C 語言函數調用棧(一)

https://www.cnblogs.com/clover-toeic/p/3755401.html

程序的執行過程可看作連續的函數調用。當一個函數執行完畢時，程序要回到調用指令的下一條指令(緊接 call 指令)處繼續執行。函數調用過程通常使用堆棧實現，每個用戶態進程對應一個調用棧結構(call stack)。編譯器使用堆棧傳遞函數參數、保存返回地址、臨時保存寄存器原有值(即函數調用的上下文)以備恢復以及存儲本地局部變量。

不同處理器和編譯器的堆棧佈局、函數調用方法都可能不同，但堆棧的基本概念是一樣的。

1 寄存器分配

寄存器是處理器加工數據或運行程序的重要載體，用於存放程序執行中用到的數據和指令。因此函數調用棧的實現與處理器寄存器組密切相關。

Intel 32 位體系結構(簡稱 IA32)處理器包含 8 個四字節寄存器，如下圖所示：

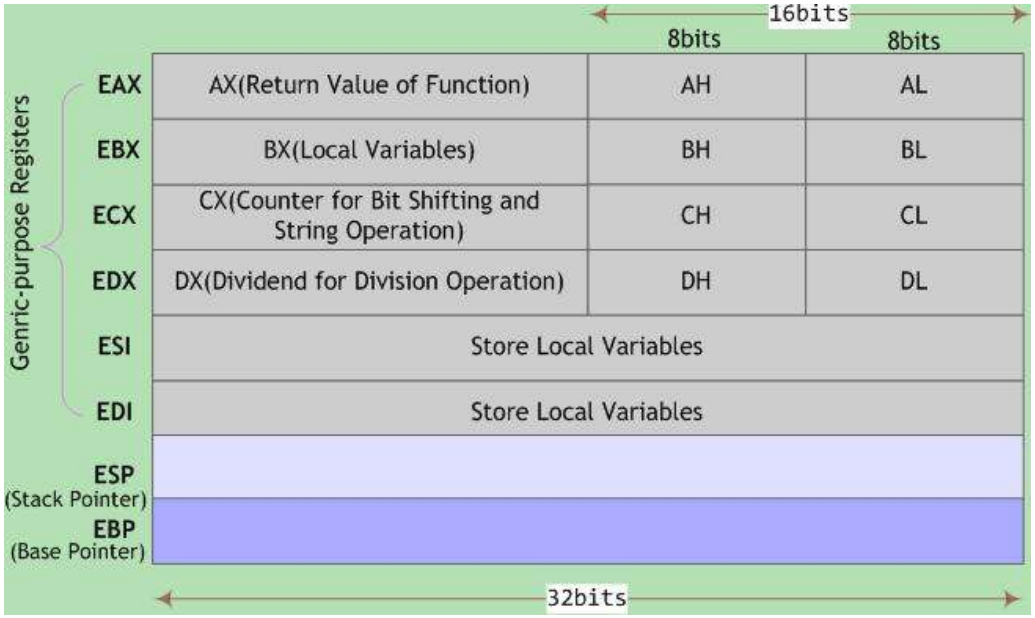

圖 1 IA32 處理器寄存器

最初的 8086 中寄存器是 16 位，每個都有特殊用途，寄存器名城反映其不同用途。由於 IA32 平台採用平面尋址模式，對特殊寄存器的需求大大降低，但由於歷史原因，這些寄存器名稱被保留下來。在大多數情況下，上圖所示的前 6 個寄存器均可作為通用寄存器使用。某些指令可能以固定的寄存器作為源寄存器或目的寄存器，如一些特殊的算術操作指令 imull/mull/cltd/idivl/divl 要求一個參數必須在%eax 中，其運算結果存放在%edx(higher 32-bit)和%eax (lower32-bit)中；又如函數返回值通常保存在%eax 中，等等。為避免兼容性問題，ABI 規範對這組通用寄存器的具體作用加以定義(如圖中所示)。

對於寄存器%eax、%ebx、%ecx 和%edx，各自可作為兩個獨立的 16 位寄存器使用，而低 16 位寄存器還可繼續分為兩個獨立的 8 位寄存器使用。編譯器會根據操作數大小選擇合適的寄存器來生成彙編代碼。在彙編語言層面，這組通用寄存器以%e(AT&T 語法)或直接以 e(Intel 語法)開頭來引用，例如 mov $5, %eax 或 mov eax, 5 表示將立即數 5 賦值給寄存器%eax。

在 x86 處理器中，EIP(Instruction Pointer)是指令寄存器，指向處理器下條等待執行的指令地址(代碼段內的偏移量)，每次執行完相應彙編指令 EIP 值就會增加。ESP(Stack Pointer)是堆棧指針寄存器，存放執行函數對應棧幀的棧頂地址(也是系統棧的頂部)，

且始終指向棧頂；EBP(Base Pointer)是棧幀基址指針寄存器，存放執行函數對應棧幀的棧底地址，用於 C 運行庫訪問棧中的局部變量和參數。

注意，EIP 是個特殊寄存器，不能像訪問通用寄存器那樣訪問它，即找不到可用來尋址 EIP 並對其進行讀寫的操作碼(OpCode)。EIP 可被 jmp、call 和 ret 等指令隱含地改變(事實上它一直都在改變)。

不同架構的 CPU，寄存器名稱被添加不同前綴以指示寄存器的大小。例如 x86 架構用字母「e(extended)」作名稱前綴，指示寄存器大小為 32 位；x86_64 架構用字母「r」作名稱前綴，指示各寄存器大小為 64 位。

編譯器在將 C 程序編譯成彙編程序時，應遵循 ABI 所規定的寄存器功能定義。同樣地，編寫彙編程序時也應遵循，否則所編寫的彙編程序可能無法與 C 程序協同工作。

【擴展閱讀】**棧幀指針寄存器**

為了訪問函數局部變量，必須能定位每個變量。局部變量相對於堆棧指針 ESP 的位置在進入函數時就已確定，理論上變量可用 ESP 加偏移量來引用，但 ESP 會在函數執行期隨變量的壓棧和出棧而變動。儘管某些情況下編譯器能跟蹤棧中的變量操作以修正偏移量，但要引入可觀的管理開銷。而且在有些機器上(如 Intel 處理器)，用 ESP 加偏移量來訪問一個變量需要多條指令才能實現。

因此，許多編譯器使用幀指針寄存器 FP(Frame Pointer)記錄棧幀基地址。局部變量和函數參數都可通過幀指針引用，因為它們到 FP 的距離不會受到壓棧和出棧操作的影響。有些資料將幀指針稱作局部基指針(LB-local base pointer)。

在 Intel CPU 中，寄存器 BP(EBP)用作幀指針。在 Motorola CPU 中，除 A7(堆棧指針 SP)外的任何地址寄存器都可用作 FP。當堆棧向下(低地址)增長時，以 FP 地址為基準，函數參數的偏移量是正值，而局部變量的偏移量是負值。

2 寄存器使用約定

程序寄存器組是唯一能被所有函數共享的資源。雖然某一時刻只有一個函數在執行，但需保證當某個函數調用其他函數時，被調函數不會修改或覆蓋主調函數稍後會使用到的寄存器值。因此，IA32 採用一套統一的寄存器使用約定，所有函數(包括庫函數)調用都必須遵守該約定。

