

Estruturas de Linguagem

Francisco Sant'Anna

francisco@ime.uerj.br

<http://github.com/fsantanna/EDL>

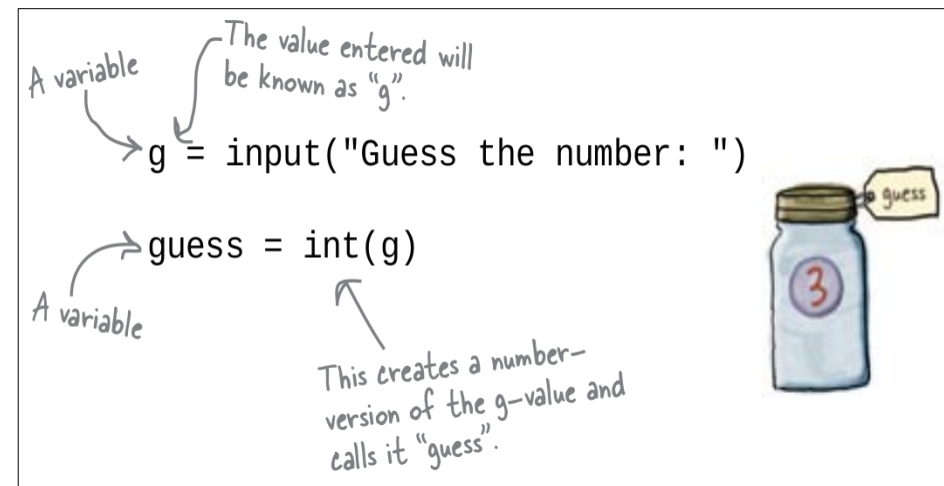
- Nomes
- Binding (amarração)
- Variáveis
- Tempo de Vida

Binding (Amarração)

- Associação entre “entidade” e “atributo”
 - *binding time*
 - *language design time*
 - *language implementation time*
 - *preprocess time*
 - *compile time*
 - *link time*
 - *load time*
 - *run time*

Variáveis

- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador
 - endereço
 - valor
 - tipo
 - **escopo**
 - **tempo de vida**



Créditos: "Head First Programming"

Binding de Memória

- Binding *nome* -> *célula/endereço de memória*
 - alocação
 - desalocação
- Tempo de vida
 - período entre alocação e desalocação
 - característica de execução
- Escopo
 - intervalo de visibilidade da variável
 - característica léxica

Tempo de Vida vs Escopo

```
#include <stdio.h>

int f (void) {
    int v = 0;
    v = !v;
    return v;
}

int main (void) {
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    return 0;
}
```

Tempo de Vida

- Estático (variáveis globais/estáticas)
- Dinâmico
 - pilha (variáveis locais)
 - heap
 - explícito (e.g., *malloc/free*)
 - implícito (e.g., *new/coletor*, construtores primitivos)

Estático

- Associação ocorre antes da execução
- Globais
- Eficiente (+)
 - Acesso direto*
 - Sem overhead de alocação/desalocação
- Expressividade (-)
 - recursão
- Espaço (-)
 - compartilhamento de memória

Pilha (dinâmico)

- Associação ocorre durante a execução
 - (associação de tipo pode ser estática)
- Locais
- Alocação na pilha
- Eficiente (+)
 - Acesso indireto (via `%ebp`)
 - Pouco overhead de alocação/desalocação (em bloco)
- Expressividade (+/-)
 - recursão
- Espaço (+)
 - compartilhamento de memória

