# 深入C++探索--- 三 函数的调用方式

## 零、声明

1. 这个系列《深入C++探索》，以汇编和内存角度对C++的一些基本的运行机制进行讨论；
2. 这个系列中出现的汇编代码，都是来自于VS2010的Debug环境下生成的；
3. 如果觉得文章对你有帮助而需要转载，也请阁下能够注明出处；
4. 如果觉得博文对问题的讨论有误，也可以给博主留言。

## 调用方式

在Visual C++的调用约定：\_\_cdecl, \_\_stdcall, \_\_fastcall等

### 1.1.\_\_cdecl

C Declaration。使用C语言的默认调用方式：参数从右往左压栈，栈的清空由调用者来负责。在生成的汇编代码中，函数名以下划线 \_ 开头。

这种的函数约定比较特殊的一点是，对参数的个数没有限制。比如

int \_\_cdecl fun1\_cdecl(int x,...)

{

va\_list argp;

int argno = 0;

int para;

va\_start( argp, x );

while (1)

{

para = va\_arg( argp, int);

int temp = 1;

}

va\_end( argp );

}

参数个数可以固定，在调用的时候也可以随便传入,如：

fun1\_cdecl(1,2,3,4,6,7);

我们再看一下具体的实现是如何的：

调用,调用的汇编代码就不再解释了，参数从右往左，由调用者平衡栈。

fun1\_cdecl(1,2,3,4,6);

00291697 push 6

00291699 push 4

0029169B push 3

0029169D push 2

0029169F push 1

002916A1 call fun1\_cdecl (29117Ch)

002916A6 add esp,14h

实现：

int \_\_cdecl fun1\_cdecl(int x,...)

{

...

va\_list argp; //这里的va\_list argp,其实就是 char \* argp;

int argno = 0; //略

0020160E mov dword ptr [argno],0

int para; //略

va\_start( argp, x );

00201615 lea eax,[ebp+0Ch] // eax = ebp + 0xC，这是一个地址

//这个地址中存放了压入的参数

00201618 mov dword ptr [argp],eax //将上述的地址存放在argp中

while (1) //开始循环

0020161B mov eax,1 //循环条件

00201620 test eax,eax //循环条件

00201622 je fun1\_cdecl+4Fh (20163Fh) //不符合条件就跳转至0020163F就是跳出循环

{

para = va\_arg( argp, int); //以下所有的操作主要为了实现

//para = \*argp;argp += sizeof(int);

00201624 mov eax,dword ptr [argp] //将argp中存放的地址赋给eax

00201627 add eax,4 //将eax向右移动4个字节

0020162A mov dword ptr [argp],eax //将eax重新存放在argp

0020162D mov ecx,dword ptr [argp] //将argp中存放的地址赋给ecx

00201630 mov edx,dword ptr [ecx-4] //获取ecx-4位置处的值赋给edx

00201633 mov dword ptr [para],edx //获取edx 赋给para

int temp = 1;

00201636 mov dword ptr [temp],1//略

}

0020163D jmp fun1\_cdecl+2Bh (20161Bh)

va\_end( argp );

0020163F mov dword ptr [argp],0 //将argp赋值为0

}

...

ret

这里有一点还是在补充说明以下，我们来观察一下argp，在进入循环前argp的地址是0x004AF6B4如下：

0x004AF6B4 **c8 f6 4a 00** cc cc cc cc a4 f7 4a 00 a6 16 29 00 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00

这边的存放里一个地址0x004af6c8，我们再看一下这个地址的内存情况：

**0x004AF6C8** 02 00 00 00 03 00 00 00 04 00 00 00 06 00 00 00 00 00 00 00 00 00 00 00 00 e0 fd 7e

这里存放的就是调用这个函数时压入的参数，因为压入的第一个参数可以直接获得，所以这里argp所指向的第一个参数时压入的第二个参数。

### 1.2．\_\_stdcall

1.参数从右向左压入堆栈

2.函数被调用者修改堆栈

3.函数名自动加前导的下划线，后面紧跟一个@符号，其后紧跟着参数的尺寸，这一个命名的也就限死了不能用可变数量的参数

相对于的\_\_cdcel较为简单

fun1\_stdcall(1,2);

int \_\_stdcall fun1\_stdcall(int x,int y)

{

return x+y;

}

详细代码：

调用：

fun1\_stdcall(1,2);

001C168E push 2 //压入参数，从右往左

001C1690 push 1 //压入参数，从右往左

001C1692 call fun1\_stdcall (1C108Ch)

实现：

int \_\_stdcall fun1\_stdcall(int x,int y)

{

...

return x+y;

001515CE mov eax,dword ptr [x]

001515D1 add eax,dword ptr [y]

}

...

001515DA ret 8 //这个就是stdcall需要函数本身维持栈的平衡

//因为压入了两个int所以这里需要移动8来平衡。

这里前面有介绍过，最有一点留意的是ret 8。在win32应用程序里,宏APIENTRY，WINAPI，都表示\_stdcall。

## 三、总结

其他的调用方式这里不再做展开讨论了。