根據慣例，寄存器%eax、%edx 和%ecx 為主調函數保存寄存器(caller-saved registers)，當函數調用時，若主調函數希望保持這些寄存器的值，則必須在調用前顯式地將其保存在棧中；被調函數可以覆蓋這些寄存器，而不會破壞主調函數所需的數據。寄存器%ebx、%esi 和%edi 為被調函數保存寄存器(callee-saved registers)，即被調函數在覆蓋這些寄存器的值時，必須先將寄存器原值壓入棧中保存起來，並在函數返回前從棧中恢復其原值，因為主調函數可能也在使用這些寄存器。此外，被調函數必須保持寄存器%ebp 和%esp，並在函數返回後將其恢復到調用前的值，亦即必須恢復主調函數的棧幀。

當然，這些工作都由編譯器在幕後進行。不過在編寫彙編程序時應注意遵守上述慣例。

3 棧幀結構

函數調用經常是嵌套的，在同一時刻，堆棧中會有多個函數的信息。每個未完成運行的函數佔用一個獨立的連續區域，稱作棧幀(Stack Frame)。棧幀是堆棧的邏輯片段，當調用函數時邏輯棧幀被壓入堆棧，當函數返回時邏輯棧幀被從堆棧中彈出。棧幀存放著函數參數，局部變量及恢復前一棧幀所需要的數據等。

編譯器利用棧幀，使得函數參數和函數中局部變量的分配與釋放對程序員透明。編譯器將控制權移交函數本身之前，插入特定代碼將函數參數壓入棧幀中，並分配足夠的內存空間用於存放函數中的局部變量。使用棧幀的一個好處是使得遞歸變為可能，因為對函數的每次遞歸調用，都會分配給該函數一個新的棧幀，這樣就巧妙地隔離當前調用與上次調用。

棧幀的邊界由棧幀基地址指針 EBP 和堆棧指針 ESP 界定(指針存放在相應寄存器中)。EBP 指向當前棧幀底部(高地址)，在當前棧幀內位置固定；ESP 指向當前棧幀頂部(低地址)，當程序執行時 ESP 會隨著數據的入棧和出棧而移動。因此函數中對大部分數據的訪問都基於 EBP 進行。

為更具描述性，以下稱 EBP 為幀基指針， ESP 為棧頂指針，並在引用彙編代碼時分別記為%ebp 和%esp。

函數調用棧的典型內存佈局如下圖所示：

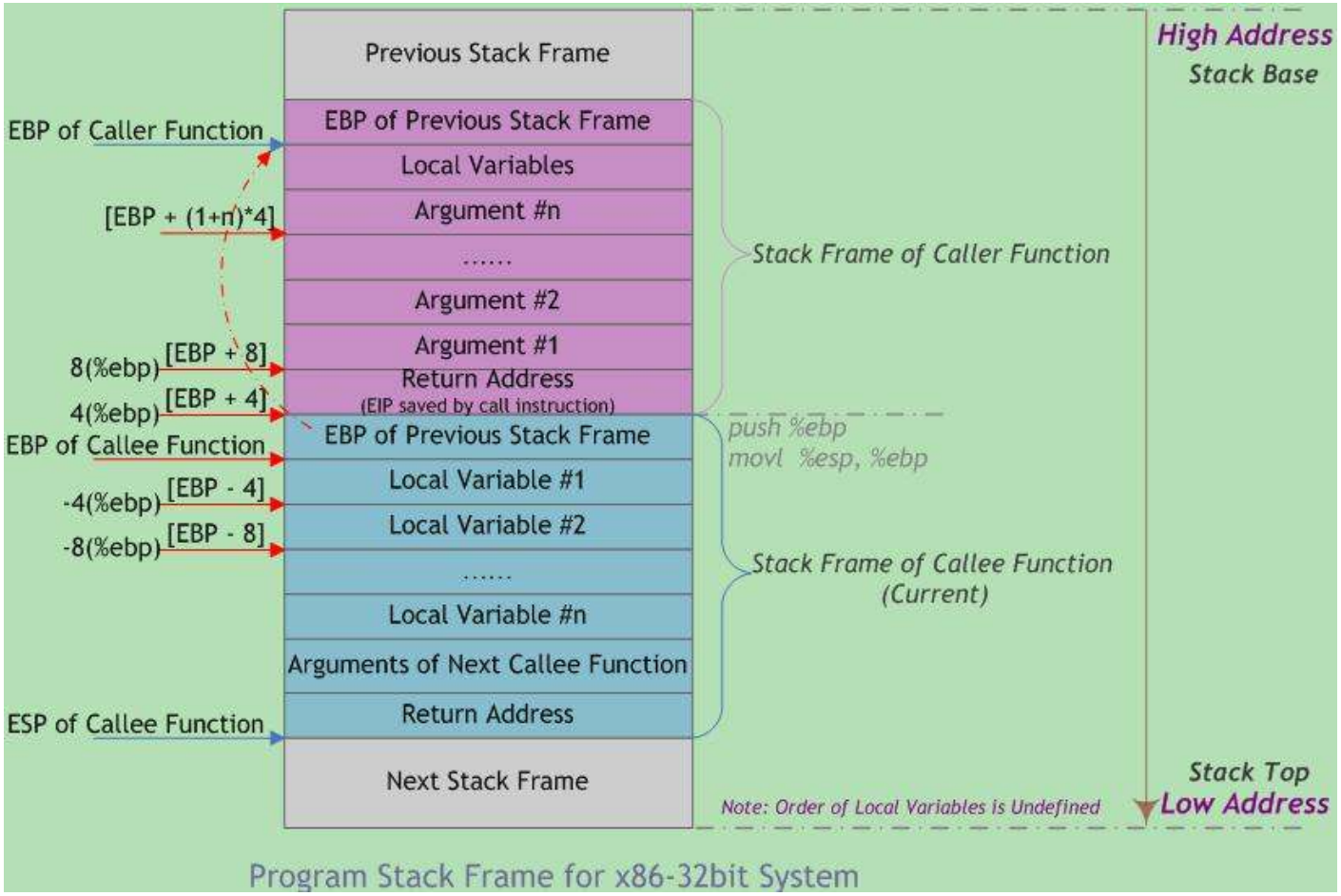

圖 2 函數調用棧的典型內存佈局

圖中給出主調函數(caller)和被調函數(callee)的棧幀佈局，"m(%ebp)"表示以 EBP 為基地址、偏移量為 m 字節的內存空間(中的內容)。該圖基於兩個假設：第一，函數返回值不是結構體或聯合體，否則第一個參數將位於"12(%ebp)" 處；第二，每個參數都是 4 字節大小(棧的粒度為 4 字節)。在本文後續章節將就參數的傳遞和大小問題做進一步的探討。　此外，函數可以沒有參數和局部變量，故圖中「Argument(參數)」和「Local Variable(局部變量)」不是函數棧幀結構的必需部分。

從圖中可以看出，函數調用時入棧順序為

實參 N~1→主調函數返回地址→主調函數幀基指針 EBP→被調函數局部變量 1~N

其中，主調函數將參數按照調用約定依次入棧(圖中為從右到左)，然後將指令指針 EIP 入棧以保存主調函數的返回地址(下一條待執行指令的地址)。進入被調函數時，被調函數將主調函數的幀基指針 EBP 入棧，並將主調函數的棧頂指針 ESP 值賦給被調函數的 EBP(作為被調函數的棧底)，接著改變 ESP 值來為函數局部變量預留空間。此時被調函數幀基指針指向被調函數的棧底。以該地址為基準，向上(棧底方向)可獲取主調函數的返回地址、參數值，向下(棧頂方向)能獲取被調函數的局部變量值，而該地址處又存放著上一層主調函數的幀基指針值。本級調用結束後，將 EBP 指針值賦給 ESP，使 ESP 再次指向被調函數棧底以釋放局部變量；再將已壓棧的主調函數幀基指針彈出到 EBP，並彈出返回地址到 EIP。ESP 繼續上移越過參數，最終回到函數調用前的狀態，即恢復原來主調函數的棧幀。如此遞歸便形成函數調用棧。

