**利用反汇编手段解析C语言函数**

技匠志 2020-02-13 20:33:56

1、问题的提出

函数是 C语言中的重要概念。利用好函数能够充分利用系统库的功能写出模块独立、易于维护和修改的程序。函数并不是 C 语言独有的概念，其他语言中的方法、过程等本质上都是函数。

2、解决方法

在《微机原理》 课程介绍了堆栈、汇编语言等必要的相关知识之后，通过在高级语言开发环境下反汇编C 语言程序代码，通过分析汇编代码来理解函数调用中的堆栈变化，可以在实践中理解高级语言和低级语言的底层映射关系，理解函数调用的实质。本文通过在 Visual C++6.0 下反汇编一个 32 位 C语言程序的部分代码来解析解释函数调用的具体过程。

3、函数调用过程

函数调用过程主要由参数传递、地址跳转、局部变量分配和赋初值、执行函数体，结果返回等几个步骤组成。

3.1、参数传递及函数跳转

参数由实参传递给形参。在底层实现上，即是实参按照函数调用规定压入堆栈。参数传递完成后就通过CALL指令由当前程序跳转到子程序处。

3.2、局部变量分配并赋值

函 数的"{"被认为是分配局部变量空间的时机。在汇编层面局部变量分配体现为堆栈中以 EBP 寄存器为基址向低地址端分配的一个连续区域，通过 EBP 寄存器的相对寻址方式来寻址函数内的局部变量。由于堆栈增长的方向是高地址端到低地址端，因此函数中先定义的局部变量地址较大，后定义的变量地址逐渐变小，相邻定义的变量其地址一定相邻[2]。由于全局数据和局部数据定义在不用的数据区而并不与局部变量相邻，根据程序局部性原理，相邻的数据会被缓存，因此对相同的运算，局部变量作为操作数的运算效率就可能高于有全局变量参与的运算。同时，局部变量分配和回收只需要移动堆栈指针ESP，因此效率最高。

3.3、寻址函数的参数

参数存放在以 EBP 为基址的高地址端。对参数的访问同样是通过EBP 寄存器相对寻址操作来实现。

3.4、执行函数体内的语句

函数内和具体功能相关的语句被转化成一系列汇编语句。

3.5、返回值

return 语句将返回值返回到主调函数。在底层，参数是通过 EAX 寄存器或 EDX 寄存器传递给主调函数。

3.6、返回主调函数

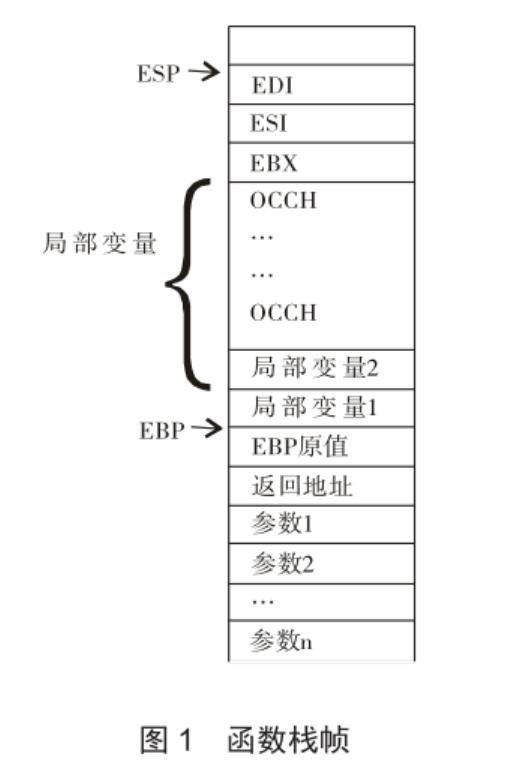
函数的"}"被解释为函数体已经执行完。遇到"}"时，会将堆栈中的局部变量、程序中压入堆栈的寄存器的值全部弹出，将之前 CALL指令执行时压入堆栈的函数返回地址弹到指令指针寄存器 EIP，从而返回到主调函数。

