及其POST与GET操作差异 & C#中如何使用POST、GET等(142)
3. Android开发之旅：环境搭建及HelloWorld(138)
4. Linux Socket编程（不限Linux）(133)
5. 浏览器缓存机制(89)
6. Unity3D手游开发实践(87)
7. C++静态库与动态库(83)
8. HTTP Keep-Alive模式(60)
9. Linux多线程编程（不限Linux）(60)
10. C++项目中的extern "C" {}(52)

阅读排行榜

1. Android开发之旅：环境搭建及HelloWorld(1067837)
2. Linux Socket编程（不限Linux）(275272)
3. Nginx安装与使用(272279)
4. 字符集和字符编码（Charset & Encoding）(235850)
5. 【日常小记】linux中强大且常用命令：find、grep(141732)
6. C++的函数重载(118616)
7. C++静态库与动态库(108328)
8. Android开发之旅：android架构(107590)
9. C/C++内存泄漏及检测(101809)
10. Android 开发之旅：view的几种布局方式及实践(101124)

2014年11月(1)
2014年2月(3)
2013年11月(1)
2013年10月(1)
2013年9月(1)
2013年5月(1)
2013年3月(2)
2013年2月(2)
2013年1月(2)
2012年12月(4)
2012年11月(1)
2012年8月(1)
2012年4月(1)
2012年3月(2)
2012年1月(1)
2011年7月(1)
2011年6月(5)
2011年5月(3)
2011年3月(2)
2011年2月(1)
2011年1月(2)
2010年12月(6)
2010年10月(1)
2010年9月(4)
2010年7月(12)
2010年6月(4)
2010年5月(14)
2010年4月(12)
2010年3月(10)

```
08048934 <_Z5printi>:
 8048934:       55                      push   %ebp
 8048935:       89 e5                   mov    %esp,%ebp
 8048937:       83 ec 18                sub    $0x18,%esp
 804893a:       c7 44 24 04 70 8b 04    movl   $0x8048b70,0x4(%esp)
 8048941:       08
 8048942:       c7 04 24 60 a0 04 08    movl   $0x804a060,(%esp)
 8048949:       e8 92 fe ff ff          call   80487e0 <_ZStlsISt11char_traits
IcEERSt13basic_ostreamIcT_ES5_PKc@plt>
 804894e:       8b 55 08                mov    0x8(%ebp),%edx
 8048951:       89 54 24 04             mov    %edx,0x4(%esp)
 8048955:       89 04 24                mov    %eax,(%esp)
 8048958:       e8 23 fe ff ff          call   8048780 <_ZNSolsEi@plt>
 804895d:       c7 44 24 04 40 88 04    movl   $0x8048840,0x4(%esp)
 8048964:       08
 8048965:       89 04 24                mov    %eax,(%esp)
 8048968:       e8 c3 fe ff ff          call   8048830 <_ZNSolsEPFRSoS_E@plt>
 804896d:       c9                      leave
 804896e:       c3                      ret
```

发现函数void print(string str) 编译之后为：（注意它的函数签名变为——**_Z5printSs**）

```
0804896f <_Z5printSs>:
 804896f:       55                      push   %ebp
 8048970:       89 e5                   mov    %esp,%ebp
 8048972:       83 ec 18                sub    $0x18,%esp
 8048975:       c7 44 24 04 82 8b 04    movl   $0x8048b82,0x4(%esp)
 804897c:       08
 804897d:       c7 04 24 60 a0 04 08    movl   $0x804a060,(%esp)
 8048984:       e8 57 fe ff ff          call   80487e0 <_ZStlsISt11char_traits
IcEERSt13basic_ostreamIcT_ES5_PKc@plt>
 8048989:       8b 55 08                mov    0x8(%ebp),%edx
 804898c:       89 54 24 04             mov    %edx,0x4(%esp)
 8048990:       89 04 24                mov    %eax,(%esp)
 8048993:       e8 78 fe ff ff          call   8048810 <_ZStlsIcSt11char_trait
sIcESaIcEERSt13basic_ostreamIT_T0_ES7_RKSbIS4_S5_T1_E@plt>
 8048998:       c7 44 24 04 40 88 04    movl   $0x8048840,0x4(%esp)
 804899f:       08
 80489a0:       89 04 24                mov    %eax,(%esp)
 80489a3:       e8 88 fe ff ff          call   8048830 <_ZNSolsEPFRSoS_E@plt>
 80489a8:       c9                      leave
 80489a9:       c3                      ret
```

