C++ Tricks 2.3 I386平台C函数内部的栈分配

从 farseerfc.wordpress.com 导入

2.3 I386平台C函数内部的栈分配

函数使用栈来保存局部变量，传递函数参数。进入函数时，函数在栈上为函数中的变量统一预留栈空间，将esp减去相应字节数。当函数执行流程途径变量声明语句时，如有需要就调用相应构造函数将变量初始化。当执行流程即将离开声明所在代码块时，以初始化的顺序的相反顺序逐一调用析构函数。当执行流程离开函数体时，将esp加上相应字节数，归还栈空间。

为了访问函数变量，必须有方法定位每一个变量。变量相对于栈顶esp的位置在进入函数体时就已确定，但是由于esp会在函数执行期变动，所以将esp的值保存在ebp中，并事先将ebp的值压栈。随后，在函数体中通过ebp减去偏移量来访问变量。以一个最简单的函数为例：

```
void f()
{
int a=0; //a的地址被分配为ebp-4
char c=1; //c的地址被分配为ebp-8
}
```

产生的汇编代码为：

```
push ebp ;将ebp压栈
mov ebp,esp ;ebp=esp 用栈底备份栈顶指针
sub esp,8 ;esp-=8，为a和c预留空间，包括边界对齐
mov dword ptr[ebp-4],0 ;a=0
```

mov byte ptr[ebp-8],1 ;c=1

add esp,8 ;esp+=8，归还a和c的空间

mov esp,ebp ;esp=ebp 从栈底恢复栈顶指针

pop ebp ;恢复ebp

ret ;返回

相应的内存布局是这样：

09992:c=1 <-esp

09996:a=0

10000:旧ebp <-ebp

10004:……

注:汇编中的pop、push、call、ret语句是栈操作指令，其功能可以用普通指令替换

push ebp相当于:

add esp,4

mov dword ptr[esp],ebp

pop ebp相当于：

mov ebp,dword ptr[esp]

sub esp,4

call fun_address相当于：

push eip

jmp fun_address

ret相当于

add esp,4

jmp dword ptr[esp-4]

带参数的ret

ret 8相当于

add esp,12

jmp dword ptr[esp-4]

所有局部变量都在栈中由函数统一分配，形成了类似逆序数组的结构，可以通过指针逐一访问。这一特点具有很多有趣性质，比如，考虑如下函数，找出其中的错误及其造成的结果：

```
void f()
{
int i,a[10];
for(i=0;i<=10;++i)a[i]=0;/An error occurs here!
}
```

这个函数中包含的错误，即使是C++新手也很容易发现，这是老生常谈的越界访问问题。但是这个错误造成的结果，是很多人没有想到的。这次的越界访问，并不

会像很多新手预料的那样造成一个“非法操作”消息，也不会像很多老手估计的那样会默不作声，而是导致一个，呃，死循环！

错误的本质显而易见，我们访问了a[10]，但是a[10]并不存在。C++标准对于越界访问只是说“未定义操作”。我们知道，a[10]是数组a所在位置之后的一个位置，但问题是，是谁在这个位置上。是i!

根据前面的讨论，i在数组a之前被声明，所以在a之前分配在栈上。但是，I386上栈是向下增长的，所以，a的地址低于i的地址。其结果是在循环的最后，a[i]引用到了i自己！接下来的事情就不难预见了，a[i]，也就是i，被重置为0，然后继续循环的条件仍然成立……这个循环会一直继续下去，直到在你的帐单上产生高额电费，直到耗光地球电能，直到太阳停止燃烧……呵呵，或者直到聪明的你把程序Kill了……