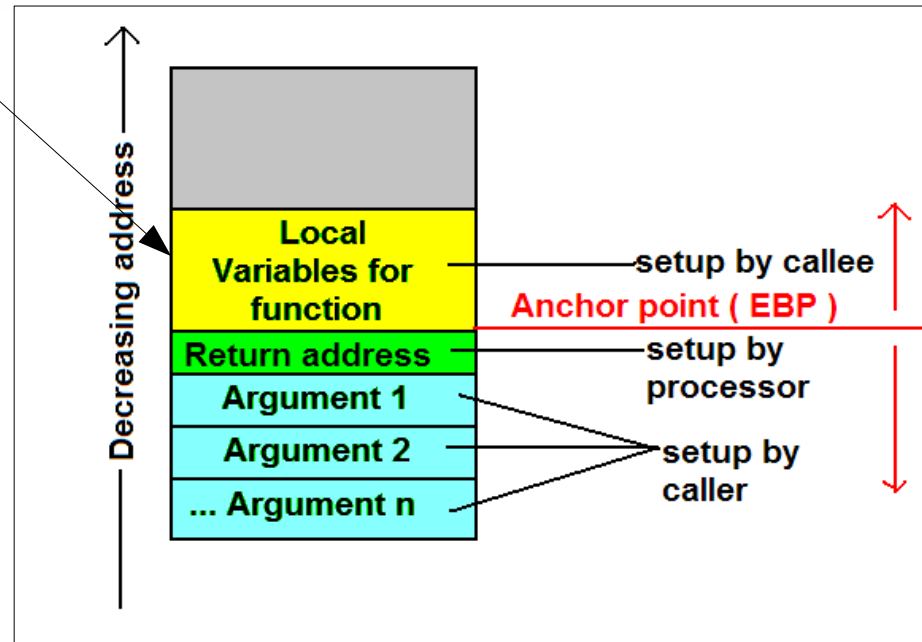
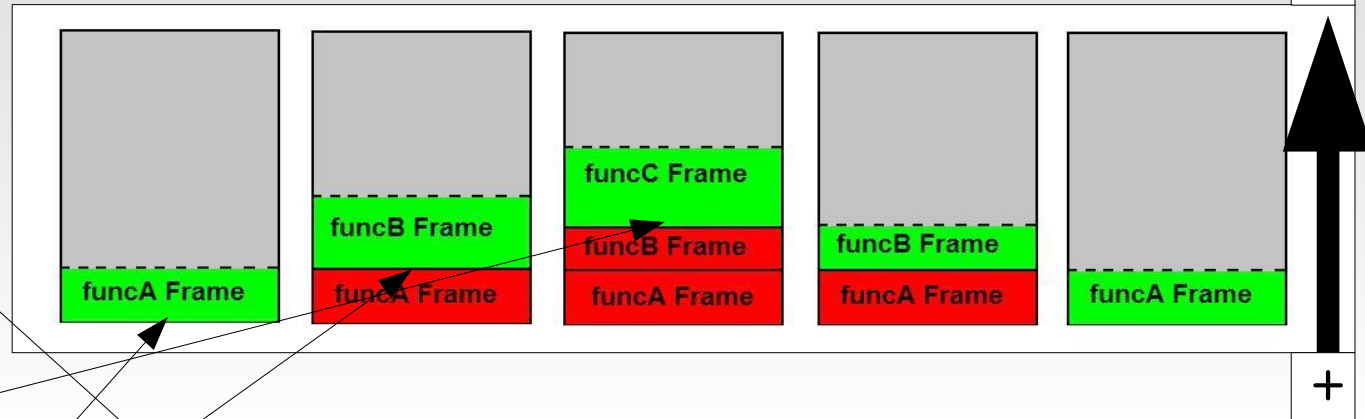
Pilha (dinâmico)

```
function funcC ()  
    local a,b,c  
end
```

```
function funcB ()  
    local a,b,c  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```



Acesso direto/indireto

```
#include <stdio.h>
```

```
int f (void) {  
    int v = 0;  
    v = !v;  
    return v;  
}
```

```
int main (void) {  
    printf("f = %d\n", f());  
    printf("f = %d\n", f());  
    printf("f = %d\n", f());  
    printf("f = %d\n", f());  
    return 0;  
}
```