3.7、堆栈平衡

堆栈平衡指的是将函数调用前压入堆栈的参数弹出堆栈，使堆栈恢复到其调用前的状态[3]。由于函数调用完成后，参数就是无用的数据了，因此需要将其移出堆栈。

在 C语言中不需要进行堆栈平衡。而在汇编层面上却根据调用约定来确定由主调函数或是被调函数完成堆栈平衡。

C语言函数调用堆栈常见形式如图 1 所示[4]：



参数由主调函数压入堆栈，CALL 指令将函数返回地址入栈。进入子函数后，需要保存 EBP 原值、分配局部变量空间、保存寄存器初始值。函数内通过"EBP-位移量"方式访问局部变量，通过"EBP+位移量"方式访问参数[5]。

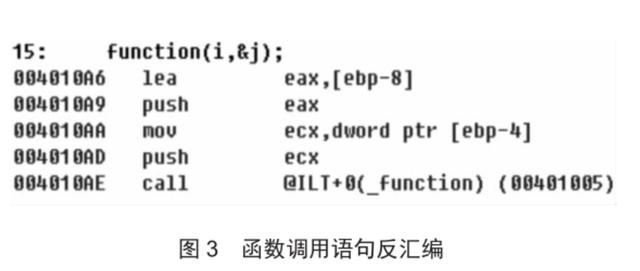
每发生一次函数调用，就会在堆栈中建立一个栈帧，栈帧在函数调用后释放。但是系统的堆栈资源有限，因此如果函数调用（如递归调用）层数过多，则可能发生堆栈溢出错误。

4.反汇编代码分析

以下将函数 function 的调用相关代码在VisualC++6.0 Debug模式反汇编，通过对汇编代码的分析揭示函数调用的关键点和细节。完整的 C语言程序代码如图 2 所示：

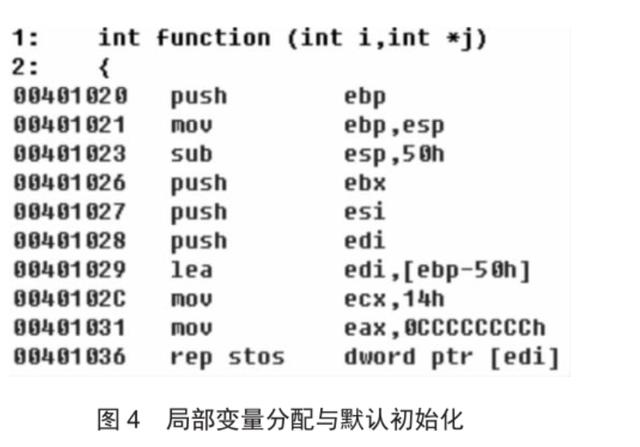


Function(i，&j)语句的反汇编代码如图 3 所示：



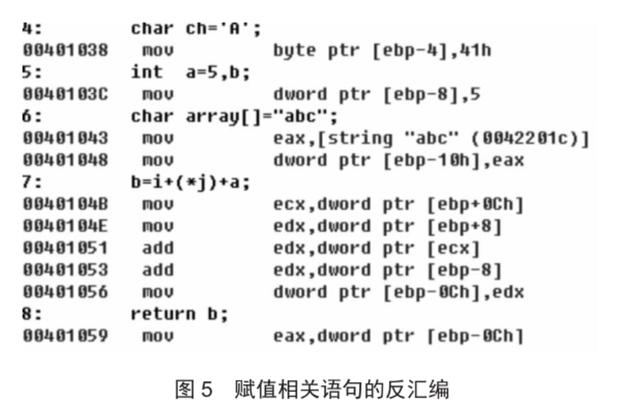
先 找到主函数中的局部变量 i，j（其在堆栈中位置为 EBP- 8和 EBP- 4），将其压入堆栈。Visual C/C++的编译器对 C 语言程序的默认函数约定为 \_cdecl[6]。此参数入栈约定为自右向左，并且对函数名前加"\_"修饰符。先将 j 的地址压入堆栈，后将 i 的值压入堆栈。通过 call 指令调用函数。从 Call 指令可见 fuction函数编译后加了"\_"修饰符。Call 指令执行时自动将函数的返回地址入栈，之后转到 function 定义处开始执行此函数。