我们可以发现编译之后，**重载函数的名字变了不再都是print**！这样不存在命名冲突的问题了，但又有新的问题了——变名机制是怎样的，即如何将一个重载函数的签名映射到一个新的标识？我的第一反应是：**函数名+参数列表**，因为函数重载取决于参数的类型、个数，而跟返回类型无关。但看下面的映射关系：

void print(int i) --> **_Z5printi**

void print(string str) --> **_Z5printSs**

进一步猜想，前面的Z5表示返回值类型，print函数名，i表示整型int，Ss表示字符串string，即映射为**返回类型+函数名+参数列表**。最后在main函数中就是通过**_Z5printi**、**_Z5printSs**来调用对应的函数的：

80489bc: e8 73 ff ff ff call 8048934 <**_Z5printi**>

.....

80489f0: e8 7a ff ff ff call 804896f <**_Z5printSs**>

我们再写几个重载函数来验证一下猜想，如：

void print(long l) --> **_Z5printl**

void print(char str) --> **_Z5printc**

可以发现大概是int->i，long->l，char->c，string->Ss....基本上都是用首字母代表，现在我们来现在一个函数的返回值类型是否真的对函数变名有影响，如：

```
#include<iostream>
using namespace std;

int max(int a,int b)
{
```

```
        return a>=b?a:b;
}

double max(double a,double b)
{
        return a>=b?a:b;
}
int main()
{
        cout<<"max int is: "<<max(1,3)<<endl;
        cout<<"max double is: "<<max(1.2,1.3)<<endl;
        return 0;
}
```

int max(int a,int b) 映射为**_Z3maxii**、double max(double a,double b) 映射为**_Z3maxdd**，这证实了我的猜想，Z后面的数字代码各种返回类型。更加详细的对应关系，如那个数字对应那个返回类型，哪个字符代表哪重参数类型，就不去具体研究了，因为这个东西跟编译器有关，上面的研究都是基于g++编译器，如果用的是vs编译器的话，对应关系跟这个肯定不一样。但是规则是一样的："**返回类型+函数名+参数列表**"。

既然返回类型也考虑到映射机制中，这样不同的返回类型映射之后的函数名肯定不一样了，但为什么不将函数返回类型考虑到函数重载中呢？——这是为了保持解析操作符或函数调用时，独立于上下文（不依赖于上下文），看下面的例子

```
float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
      float fl=sqrt(da);//调用sqrt(double)
      double d=sqrt(da);//调用sqrt(double)

      fl=sqrt(fla);//调用sqrt(float)
   d=sqrt(fla);//调用sqrt(float)
}
```

如果返回类型考虑到函数重载中，这样将不可能再独立于上下文决定调用哪个函数。

至此似乎已经完全分析清楚了，但我们还漏了函数重载的重要限定——**作用域**。上面我们介绍的函数重载都是全局函数，下面我们来看一下一个类中的函数重载，用类的对象调用print函数，并根据实参调用不同的函数：

```
#include<iostream>
using namespace std;

class test{
public:
        void print(int i)
        {
                cout<<"int"<<endl;
        }
        void print(char c)
        {
                cout<<"char"<<endl;
        }
};
int main()
```

```
{
        test t;
        t.print(1);
        t.print('a');
        return 0;
}
```

我们现在再来看一下这时print函数映射之后的函数名：

void print(int i) --> **_ZN4test5printEi**

void print(char c) --> **_ZN4test5printEc**

注意前面的N4test，我们可以很容易猜到应该表示作用域，N4可能为命名空间、test类名等等。这说明最准确的映射机制为：**作用域+返回类型+函数名+参数列表**

### 3、重载函数的调用匹配

现在已经解决了重载函数命名冲突的问题，在定义完重载函数之后，用函数名调用的时候是如何去解析的？为了估计哪个重载函数最适合，需要依次按照下列规则来判断：

- **精确匹配**：参数匹配而不做转换，或者只是做微不足道的转换，如数组名到指针、函数名到指向函数的指针、T到const T；
- **提升匹配**：即整数提升（如bool 到 int、char到int、short 到int），float到double
- **使用标准转换匹配**：如int 到double、double到int、double到long double、Derived*到Base*、T*到void*、int到unsigned int；
- **使用用户自定义匹配**；
- **使用省略号匹配**：类似printf中省略号参数