```
chico@samsung:/data/UERJ/EDL/code$ diff inv.local.s inv.static.s  
13,15c13,14  
<      subl    $16, %esp  
<      movl    $0, -4(%ebp)  
<      cmpl    $0, -4(%ebp)  
---  
>      movl    v.1934, %eax  
>      testl   %eax, %eax  
18,20c17,19  
<      movl    %eax, -4(%ebp)  
<      movl    -4(%ebp), %eax  
<      leave  
---  
>      movl    %eax, v.1934  
>      movl    v.1934, %eax  
>      popl    %ebp  
86a86,87  
>      .local   v.1934  
>      .comm    v.1934,4,4
```

Pilha (dinâmico)



Wikipedia says:

15

Early languages like Fortran did not initially support recursion because variables were statically allocated, as well as the location for the return address.



http://en.wikipedia.org/wiki/Subroutine#Local_variables.2C_recursion_and_re-entrancy

FORTRAN 77 does not allow recursion, Fortran 90 does, (recursive routines must be explicitly declared so).

Most FORTRAN 77 compilers allow recursion, some (e.g. DEC) require using a compiler option (see compiler options chapter). The GNU g77, which conforms strictly to the Fortran 77 standard, doesn't allow recursion at all.

<http://www.ibiblio.org/pub/languages/fortran/ch1-12.html>

share improve this answer

answered Nov 24 '10 at 21:31



Robert Harvey

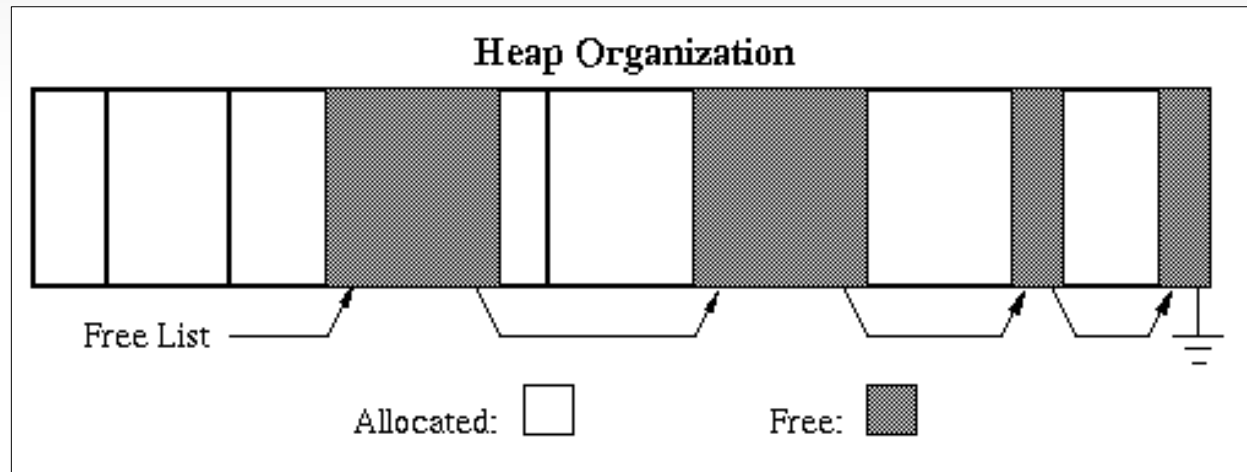
124k ● 30 ● 276 ● 458

Heap (dinâmico)

- Associação ocorre durante a execução
 - (associação de tipo pode ser estática)
- Alocação e Desalocação através de instruções (e.g., *malloc*, *free*)
 - (desalocação pode ser automática: coletor de lixo)
- Eficiente (-)
 - acesso indireto (via ponteiro/alias)
 - overhead de alocação/desalocação (por objeto)
- Expressividade (+)
 - estruturas dinâmicas ajustáveis (vetores, listas, árvores)
- Espaço (+/-)
 - controle total de alocação e desalocação
 - depende do alocador

Heap (dinâmico)

- Alocação “aleatória”
 - impossível determinar padrão estaticamente



Heap (dinâmico)

- Associação “frágil”
 - *dangling pointer* (ponteiro pendente)
 - *memory leak* (vazamento de memória)

valgrind