EBP 指針在當前函數運行過程中(未調用其他函數時)保持不變。在函數調用前，ESP 指針指向棧頂地址，也是棧底地址。在函數完成現場保護之類的初始化工作後，ESP 會始終指向當前函數棧幀的棧頂，此時，若當前函數又調用另一個函數，則會將此時的 EBP 視為舊 EBP 壓棧，而與新調用函數有關的內容會從當前 ESP 所指向位置開始壓棧。

若需在函數中保存被調函數保存寄存器(如 ESI、EDI)，則編譯器在保存 EBP 值時進行保存，或延遲保存直到局部變量空間被分配。在棧幀中並未為被調函數保存寄存器的空間指定標準的存儲位置。包含寄存器和臨時變量的函數調用棧佈局可能如下圖所示：

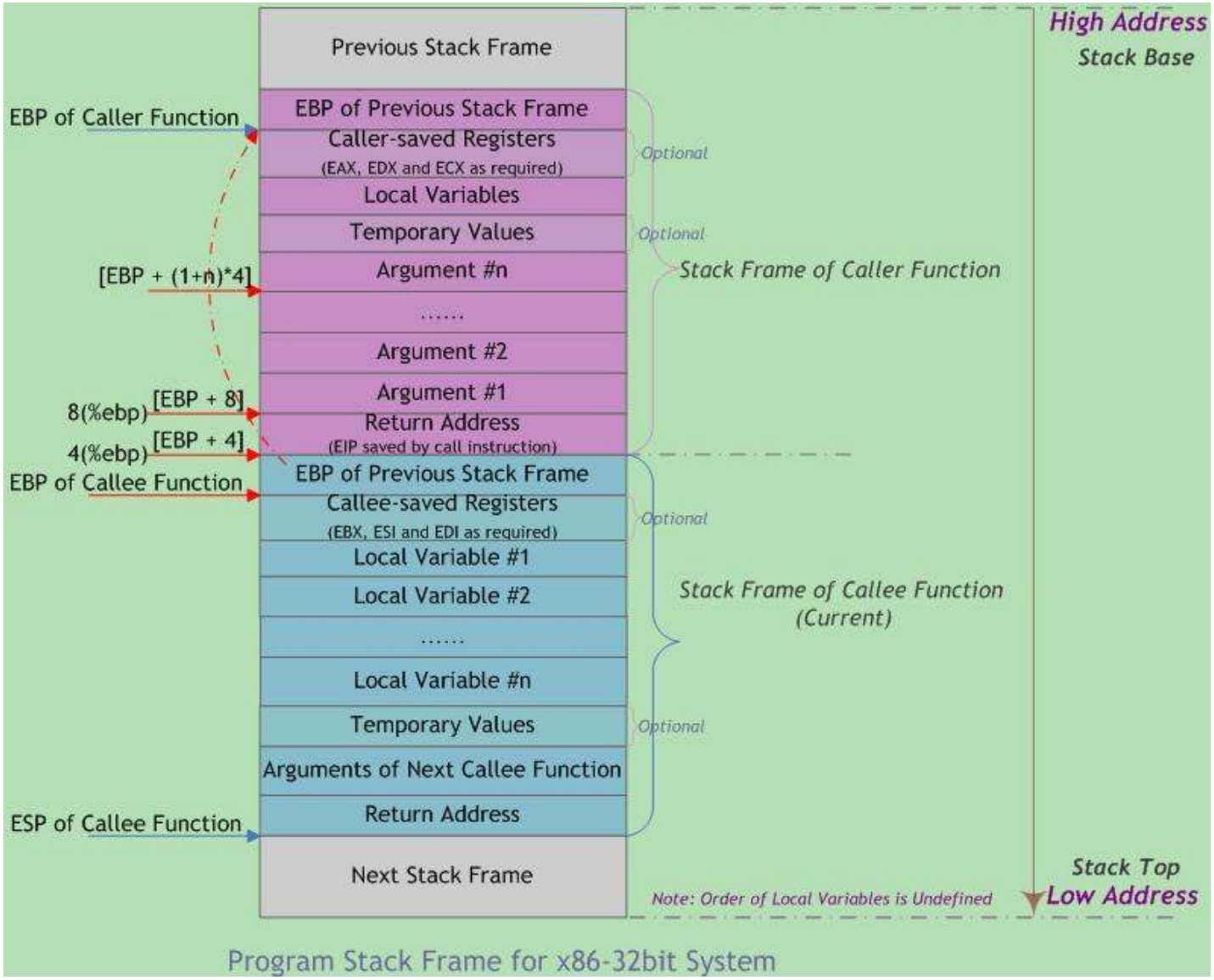

圖 3 函數調用棧的可能內存佈局

在多線程(任務)環境，棧頂指針指向的存儲器區域就是當前使用的堆棧。切換線程的一個重要工作，就是將棧頂指針設為當前線程的堆棧棧頂地址。

以下代碼用於函數棧佈局示例：

```
 1 //StackFrame.c
 2 #include <stdio.h>
 3 #include <string.h>
 4
 5 struct Strt{
 6     int member1;
 7     int member2;
 8     int member3;
 9 };
10
11 #define PRINT_ADDR(x)     printf("&"#x" = %p\n", &x)
12 int StackFrameContent(int para1, int para2, int para3){
13     int locVar1 = 1;
14     int locVar2 = 2;
15     int locVar3 = 3;
16     int arr[] = {0x11,0x22,0x33};
17     struct Strt tStrt = {0};
18     PRINT_ADDR(para1); //若 para1 為 char 或 short 型，則打印 para1 所對應的棧上整型臨時變量地址！
19     PRINT_ADDR(para2);
20     PRINT_ADDR(para3);
21     PRINT_ADDR(locVar1);
22     PRINT_ADDR(locVar2);
```

```
23     PRINT_ADDR(locVar3);
24     PRINT_ADDR(arr);
25     PRINT_ADDR(arr[0]);
26     PRINT_ADDR(arr[1]);
27     PRINT_ADDR(arr[2]);
28     PRINT_ADDR(tStrt);
29     PRINT_ADDR(tStrt.member1);
30     PRINT_ADDR(tStrt.member2);
31     PRINT_ADDR(tStrt.member3);
32     return 0;
33 }
34
35 int main(void){
36     int locMain1 = 1, locMain2 = 2, locMain3 = 3;
37     PRINT_ADDR(locMain1);
38     PRINT_ADDR(locMain2);
39     PRINT_ADDR(locMain3);
40     StackFrameContent(locMain1, locMain2, locMain3);
41     printf("[locMain1,2,3] = [%d, %d, %d]\n", locMain1, locMain2, locMain3);
42     memset(&locMain2, 0, 2*sizeof(int));
43     printf("[locMain1,2,3] = [%d, %d, %d]\n", locMain1, locMain2, locMain3);
44     return 0;
45 }
```