对funciton函数的"{"的反汇编结果如图 4 所示：



在函数内，遇到"{"时分配局部空间，并用值"0xCCH"进行初始化。未在定义时初始化的局部变量其初值就与"0xCCH"相关。因此 int 类型变量由于占四个字节，其初值为 - 858993460（0xCCCCC-CCCH）；两个连续的 0xCCH 对应汉字"烫"字，因此当以字符形式显示函数内未初始化的变量时会显示为"烫烫…"；指针类型变量就指向了地址为 0xCCCC-CCH 的内存。由此在调试模式下能很容易发现未初始化的变量。

堆栈基本的存储单位为四字节，对于小于四字节的数据按四字节对齐方式分配空间。因此 char 类型变量 ch 虽然数据本身需要两个字节，也分配了四个字节空间。array 字节数组分配空间时每个字符占一个字节，不够四个字符时按四字节对齐存放。因此局部变量空间总数为 40H+4+4×2+4=50H。局部变量 ch 的地址为 EBP- 4，a、b 的地址分别为 EBP- 8 ，EBP- 0CH,array数组的地址为 EBP- 10h。函数左括号右括号间的所有的语句反汇编结果如图 5 所示：



若变量有初值，则反汇编就会为其生成一条 Mov指令为其赋值。对于没有初值的变量其每个字节都为0xCCH。对于字符数组，情况稍微复杂一些。字符串常量"abc"被存放在全局数据区中。当需要引用其值对数组进行初始化时，实际是将全局数据拷贝到堆栈中的局部数组 array里。由于寄存器是 32 位，每次最多只能赋值 4 个字符，因此对数组赋初值的语句反汇编后可能产生一至多条汇编语句。对数组内容的访通过[ "EBP+ 数组首地址 + 偏移量]的寄存器间址来完成，因此局部数组初始化费时但访问时的效率高。

在函数内访问局部变量和参数通过 [EBP + 位移量 /- 位移量]来完成。函数返回值被放到 EAX 寄存器中供主调函数使用。

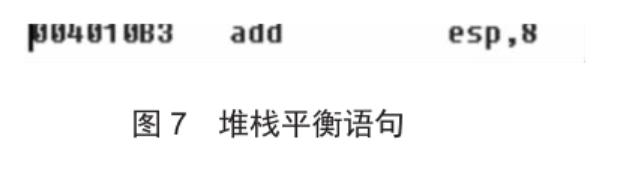
可见，在汇编层面上，函数内部并不存储局部变量，局部变量只有当函数调用发生时才会在栈上为函数分配空间。因此当函数调用后返回局部变量的值是错误的。

遇到函数"}"时的操作如图 6 所示：



将寄存器 EDI、ESI、EBX 恢复原值；将 ESP 调回到 EBP 处；将 EBP原值弹出。此时 ESP 指向函数返回地址。执行出栈指令，将函数的返回地址弹入 EIP 寄存器返回到主调函数。此时堆栈中只残留有调用函数时压入的参数还没有清理。

主调函数中的堆栈平衡语句如图 7 所示：



根据 \_cdecl 约定，需要由主调函数完成堆栈平衡。主调函数根据压入堆栈的参数的数目 2 和参数大小，利用指令 add ESP，8 将参数全部弹出。此时堆栈就恢复到其调用前的状态。一个完整的函数调用过程完成。