如果在最高层有多个匹配函数找到，调用将被拒绝（因为有歧义、模凌两可）。看下面的例子：

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c,int i,short s, float f)
{
    print(c);//精确匹配，调用print(char)
    print(i);//精确匹配，调用print(int)
    print(s);//整数提升，调用print(int)
    print(f);//float到double的提升，调用print(double)

    print('a');//精确匹配，调用print(char)
    print(49);//精确匹配，调用print(int)
    print(0);//精确匹配，调用print(int)
    print("a");//精确匹配，调用print(const char*)
}
```

定义太少或太多的重载函数，都有可能导致模凌两可，看下面的一个例子：

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i)
```

```
{
        f1(i);//调用f1(char)？ f1(long)？
        f2(0);//调用f2(char*)？f2(int*)？
}
```

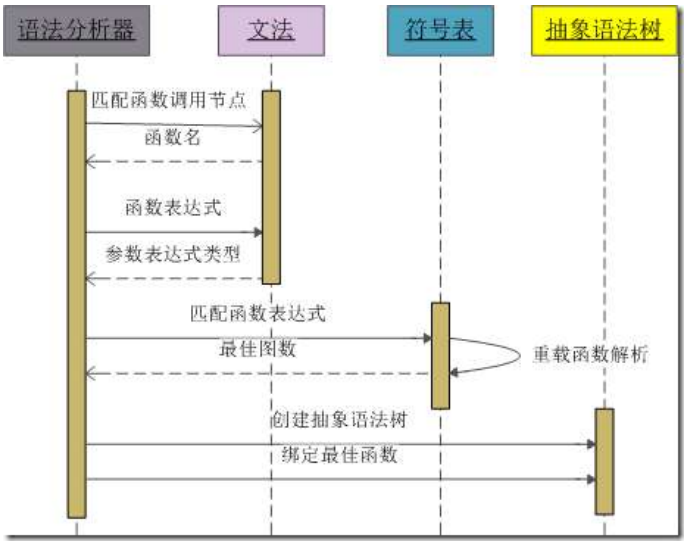
这时侯编译器就会报错，将错误抛给用户自己来处理：通过显示类型转换来调用等等（如f2(static_cast<int *>(0)，当然这样做很丑，而且你想调用别的方法时有用做转换）。上面的例子只是一个参数的情况，下面我们再来看一个两个参数的情况：

```
int pow(int ,int);
double pow(double,double);

void g()
{
        double d=pow(2.0,2)//调用pow(int(2.0),2)？
pow(2.0,double(2))？
}
```

#### 4、编译器是如何解析重载函数调用的?

编译器实现调用重载函数解析机制的时候，肯定是首先找出同名的一些候选函数，然后从候选函数中找出最符合的，如果找不到就报错。下面介绍一种重载函数解析的方法：编译器在对重载函数调用进行处理时，由语法分析、C++文法、符号表、抽象语法树交互处理，交互图大致如下：




这个四个解析步骤所做的事情大致如下：

- 由匹配文法中的函数调用，获取函数名；
- 获得函数各参数表达式类型；
- 语法分析器查找重载函数，符号表内部经过**重载解析**返回最佳的函数
- 语法分析器创建抽象语法树，将符号表中存储的最佳函数绑定到抽象语法树上

下面我们重点解释一下重载解析，重载解析要满足前面《3、重载函数的调用匹配》中介绍的匹配顺序和规则。重载函数解析大致可以分为三步：

- 根据函数名确定候选函数集
- 从候选函数集中选择可用函数集合
- 从可用函数集中确定最佳函数，或由于模凌两可返回错误

#### 4.1、根据函数名确定候选函数集

根据函数在**同一作用域内**所有同名的函数，并且要求是可见的（像private、protected、public、friend之类）。“同一作用域”也是在函数重载的定义中的

一个限定，如果不在一个作用域，不能算是函数重载，如下面的代码：

```
void f(int);

void g()
{
        void f(double);
        f(1); //这里调用的是f(double)，而不是f(int)
}
```

即**内层作用域的函数会隐藏外层的同名函数！同样的派生类的成员函数会隐藏基类的同名函数。**这很好理解，变量的访问也是如此，如一个函数体内要访问全局的同名变量要用“::”限定。

为了查找候选函数集，一般采用**深度优选**搜索算法：

step1：从函数调用点开始查找，逐层作用域向外查找可见的候选函数
step2：如果上一步收集的不在用户自定义命名空间中，则用到了using机制引入的命名空间中的候选函数，否则结束

在收集候选函数时，如果调用函数的实参类型为非结构体类型，候选函数仅包含调用点可见的函数；如果调用函数的实参类型包括类类型对象、类类型指针、类类型引用或指向类成员的指针，候选函数为下面集合的并：