編譯鏈接並執行後，輸出打印如下：

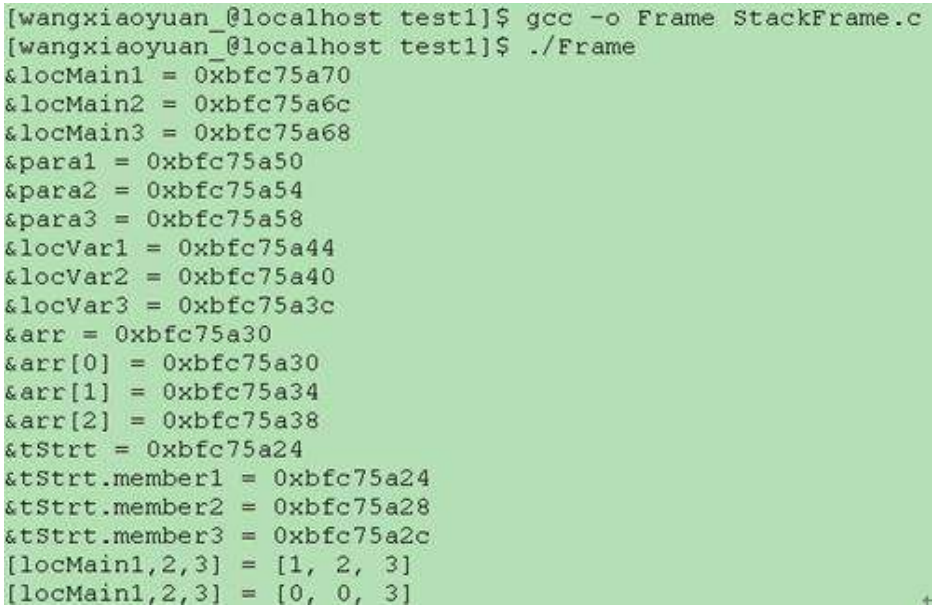

圖 4 StackFrame 輸出

函數棧佈局示例如下圖所示。為直觀起見，低於起始高地址 0xbfc75a58 的其他地址採用點記法，如 0x.54 表示 0xbfc75a54，以此類推。

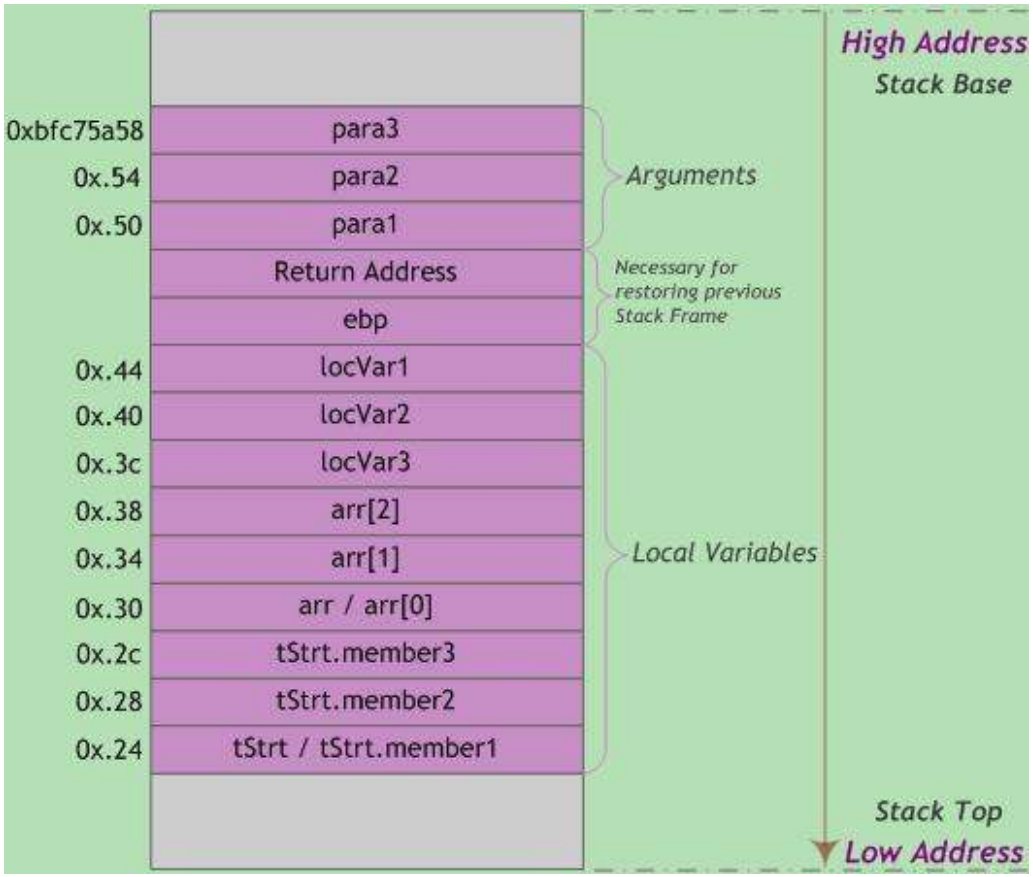

圖 5 StackFrame 棧幀

內存地址從棧底到棧頂遞減，壓棧就是把 ESP 指針逐漸往地低址移動的過程。而結構體 tStrt 中的成員變量 memberX 地址=tStrt 首地址+(memberX 偏移量)，即越靠近 tStrt 首地址的成員變量其內存地址越小。因此，結構體成員變量的入棧順序與其在結構體中聲明的順序相反。

函數調用以值傳遞時，傳入的實參(locMain1~3)與被調函數內操作的形參(para1~3)兩者存儲地址不同，因此被調函數無法直接修改主調函數實參值(對形參的操作相當於修改實參的副本)。為達到修改目的，需要向被調函數傳遞實參變量的指針(即變量的地址)。

此外，"[locMain1,2,3] = [0, 0, 3]"是因為對四字節參數 locMain2 調用 memset 函數時，會從低地址向高地址連續清零 8 個字節，從而誤將位於高地址 locMain1 清零。

注意，局部變量的佈局依賴於編譯器實現等因素。因此，當 StackFrameContent 函數中刪除打印語句時，變量 locVar3、locVar2 和 locVar1 可能按照從高到低的順序依次存儲！而且，局部變量並不總在棧中，有時出於性能(速度)考慮會存放在寄存器中。數組/結構體型的局部變量通常分配在棧內存中。

【擴展閱讀】函數局部變量佈局方式

與函數調用約定規定參數如何傳入不同，局部變量以何種方式佈局並未規定。編譯器計算函數局部變量所需要的空間總數，並確定這些變量存儲在寄存器上還是分配在程序棧上(甚至被優化掉)—某些處理器並沒有堆棧。局部變量的空間分配與主調函數和被調函數無關，僅僅從函數源代碼上無法確定該函數的局部變量分佈情況。

基於不同的編譯器版本(gcc3.4 中局部變量按照定義順序依次入棧，gcc4 及以上版本則不定)、優化級別、目標處理器架構、棧安全性等，相鄰定義的兩個變量在內存位置上可能相鄰，也可能不相鄰，前後關係也不固定。若要確保兩個對象在內存上相鄰且前後關係固定，可使用結構體或數組定義。

4 堆棧操作

函數調用時的具體步驟如下：

1) 主調函數將被調函數所要求的參數，根據相應的函數調用約定，保存在運行時棧中。該操作會改變程序的棧指針。

註：x86 平台將參數壓入調用棧中。而 x86_64 平台具有 16 個通用 64 位寄存器，故調用函數時前 6 個參數通常由寄存器傳遞，其餘參數才通過棧傳遞。

2) 主調函數將控制權移交給被調函數(使用 call 指令)。函數的返回地址(待執行的下條指令地址)保存在程序棧中(壓棧操作隱含在 call 指令中)。

3）若有必要，被調函數會設置幀基指針，並保存被調函數希望保持不變的寄存器值。

4）被調函數通過修改棧頂指針的值，為自己的局部變量在運行時棧中分配內存空間，並從幀基指針的位置處向低地址方向存放被調函數的局部變量和臨時變量。

5）被調函數執行自己任務，此時可能需要訪問由主調函數傳入的參數。若被調函數返回一個值，該值通常保存在一個指定寄存器中(如EAX)。

6）一旦被調函數完成操作，為該函數局部變量分配的棧空間將被釋放。這通常是步驟 4 的逆向執行。

7）恢復步驟 3 中保存的寄存器值，包含主調函數的幀基指針寄存器。

8）被調函數將控制權交還主調函數(使用 ret 指令)。根據使用的函數調用約定，該操作也可能從程序棧上清除先前傳入的參數。

9）主調函數再次獲得控制權後，可能需要將先前的參數從棧上清除。在這種情況下，對棧的修改需要將幀基指針值恢復到步驟 1 之前的值。

步驟 3 與步驟 4 在函數調用之初常一同出現，統稱為函數序(prologue)；步驟 6 到步驟 8 在函數調用的最後常一同出現，統稱為函數跋(epilogue)。函數序和函數跋是編譯器自動添加的開始和結束彙編代碼，其實現與 CPU 架構和編譯器相關。除步驟 5 代表函數實體外，其它所有操作組成函數調用。

