C++ Tricks 2.3 I386 平臺C函數內部的棧分配

從 farseerfc.wordpress.com 導入

2.3 I386平臺C函數內部的棧分配

函數使用棧來保存局部變量，傳遞函數參數。進入函數時，函數在棧上爲函數中的變量統一預留棧空間，將esp減去相應字節數。當函數執行流程途徑變量聲明語句時，如有需要就調用相應構造函數將變量初始化。當執行流程即將離開聲明所在代碼塊時，以初始化的順序的相反順序逐一調用析構函數。當執行流程離開函數體時，將esp加上相應字節數，歸還棧空間。

爲了訪問函數變量，必須有方法定位每一個變量。變量相對於棧頂esp的位置在進入函數體時就已確定，但是由於esp會在函數執行期變動，所以將esp的值保存在ebp中，並事先將ebp的值壓棧。隨後，在函數體中通過ebp減去偏移量來訪問變量。以一個最簡單的函數爲例：

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

產生的彙編代碼爲：

```
push ebp ;將ebp壓棧
mov ebp,esp ;ebp=esp 用棧底備份棧頂指針
sub esp,8 ;esp-=8，爲a和c預留空間，包括邊界對齊
mov dword ptr[ebp-4],0 ;a=0
```

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

add esp,8 ;esp+=8，歸還a和c的空間

mov esp,ebp ;esp=ebp 從棧底恢復棧頂指針

pop ebp ;恢復ebp

ret ;返回

相應的內存佈局是這樣：

09992:c=1 <-esp

09996:a=0

10000:舊ebp <-ebp

10004:……

注:彙編中的pop、push、call、ret語句是棧操作指令，其功能可以用普通指令替換

push ebp相當於:

add esp,4

mov dword ptr[esp],ebp

pop ebp相當於：

mov ebp,dword ptr[esp]

sub esp,4

call fun_address相當於：

push eip

jmp fun_address

ret相當於

add esp,4

jmp dword ptr[esp-4]

帶參數的ret

ret 8相當於

add esp,12

jmp dword ptr[esp-4]

所有局部變量都在棧中由函數統一分配，形成了類似逆序數組的結構，可以通過指針逐一訪問。這一特點具有很多有趣性質，比如，考慮如下函數，找出其中的錯誤及其造成的結果：

void f()

{

int i,a[10];

for(i=0;i<=10;++i)a[i]=0;/An error occurs here!

}

這個函數中包含的錯誤，即使是C++新手也很容易發現，這是老生常談的越界訪問問題。但是這個錯誤造成的結果，是很多人沒有想到的。這次的越界訪問，並不

會像很多新手預料的那樣造成一個“非法操作”消息，也不會像很多老手估計的那樣會默不作聲，而是導致一個，呃，死循環！

錯誤的本質顯而易見，我們訪問了a[10]，但是a[10]並不存在。C++標準對於越界訪問只是說“未定義操作”。我們知道，a[10]是數組a所在位置之後的一個位置，但問題是，是誰在這個位置上。是i!

根據前面的討論，i在數組a之前被聲明，所以在a之前分配在棧上。但是，I386上棧是向下增長的，所以，a的地址低於i的地址。其結果是在循環的最後，a[i]引用到了i自己！接下來的事情就不難預見了，a[i]，也就是i，被重置爲0，然後繼續循環的條件仍然成立……這個循環會一直繼續下去，直到在你的帳單上產生高額電費，直到耗光地球電能，直到太陽停止燃燒……呵呵，或者直到聰明的你把程序Kill了……