- (1)在调用点上可见的函数;
- (2)在定义该类类型的名字空间或定义该类的基类的名字空间中声明的函数;
- (3)该类或其基类的友元函数;

下面我们来看一个例子更直观：

```
void f();
void f(int);
void f(double, double = 3.14);
names pace N
{
    void f(char3 ,char3);
}
classA{
    public: operat or double() { }
};
int main ( )
{
    using names pace N; //using指示符
    A a;
    f(a);
    return 0;
}
```

根据上述方法，由于实参是类类型的对象，候选函数的收集分为3步：

(1)从函数调用所在的main函数作用域内开始查找函数f的声明， 结果未找到。到main函数
作用域的外层作用域查找，此时在全局作用域找到3个函数f的声明，将它们放入候选集合；

(2)到using指示符所指向的命名空间 N中收集f ( char3 , char3 ) ；

(3)考虑2类集合。其一为定义该类类型的名字空间或定义该类的基类的名字空间中声明的函
数；其二为该类或其基类的友元函数。本例中这2类集合为空。

最终候选集合为上述所列的 4个函数f。

**4.2、确定可用函数**

可用的函数是指：函数参数个数匹配并且每一个参数都有隐式转换序列。

- (1)如果实参有m个参数，所有候选参数中，有且只有 m个参数；
- (2)所有候选参数中，参数个数不足m个，当前仅当参数列表中有省略号；
- (3)所有候选参数中，参数个数超过 m个，当前仅当第m + 1个参数以后都有缺省值。如果可用
  集合为空，函数调用会失败。

这些规则在前面的《3、重载函数的调用匹配》中就有所体现了。

**4.3、确定最佳匹配函数**

确定可用函数之后，对可用函数集中的每一个函数，如果调用函数的实参要调用它计算优先级，最后选出优先级最高的。如对《3、重载函数的调用匹配》中介绍的匹配规则中按顺序分配权重，然后计算总的优先级，最后选出最优的函数。

**5、总结**

本文介绍了什么是函数重载、为什么需要函数重载、编译器如何解决函数重名问题、编译器如何解析重载函数的调用。通过本文，我想大家对C++中的重载应该算是比较清楚了。说明：在介绍函数名映射机制是基于g++编译器，不同的编译器映射有些差别；编译器解析重载函数的调用，也只是所有编译器中的一种。如果你对某个编译器感兴趣，请自己深入去研究。

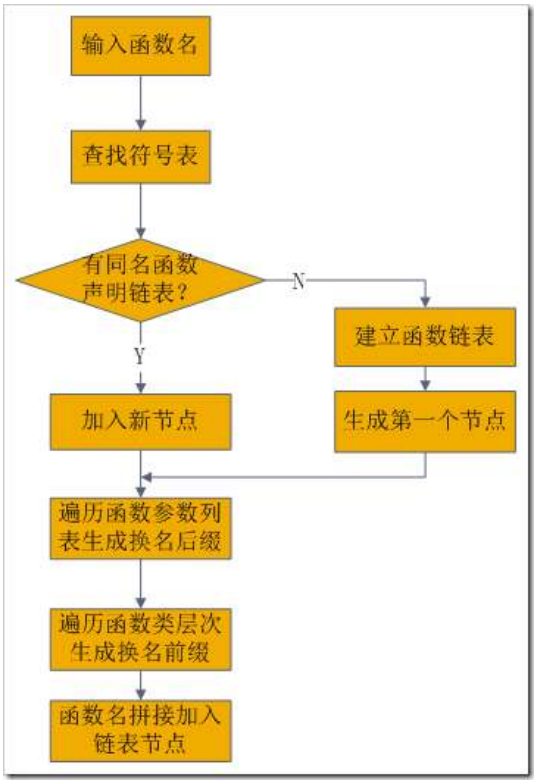
最后我抛给大家两个问题：

- 1、在C++中加号+，即可用于两个int型之间的相加、也可以用于浮点数数之间的相加、字符串之间的连接，那+算不算是操作符重载呢？换个场景C语言中加号+，即可用于两个int型之间的相加、也可以用于浮点数数之间的相加，那算不算操作符重载呢？
- 2、模板（template）的重载时怎么样的？模板函数和普通函数构成的重载，调用时又是如何匹配的呢？