以下介紹函數調用過程中的主要指令。

壓棧(push)：棧頂指針 ESP 減小 4 個字節；以字節為單位將寄存器數據(四字節，不足補零)壓入堆棧，從高到低按字節依次將數據存入 ESP-1、ESP-2、ESP-3、ESP-4 指向的地址單元。

出棧(pop)：棧頂指針 ESP 指向的棧中數據被取回到寄存器；棧頂指針 ESP 增加 4 個字節。

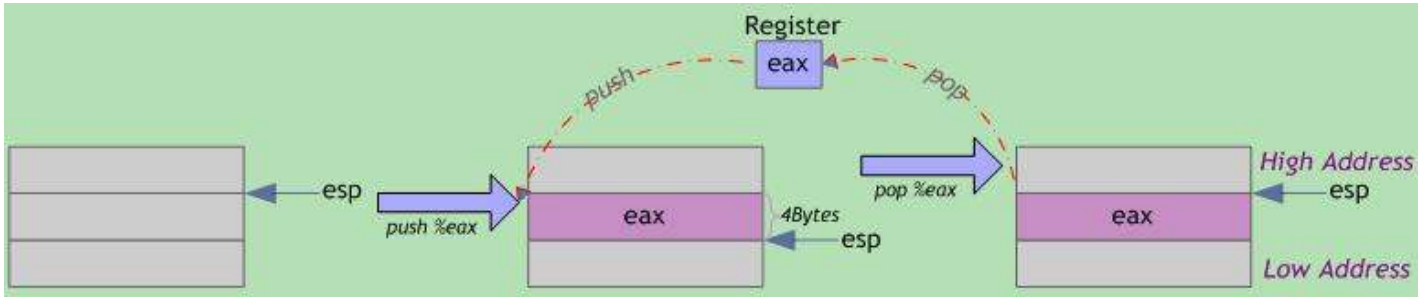

圖 6 出棧入棧操作示意

可見，壓棧操作將寄存器內容存入棧內存中(寄存器原內容不變)，棧頂地址減小；出棧操作從棧內存中取回寄存器內容(棧內已存數據不會自動清零)，棧頂地址增大。棧頂指針 ESP 總是指向棧中下一個可用數據。

調用(call)：將當前的指令指針 EIP(該指針指向緊接在 call 指令後的下條指令)壓入堆棧，以備返回時能恢復執行下條指令；然後設置 EIP 指向被調函數代碼開始處，以跳轉到被調函數的入口地址執行。

離開(leave)： 恢復主調函數的棧幀以準備返回。等價於指令序列 movl %ebp, %esp(恢復原 ESP 值，指向被調函數棧幀開始處)和 popl %ebp(恢復原 ebp 的值，即主調函數幀基指針)。

返回(ret)：與 call 指令配合，用於從函數或過程返回。從棧頂彈出返回地址(之前 call 指令保存的下條指令地址)到 EIP 寄存器中，程序轉到該地址處繼續執行(此時 ESP 指向進入函數時的第一個參數)。若帶立即數，ESP 再加立即數(丟棄一些在執行 call 前入棧的參數)。使用該指令前，應使當前棧頂指針所指向位置的內容正好是先前 call 指令保存的返回地址。

基於以上指令，使用 C 調用約定的被調函數典型的函數序和函數跋實現如下：

	指令序列	含義
函數序 (prologue)	push %ebp	將主調函數的幀基指針%ebp 壓棧，即保存舊棧幀中的幀基指針以便函數返回時恢復舊棧幀
	mov %esp, %ebp	將主調函數的棧頂指針%esp 賦給被調函數幀基指針%ebp。此

		時，%ebp 指向被調函數新棧幀的起始地址(棧底)，亦即舊%ebp 入棧後的棧頂
	sub <n>, %esp	將棧頂指針%esp 減去指定字節數(棧頂下移)，即為被調函數局部變量開闢棧空間。<n>為立即數且通常為 16 的整數倍(可能大於局部變量字節總數而稍顯浪費，但 gcc 採用該規則保證數據的嚴格對齊以有效運用各種優化編譯技術)
	push <r>	可選。如有必要，被調函數負責保存某些寄存器(%edi/%esi/%ebx)值
函數跋 (epilogue)	pop <r>	可選。如有必要，被調函數負責恢復某些寄存器(%edi/%esi/%ebx)值
	mov %ebp, %esp*	恢復主調函數的棧頂指針%esp，將其指向被調函數棧底。此時，局部變量佔用的棧空間被釋放，但變量內容未被清除(跳過該處理)
	pop %ebp*	主調函數的幀基指針%ebp 出棧，即恢復主調函數棧底。此時，棧頂指針%esp 指向主調函數棧頂(espßesp-4)，亦即返回地址存放處
	ret	從棧頂彈出主調函數壓在棧中的返回地址到指令指針寄存器%eip 中，跳回主調函數該位置處繼續執行。再由主調函數恢復到調用前的棧
*：這兩條指令序列也可由 leave 指令實現，具體用哪種方式由編譯器決定。		

若主調函數和調函數均未使用局部變量寄存器 EDI、ESI 和 EBX，則編譯器無須在函數序中對其壓棧，以便提高程序的執行效率。

參數壓棧指令因編譯器而異，如下兩種壓棧方式基本等效：

```
extern CdeclDemo(int w, int x, int y, intz);  //調用 CdeclDemo 函數
```

CdeclDemo(1, 2, 3, 4); //調用 CdeclDemo 函數	
壓棧方式一	**壓棧方式二**

pushl 4 //壓入參數 z
 	
pushl 3 //壓入參數 y
 	
pushl 2 //壓入參數 x
 	
pushl 1 //壓入參數 w
 	
call CdeclDemo //調用函數
 	
addl $16, %esp //恢復 ESP 原值，使其指向調用前保存的返回地址
	subl $16, %esp //多次調用僅執行一遍
	
	movl $4, 12(%esp) //傳送參數 z 至堆棧第四個位置
	
	movl $3, 8(%esp) //傳送參數 y 至堆棧第三個位置
	
	movl $2, 4(%esp) //傳送參數 x 至堆棧第二個位置
	
	movl $1, (%esp) //傳送參數 w 至堆棧棧頂
	
	call CdeclDemo //調用函數

兩種壓棧方式均遵循 C 調用約定，但方式二中主調函數在調用返回後並未顯式清理堆棧空間。因為在被調函數序階段，編譯器在棧頂為函數參數預先分配內存空間(sub 指令)。函數參數被覆制到棧中(而非壓入棧中)，並未修改棧頂指針，故調用返回時主調函數也無需修改棧頂指針。gcc3.4(或更高版本)編譯器採用該技術將函數參數傳遞至棧上，相比棧頂指針隨每次參數壓棧而多次下移，一次性設置好棧頂指針更為高效。設想連續調用多個函數時，方式二僅需預先分配一次參數內存(大小足夠容納參數尺寸和最大的函數即可)，後續調用無需每次都恢復棧頂指針。注意，函數被調用時，兩種方式均使棧頂指針指向函數最左邊的參數。本文不再區分兩種壓棧方式，"壓棧"或"入棧"所提之處均按相應彙編代碼理解，若無彙編則指方式二。

