# Instruções `enter` e `leave`

`enter` cria um stack frame e `leave` destrói um stack frame. Com os parâmetros 0,0 na entrada, eles são basicamente equivalentes a:

```
; enter 0, 0
push ebp
mov ebp, esp
; leave
mov esp, ebp
pop ebp
```

O primeiro parâmetro para entrar especifica uma quantidade de espaço para alocar variáveis locais. Por exemplo, `enter 5, 0` equivale aproximadamente a:

```
push ebp
mov ebp, esp
sub esp, 5      ;desce um pouco mais na pilha, deixando 5 bytes
                ;para serem usados para variáveis locais
```

`enter` também suporta linguagens como Pascal que podem usar funções / procedimentos aninhados:

```
procedure X;
   procedure Y;
   begin
     {…}
   end;
begin
  {…}
end;
```

Em um caso como este, `Y` tem acesso não só às suas próprias variáveis locais, mas também a todas as variáveis locais de `X`. Estas podem ser aninhadas a profundidade arbitrária, então você poderia ter um `Z` dentro de `Y` que tivesse acesso às suas próprias variáveis locais e também das variáveis de Y e de X. O segundo parâmetro a inserir especifica a profundidade de aninhamento, então `X` usaria `enter Sx, 0`, Y usaria `enter Sy, 1` e `Z` usaria `enter Sz, 2` (onde `Sx , Sy` e `Sz` significam o tamanho das variáveis locais para `X`, `Y` e `Z`, respectivamente).
Isso criaria uma cadeia de stack frames para que `Z` tivesse acesso a variáveis locais de `Y` e `X`, e assim por diante. Isso não é muito trivial se as funções são recursivas, então uma invocação de `Z` não pode simplesmente subir a pilha para os dois stack frames mais recentes - ele precisa saltar para stack frames de invocações anteriores de si mesmo e diretamente retornar aos stack frames função / procedimento pai, que é diferente do seu chamador no caso de recursão.
Essa complexidade é também a razão pela qual C e C ++ proíbem funções aninhadas. Dada a presença de `enter / leave`, eles são bastante fáceis de suportar em processadores Intel, mas podem ser consideravelmente mais difíceis em muitos outros processadores que não possuem suporte direto.
Isso, pelo menos, ajuda a explicar uma outra ... característica de enter - para o caso trivial que está sendo usado aqui (ou seja, `enter 0, 0`) é um pouco mais lento que o equivalente usando `push / mov`.