**附录：一种C++函数重载机制**

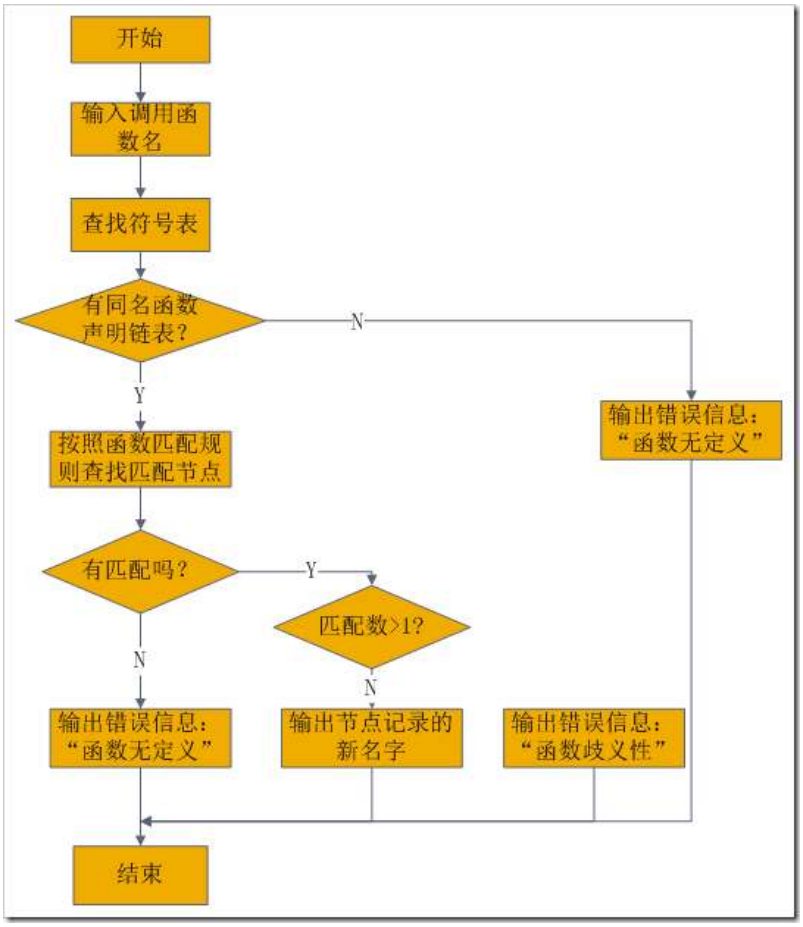
这个机制是由张素琴等人提出并实现的，他们写了一个C++的编译系统COC++（开发在国产机上，UNIX操作系统环境下具有中国自己版权的C、C++和FORTRAN语言编译系统，这些编译系统分别满足了ISOC90、AT&T的C++85和ISOFORTRAN90标准）。COC++中的函数重载处理过程主要包括两个子过程：

- 1、在函数声明时的处理过程中，编译系统建立函数声明原型链表，按照换名规则进行换名并在函数声明原型链表中记录函数换名后的名字（换名规则跟本文上面描述的差不多，只是那个int-》为哪个字符、char-》为哪个字符等等类似的差异）




图附1、过程1-建立函数链表（说明，函数名的编码格式为：<原函数名>_<作用域换名><函数参数表编码>，这跟g++中的有点不一样）

- 2、在函数调用语句翻译过程中，访问符号表，查找相应函数声明原型链表，按照类型匹配原则，查找最优匹配函数节点，并输出换名后的名字下面给出两个子过程的算法建立函数声明原型链表算法流程如图附1，函数调用语句翻译算法流程如图附2。




图附2、过程2-重载函数调用，查找链表

附-模板函数和普通函数构成的重载，调用时又是如何匹配的呢?

下面是C++创始人Bjarne Stroustrup的回答：

1)Find the set of function template specializations that will take part in overload resolution.

2)if two template functions can be called and one is more specified than the other, consider only the most specialized template function in the following steps.

3)Do overload resolution for this set of functions, plus any ordinary functions as for ordinary functions.

4)If a function and a specialization are equally good matches, the function is perferred.

5)If no match is found, the call is an error.