某些情況下，編譯器生成的函數調用進入/退出指令序列並不按照以上方式進行。例如，若 C 函數聲明為 static(只在本編譯單元內可見)且函數在編譯單元內被直接調用，未被顯示或隱式取地址(即沒有任何函數指針指向該函數)，此時編譯器確信該函數不會被其它編譯單元調用，因此可隨意修改其進/出指令序列以達到優化目的。

儘管使用的寄存器名字和指令在不同處理器架構上有所不同，但創建棧幀的基本過程一致。

注意，棧幀是運行時概念，若程序不運行，就不存在棧和棧幀。但通過分析目標文件中建立函數棧幀的彙編代碼(尤其是函數序和函數跋過程)，即使函數沒有運行，也能瞭解函數的棧幀結構。通過分析可確定分配在函數棧幀上的局部變量空間準確值，函數中是否使用幀基指針，以及識別函數棧幀中對變量的所有內存引用。

C 語言函數調用棧(二)

https://www.cnblogs.com/clover-toeic/p/3756668.html

5 函數調用約定

創建一個棧幀的最重要步驟是主調函數如何向棧中傳遞函數參數。主調函數必須精確存儲這些參數，以便被調函數能夠訪問到它們。函數通過選擇特定的調用約定，來表明其希望以特定方式接收參數。此外，當被調函數完成任務後，調用約定規定先前入棧的參數由主調函數還是被調函數負責清除，以保證程序的棧頂指針完整性。

函數調用約定通常規定如下幾方面內容：

1) 函數參數的傳遞順序和方式

最常見的參數傳遞方式是通過堆棧傳遞。主調函數將參數壓入棧中，被調函數以相對於幀基指針的正偏移量來訪問棧中的參數。對於有多個參數的函數，調用約定需規定主調函數將參數壓棧的順序(從左至右還是從右至左)。某些調用約定允許使用寄存器傳參以提高性能。

2) 棧的維護方式

主調函數將參數壓棧後調用被調函數體，返回時需將被壓棧的參數全部彈出，以便將棧恢復到調用前的狀態。該清棧過程可由主調函數負責完成，也可由被調函數負責完成。

3) 名字修飾(Name-mangling)策略

又稱函數名修飾(Decorated Name)規則。編譯器在鏈接時為區分不同函數，對函數名作不同修飾。

若函數之間的調用約定不匹配，可能會產生堆棧異常或鏈接錯誤等問題。因此，為了保證程序能正確執行，所有的函數調用均應遵守一致的調用約定。

5.1 常見調用約定

下面分別介紹常見的幾種函數調用約定。

1. cdecl 調用約定

又稱 C 調用約定，是 C/C++編譯器默認的函數調用約定。所有非 C++成員函數和未使用 stdcall 或 fastcall 聲明的函數都默認是 cdecl 方式。函數參數按照從右到左的順序入棧，函數調用者負責清除棧中的參數，返回值在 EAX 中。由於每次函數調用都要產生清除(還原)堆棧的代碼，故使用 cdecl 方式編譯的程序比使用 stdcall 方式編譯的程序大(後者僅需在被調函數內產生一份清棧代碼)。但 cdecl 調用方式支持可變參數函數(即函數帶有可變數目的參數，如 printf)，且調用時即使實參和形參數目不符也不會導致堆棧錯誤。對於 C 函數，cdecl 方式的名字修飾約定是在函數名前添加一個下劃線；對於 C++函數，除非特別使用 extern "C"，C++函數使用不同的名字修飾方式。

【擴展閱讀】**可變參數函數支持條件**

若要支持可變參數的函數，則參數應自右向左進棧，並且由主調函數負責清除棧中的參數(參數出棧)。

首先，參數按照從右向左的順序壓棧，則參數列表最左邊(第一個)的參數最接近棧頂位置。所有參數距離幀基指針的偏移量都是常數，而不必關心已入棧的參數數目。只要不定的參數的數目能根據第一個已明確的參數確定，就可使用不定參數。例如 printf 函數，第一個參數即格式化字符串可作為後繼參數指示符。通過它們就可得到後續參數的類型和個數，進而知道所有參數的尺寸。當傳遞的參數過多時，以幀基指針為基準，獲取適當數目的參數，其他忽略即可。若函數參數自左向右進棧，則第一個參數距離棧幀指針的偏移量與已入棧的參數數目有關，需要計算所有參數佔用的空間後才能精確定位。當實際傳入的參數數目與函數期望接受的參數數目不同時，偏移量計算會出錯！

其次，調用函數將參數壓棧，只有它才知道棧中的參數數目和尺寸，因此調用函數可安全地清棧。而被調函數永遠也不能事先知道將要傳入函數的參數信息，難以對棧頂指針進行調整。

C++為兼容 C，仍然支持函數帶有可變的參數。但在 C++中更好的選擇常常是函數多態。

2．stdcall 調用約定(微軟命名)

Pascal 程序缺省調用方式，WinAPI 也多採用該調用約定。stdcall 調用約定主調函數參數從右向左入棧，除指針或引用類型參數外所有參數採用傳值方式傳遞，由被調函數負責清除棧中的參數，返回值在 EAX 中。stdcall 調用約定僅適用於參數個數固定的函數，因為被調函數清棧時無法精確獲知棧上有多少函數參數；而且如果調用時實參和形參數目不符會導致堆棧錯誤。對於 C 函數，stdcall 名稱修飾方式是在函數名字前添加下劃線，在函數名字後添加@和函數參數的大小，如_functionname@number。

3．fastcall 調用約定

stdcall 調用約定的變形，通常使用 ECX 和 EDX 寄存器傳遞前兩個 DWORD(四字節雙字)類型或更少字節的函數參數，其餘參數按照從右向左的順序入棧，被調函數在返回前負責清除棧中的參數，返回值在 EAX 中。因為並不是所有的參數都有壓棧操作，所以比 stdcall 和 cdecl 快些。編譯器使用兩個@修飾函數名字，後跟十進制數表示的函數參數列表大小(字節數)，如@function_name@number。需注意 fastcall 函數調用約定在不同編譯器上可能有不同的實現，比如 16 位編譯器和 32 位編譯器。另外，在使用內嵌彙編代碼時，還應注意不能和編譯器使用的寄存器有衝突。

4．thiscall 調用約定

C++類中的非靜態函數必須接收一個指向主調對象的類指針(this 指針)，並可能較頻繁的使用該指針。主調函數的對象地址必須由調用者提供，並在調用對象非靜態成員函數時將對象指針以參數形式傳遞給被調函數。編譯器默認使用 thiscall 調用約定以高效傳遞和存儲 C++類的非靜態成員函數的 this 指針參數。

thiscall 調用約定函數參數按照從右向左的順序入棧。若參數數目固定，則類實例的 this 指針通過 ECX 寄存器傳遞給被調函數，被調函數自身清理堆棧；若參數數目不定，則 this 指針在所有參數入棧後再入棧，主調函數清理堆棧。thiscall 不是 C++關鍵字，故不能使用 thiscall 聲明函數，它只能由編譯器使用。

注意，該調用約定特點隨編譯器不同而不同，g++中 thiscall 與 cdecl 基本相同，只是隱式地將 this 指針當作非靜態成員函數的第 1 個參數，主調函數在調用返回後負責清理棧上參數；而在 VC 中，this 指針存放在%ecx 寄存器中，參數從右至左壓棧，非靜態成員函數負責清理棧上參數。