```
int main (void) {  
    int *v1, *v2;  
    v1 = (int*) malloc(sizeof(int));  
    v2 = (int*) malloc(sizeof(int));  
  
    *v1 = 10;  
    *v2 = 100;  
  
    v1 = v2;  
  
    free(v2);  
  
    printf(">>> v1=%d v2=%d\n", *v1, *v2);  
  
    return 0;  
}
```

associação indeterminada
(ou NULL)

tipagem fraca

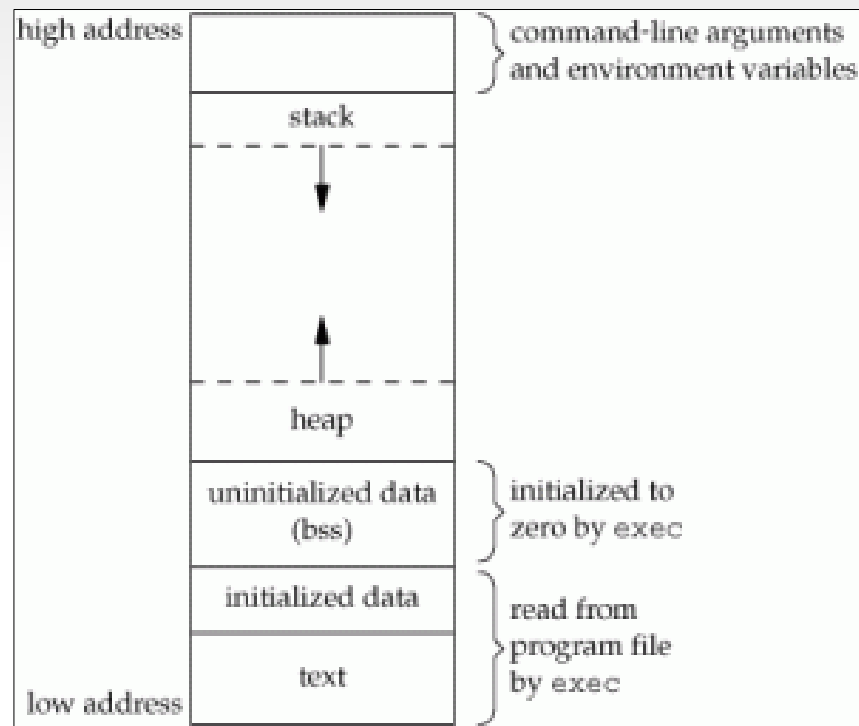
indireção (custo explícito)

re-binding

memory leak

dangling pointers

Organização da Memória



Exemplo em Lua

```
chico@samsung:/data/UERJ/EDL/code/scope$ cat ex.lua
```

```
A = 1
local a = 2
t = { a=3 }
print(A + a + t.a)
```

```
chico@samsung:/data/UERJ/EDL/code/scope$ lua ex.lua
```

```
6
```

```
chico@samsung:/data/UERJ/EDL/code/scope$ luac -l ex.lua
```

```
main <ex.lua:0,0> (14 instructions, 56 bytes at 0x1a0a530)
```

```
0+ params, 4 slots, 0 upvalues, 1 local, 7 constants, 0 functions
```

1	[1]	LOADK	0 -2	; 1
2	[1]	SETGLOBAL	0 -1	; A
3	[2]	LOADK	0 -3	; 2
4	[3]	NEWTABLE	1 0 1	
5	[3]	SETTABLE	1 -5 -6	; "a" 3
6	[3]	SETGLOBAL	1 -4	; t
7	[4]	GETGLOBAL	1 -7	; print
8	[4]	GETGLOBAL	2 -1	; A
9	[4]	ADD	2 2 0	
10	[4]	GETGLOBAL	3 -4	; t
11	[4]	GETTABLE	3 3 -5	; "a"
12	[4]	ADD	2 2 3	
13	[4]	CALL	1 2 1	
14	[4]	RETURN	0 1	