« 上一篇：Android开发之旅: Intents和Intent Filters（实例部分）
» 下一篇：C++中的指针与引用

分类：C/C++ Internals

ADD YOUR COMMENT

#1楼 Wang.B
2010-09-05 21:41
分析地不错
支持(0) 反对(0)
回复 引用

#2楼 kanong
2010-09-05 23:04
文章不错，支持一下!
支持(0) 反对(0)
回复 引用

#3楼 烛秋
2010-09-05 23:48
不错。我一直只分析运行时的情况，没怎么了解编译时的情况，只知道有Name-Mangling。没想过编译原理的那一层。顶了。
支持(0) 反对(0)
回复 引用

#4楼 douzifly
2010-09-06 08:41
飘过~~
支持(0) 反对(0)
回复 引用

#5楼 lgz
2010-09-06 20:53
文章不错，支持一下!
支持(0) 反对(0)

回复 引用

#6楼 aa19870406
2011-04-06 20:08
文章非常好，受教了

支持(0) 反对(0)

回复 引用

#7楼 ligand
2013-01-15 11:39
呵呵，关于函数重载，一个被大多数遗忘或者疏漏的地方：
//这种形参类型、形参个数完全相同的函数也是可以重载的，只要它的const/volatiles属性不同即可！
//此例已经在VC++2010与GCC4.6.3测试通过
```
struct foo{
int bar() const {return 2;} ; //第一个重载函数
int bar() {return 1;}; //第二个重载函数
};
int main()
{
const foo a;
a.bar(); //单步跟踪，发现是调用第一个重载函数
foo b;
b.bar(); //单步跟踪，发现是调用第二个重载函数
}
```

支持(2) 反对(0)

回复 引用

#8楼 tool
2013-03-22 20:44
文章分析得很好

支持(0) 反对(0)

回复 引用

#9楼 AN.树畔
2013-06-26 16:17
这叫一个通透啊！顶起。。。

支持(0) 反对(0)

回复 引用

#10楼 小猪尾巴1990
2013-10-03 14:10
写的可真好啊 没发现有哪本书比这个写的更好的了

支持(0) 反对(0)

回复 引用

#11楼 Jason.ygx
2013-11-12 16:11
好东西，很值得学习

支持(0) 反对(0)

回复 引用

#12楼 鳳大人
2014-01-09 16:05
浏览几篇楼主大人的文章，真是敬佩，每一篇竟如此认真，而且联系紧密、环环相扣，实在是大爱之人~

支持(0) 反对(0)

回复 引用

#13楼 heaven天明
2014-03-26 20:01
顶~

支持(0) 反对(0)

回复 引用

#14楼 Peaceful-蓝蓝的
2014-04-22 15:18
很不错

支持(0) 反对(0)

回复 引用

#15楼 Miller_S

**2016-01-09 20:19**

不单单学习到了函数重载的知识，还让我领略了您对待知识的态度和总结方法，感谢！

支持(0) 反对(0)

回复 引用

**#16楼 时光流逝的味道**

**2016-06-08 12:51**

第一个不是要加个头文件么
#include<string>

支持(0) 反对(0)

回复 引用

**#17楼 lulipro**

**2016-10-21 13:02**

博主，我有个异议："Z后面的数字代表返回类型" 这个猜想我认为不妥。
原因1:无论是C++还是Java，函数的重载都不以返回类型为依据。也就是如 double foo() 和 int foo() 不能形成重载。如果按你的说法，他们被汇编后因为返回类型不同，形成不同从重载，这是矛盾的。

原因2：下面的2个函数，他们的返回值相同，只不过名称空间不同，
全局的add函数汇编后名称为：_Z3addii 名称空间test中的汇编后为 _ZN4test3addEii

```
1  int add(int a,int b);
2
3
4  namespace test
5  {
6
7          int add(int a,int b);
8
9  }
10
11
12 int main()
13 {
14
15         int re1  = add(1,2);
16         int re2  = test::add(3,4) ;
17
18         return 0;
19 }
20
21 namespace test{
22
23         int add(int a,int b)
24         {
25                 return a+b ;
26         }
27 }
28
29 int add(int a,int b)
30 {
31         return a+b;
32 }
```

支持(1) 反对(0)

回复 引用

**#18楼 odisey**

**2018-06-22 12:25**

返回值类型并不会被重整到函数名称之中。即，仅仅依靠返回值类型的不同，无法构成重载。在这一点上，作者严重误导了初学者。

支持(0) 反对(0)

回复 引用

**#19楼 e_shannon**

**2018-08-06 10:40**

@ odisey
有人能对这个评论回复吗?

另外作者的分析方法和探索精神值得学习

支持(0) 反对(0)

回复 引用

刷新评论 刷新页面 返回顶部

发表评论

昵称: Justry2015

评论内容:

提交评论 退出 订阅评论

[Ctrl+Enter快捷键提交]