5. naked call 調用約定

對於使用 naked call 方式聲明的函數，編譯器不產生保存(prologue)和恢復(epilogue)寄存器的代碼，且不能用 return 返回返回值(只能用內嵌彙編返回結果)，故稱 naked call。該調用約定用於一些特殊場合，如聲明處於非 C/C++上下文中的函數，並由程序員自行編寫初始化和清棧的內嵌彙編指令。注意，naked call 並非類型修飾符，故該調用約定必須與__declspec 同時使用，如 VC 下定義求和函數：

代碼示例如下(Windows 採用 Intel 彙編語法，註釋符為;)：

```
1 __declspec(naked) int __stdcall function(int a, int b) {
2     ;mov DestRegister, SrcImmediate(Intel) vs. movl $SrcImmediate, %DestRegister(AT&T)
3     __asm mov eax, a
4     __asm add eax, b
5     __asm ret 8
6 }
```

注意，__declspec 是微軟關鍵字，其他系統上可能沒有。

6. pascal 調用約定

Pascal 語言調用約定，參數按照從左至右的順序入棧。Pascal 語言只支持固定參數的函數，參數的類型和數量完全可知，故由被調函數自身清理堆棧。pascal 調用約定輸出的函數名稱無任何修飾且全部大寫。

Win3.X(16 位)時支持真正的 pascal 調用約定；而 Win9.X(32 位)以後 pascal 約定由 stdcall 約定代替(以 C 約定壓棧以 Pascal 約定清棧)。

上述調用約定的主要特點如下表所示：

調用方式	stdcall(Win32)	cdecl	fastcall	thiscall(C++)	naked call
參數壓棧順序	從右至左	從右至左	從右至左，Arg1 在 ecx，Arg2 在 edx	從右至左，this 指針在 ecx	自定義
參數位置	棧	棧	棧 + 寄存器	棧，寄存器 ecx	自定義
負責清棧的函數	被調函數	主調函數	被調函數	被調函數	自定義
支持可變參數	否	是	否	否	自定義
函數名字格式	_name@number	_name	@name@number		自定義
參數表開始標識	"@@YG"	"@@YA"	"@@YI"		自定義
註：C++因支撐函數重載、命名空間和成員函數等語法特徵，採用更為複雜的名字修飾策略。 C++函數修飾名以"?"開始，後面緊跟函數名、參數表開始標識和按照類型代號拼出的返回值參數表。 例如，函數 int Function(char *var1,unsigned long)對應的 stdcall 修飾名為"?Function@@YGHPADK@Z"。					

Windows 下可直接在函數聲明前添加關鍵字__stdcall、__cdecl 或__fastcall 等標識確定函數的調用方式，如 int __stdcall func()。Linux 下可借用函數 attribute 機制，如 int __attribute__((__stdcall__)) func()。

代碼示例如下：

```
1 int __attribute__((__cdecl__)) CalleeFunc(int i, int j, int k){
2 // int __attribute__((__stdcall__)) CalleeFunc(int i, int j, int k){
3 //int __attribute__((__fastcall__)) CalleeFunc(int i, int j, int k){
4     return i+j+k;
5 }
6 void CallerFunc(void){
7     CalleeFunc(0x11, 0x22, 0x33);
8 }
9 int main(void){
```

```
10     CallerFunc();
11     return 0;
12 }
```

被調函數 CalleeFunc 分別聲明為 cdecl、stdcall 和 fastcall 約定時，其彙編代碼比較如下表所示：

	cdecl	**stdcall**	**fastcall**

主調函數職責
 	
Caller
	sub $0xc,%esp
	
	movl $0x33,0x8(%esp)
	
	movl $0x22,0x4(%esp)
	
	movl $0x11,(%esp)
	
	call 8048354 <CalleeFunc>
		sub $0xc,%esp
		
		movl $0x33,0x8(%esp)
		
		movl $0x22,0x4(%esp)
		
		movl $0x11,(%esp)
		
		call 8048354 <CalleeFunc>
		
		sub $0xc,%esp
			sub $0x4,%esp
			
			movl $0x33,(%esp)
			
			mov $0x22,%edx
			
			mov $0x11,%ecx
			
			call 8048354 <CalleeFunc>
			
			sub $0x4,%esp

被調函數職責
 	
Callee
	push %ebp
	
	mov %esp,%ebp
	
	mov 0xc(%ebp),%eax
	
	add 0x8(%ebp),%eax
	
	add 0x10(%ebp),%eax
		push %ebp
		
		mov %esp,%ebp
		
		mov 0xc(%ebp),%eax
		
		add 0x8(%ebp),%eax
		
		add 0x10(%ebp),%eax
			push %ebp
			
			mov %esp,%ebp
			
			sub $0x8,%esp
			
			mov %ecx,0xfffffffc(%ebp)
			
			mov %edx,0xfffffff8(%ebp)

	pop %ebp
	
	ret
		pop %ebp
		
		ret $0xc
		
		//執行 ret 指令並清理參數佔用的堆棧(棧頂指針上移參數個數*4=12個字節，以釋放壓棧的參數)
			mov 0xffffffff8(%ebp),%eax
			
			add 0xfffffffc(%ebp),%eax
			
			add 0x8(%ebp),%eax
			
			leave
			
			ret $0x4
			
			//ret <壓棧參數字節數>。若參數不超過兩個，則 ret 指令不帶立即數，因為無參數被壓棧

5.2 調用約定影響

當函數導出被其他程序員所使用(如庫函數)時，該函數應遵循主要的調用約定，以便於程序員使用。若函數僅供內部使用，則其調用約定可只被使用該函數的程序所瞭解。

在多語言混合編程(包括 A 語言中使用 B 語言開發的第三方庫)時，若函數的原型聲明和函數體定義不一致或調用函數時聲明了不同的函數約定，將可能導致嚴重問題(如堆棧被破壞)。

以 Delphi 調用 C 函數為例。Delphi 函數缺省採用 stdcall 調用約定，而 C 函數缺省採用 cdecl 調用約定。一般將 C 函數聲明為 stdcall 約定，如：`int __stdcall add(int a, int b);`

在 Delphi 中調用該函數時也應聲明為 stdcall 約定：

```
1 function add(a: Integer; b: Integer): Integer; stdcall; //參數類型應與 DLL 中的函數或過程參數類型一致，且引用時使用 stdcall 參數
2 external 'a.dll'; //指定被調 DLL 文件的路徑和名稱
```

不同編譯器產生棧幀的方式不盡相同，主調函數不一定能正常完成清棧工作；而被調函數必然能自己完成正常清棧，因此，在跨(開發)平台調用中，通常使用 stdcall 調用約定(不少 WinApi 均採用該約定)。

此外，主調函數和被調函數所在模塊採用相同的調用約定，但分別使用 C++和 C 語法編譯時，會出現鏈接錯誤(報告被調函數未定義)。這是因為兩種語言的函數名字修飾規則不同，解決方式是使用 extern "C"告知主調函數所在模塊：被調函數是 C 語言編譯的。採用 C 語言編譯的庫應考慮到使用該庫的程序可能是 C++程序(使用 C++編譯器)，通常應這樣聲明頭文件：

```
1 #ifdef _cplusplus
2     extern "C" {
3 #endif
4     type Func(type para);
5 #ifdef _cplusplus
6     }
7 #endif
```

這樣 C++編譯器就會按照 C 語言修飾策略鏈接 Func 函數名，而不會出現找不到函數的鏈接錯誤。

5.3 x86 函數參數傳遞方法

x86 處理器 ABI 規範中規定，所有傳遞給被調函數的參數都通過堆棧來完成，其壓棧順序是以函數參數從右到左的順序。當向被調函數傳遞參數時，所有參數最後形成一個數組。由於採用從右到左的壓棧順序，數組中參數的順序(下標 0~N-1)與函數參數聲明順序(Para1~N)一致。因此，在函數中若知道第一個參數地址和各參數佔用字節數，就可通過訪問數組的方式去訪問每個參數。

5.3.1 整型和指針參數的傳遞

整型參數與指針參數的傳遞方式相同，因為在 32 位 x86 處理器上整型與指針大小相同(均為四字節)。下表給出這兩種類型的參數在棧幀中的位置關係。注意，該表基於 tail 函數的棧幀。

調用語句	參數	棧幀地址
tail(1, 2, 3, (void *)0);	1	8(%ebp)
	2	12(%ebp)
	3	16(%ebp)
	(void *)0	20(%ebp)

5.3.2 浮點參數的傳遞

浮點參數的傳遞與整型類似，區別在於參數大小。x86 處理器中浮點類型佔 8 個字節，因此在棧中也需要佔用 8 個字節。下表給出浮點參數在棧幀中的位置關係。圖中，調用 tail 函數的第一個和第三個參數均為浮點類型，因此需各佔用 8 個字節，三個參數共佔用 20 個字節。表中 word 類型的大小是 4 字節。

調用語句	參數	棧幀地址
tail(1.414, 2, 3.998e10);	word 0: 1.414	8(%ebp)
	word 1: 1.414	12(%ebp)
	2	16(%ebp)
	word 0: 3.998e10	20(%ebp)
	word 1: 3.998e10	24(%ebp)

5.3.3 結構體和聯合體參數的傳遞

結構體和聯合體參數的傳遞與整型、浮點參數類似，只是其佔用字節大小視數據結構的定義不同而異。x86 處理器上棧寬是 4 字節，故結構體在棧上所佔用的字節數為 4 的倍數。編譯器會對結構體進行適當的填充以使得結構體大小滿足 4 字節對齊的要求。

對於一些 RISC 處理器(如 PowerPC)，其參數傳遞並不是全部通過棧來實現。PowerPC 處理器寄存器中，R3～R10 共 8 個寄存器用於傳遞整型或指針參數，F1～F8 共 8 個寄存器用於傳遞浮點參數。當所需傳遞的參數少於 8 個時，不需要用到棧。結構體和 long double 參數的傳遞通過指針來完成，這與 x86 處理器完全不同。PowerPC 的 ABI 規範中規定，結構體的傳遞採用指針方式，而不是像 x86 處理器那樣將結構從一個函數棧幀中拷貝到另一個函數棧幀中，顯然 x86 處理器的方式更低效。可見，PowerPC 程序中，函數參數採用指向結構體的指針(而非結構體)並不能提高效率，不過通常這是良好的編程習慣。

5.4 x86 函數返回值傳遞方法

函數返回值可通過寄存器傳遞。當被調用函數需要返回結果給調用函數時：

1) 若返回值不超過 4 字節(如 int、short、char、指針等類型)，通常將其保存在 EAX 寄存器中，調用方通過讀取 EAX 獲取返回值。

2) 若返回值大於 4 字節而小於 8 字節(如 long long 或_int64 類型)，則通過 EAX+EDX 寄存器聯合返回，其中 EDX 保存返回值高 4 字節，EAX 保存返回值低 4 字節。

3) 若返回值為浮點類型(如 float 和 double)，則通過專用的協處理器浮點數寄存器棧的棧頂返回。

4) 若返回值為結構體或聯合體，則主調函數向被調函數傳遞一個額外參數，該參數指向將要保存返回值的地址。即函數調用 foo(p1, p2)被轉化為 foo(&p0, p1, p2)，以引用型參數形式傳回返回值。具體步驟可能為：a.主調函數將顯式的實參逆序入棧；b.將接收返回值的結構體變量地址作為隱藏參數入棧(若未定義該接收變量，則在棧上額外開闢空間作為接收返回值的臨時變量)；c. 被調函數將待返回數據拷貝到隱藏參數所指向的內存地址，並將該地址存入%eax 寄存器。因此，在被調函數中完成返回值的賦值工作。

注意，函數如何傳遞結構體或聯合體返回值依賴於具體實現。不同編譯器、平台、調用約定甚至編譯參數下可能採用不同的實現方法。如 VC6 編譯器對於不超過 8 字節的小結構體，會通過 EAX+EDX 寄存器返回。而對於超過 8 字節的大結構體，主調函數在棧上分配用於接收返回值的臨時結構體，並將地址通過棧傳遞給被調函數；被調函數根據返回值地址設置返回值(拷貝操作)；調用返回後主調函數根據需要，再將返回值賦值給需要的臨時變量(二次拷貝)。實際使用中為提高效率，通常將結構體指針作為實參傳遞給被調函數以接收返回值。

5) 不要返回指向棧內存的指針，如返回被調函數內局部變量地址(包括局部數組名)。因為函數返回後，其棧幀空間被「釋放」，原棧幀內分配的局部變量空間的內容是不穩定和不被保證的。

函數返回值通過寄存器傳遞，無需空間分配等操作，故返回值的代價很低。基於此原因，C89 規範中約定，不寫明返回值類型的函數，返回值類型默認為 int。但這會帶來類型安全隱患，如函數定義時返回值為浮點數，而函數未聲明或聲明時未指明返回值類型，則調用時默認從寄存器 EAX(而不是浮點數寄存器)中獲取返回值，導致錯誤！因此在 C++中，不寫明返回值類型的函數返回值類型為 void，表示不返回值。

【擴展閱讀】GCC 返回結構體和聯合體

通常 GCC 被配置為使用與目標系統一致的函數調用約定。這通過機器描述宏來實現。但是，在一些目標機上採用不同方式返回結構體和聯合體的值。因此，使用 PCC 編譯的返回這些類型的函數不能被使用 GCC 編譯的代碼調用，反之亦然。但這並未造成麻煩，因為很少有 Unix 庫函數返回結構體或聯合體。

GCC 代碼使用存放 int 或 double 類型返回值的寄存器來返回 1、2、4 或 8 個字節的結構體和聯合體(GCC 通常還將此類變量分配在寄存器中)。其它大小的結構體和聯合體在返回時，將其存放在一個由調用者傳遞的地址中(通常在寄存器中)。

相比之下，PCC 在大多目標機上返回任何大小的結構體和聯合體時，都將數據複製到一個靜態存儲區域，再將該地址當作指針值返回。調用者必須將數據從那個內存區域複製到需要的地方。這比 GCC 使用的方法要慢，而且不可重入。

在一些目標機上(如 RISC 機器和 80386)，標準的系統約定是將返回值的地址傳給子程序。在這些機器上，當使用這種約定方法時，GCC 被配置為與標準編譯器兼容。這可能會對於 1，2，4 或 8 字節的結構體不兼容。

GCC 使用系統的標準約定來傳遞參數。在一些機器上，前幾個參數通過寄存器傳遞；在另一些機器上，所有的參數都通過棧傳遞。原本可在所有機器上都使用寄存器來傳遞參數，而且此法還可能顯著提高性能。但這樣就與使用標準約定的代碼完全不兼容。所以這種改變只在將 GCC 作為系統唯一的 C 編譯器時才實用。當擁有一套完整的 GNU 系統，能夠用 GCC 來編譯庫時，可在特定機器上實現寄存器參數傳遞。

在一些機器上(特別是 SPARC)，一些類型的參數通過「隱匿引用」(invisible reference)來傳遞。這意味著值存儲在內存中，將值的內存地址傳給子程序。