Abstrações de unidades
(subprogramas: parte I)

Modelos de Linguagens de Programação

Abstração: revisão

Facilidades de abstração fundamentais em LPs:

1. Abstrações de processos (ou de unidades):
   - conceito central
   - oferecidas desde os primórdios da computação
   - aumentam legibilidade e abstraem detalhes
2. Abstrações de dados (TADs):
   - igualmente importantes (a partir de 1980)
   - omitem detalhes de representação dos dados
   - tornam acessíveis um conjunto de operações

Abstração de processo

Subprograma:

Unidade lógica que agrupa instruções que realizam uma tarefa, minimizando o consumo de espaço e o esforço de desenvolvimento

- Tipos:
  - Abstração procedimental (de comandos) - define ou estende os comandos disponíveis na LP
  - Abstração funcional (de expressões ou valores) - estende a lista de operadores da linguagem

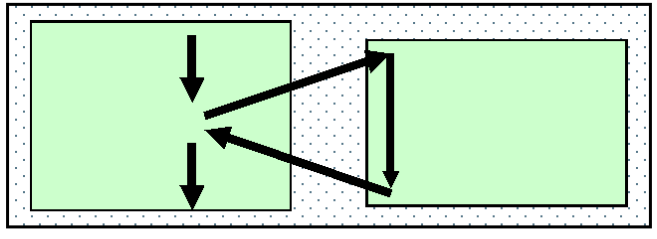
Subprogramas

Questões de projeto:

- Como se dá a ativação dos subprogramas?
- Quais ambientes de referência são permitidos/usados? São locais ou globais? Qual é o alcance e a visibilidade dos componentes?
- Qual é o funcionamento e a semântica das variáveis locais?
- Há proteção de componentes locais?
- Subprogramas aninhados são permitidos?
- Sobrecarga subprogramas é permitida?
- Podem ser usadas/declaradas unidades genéricas?

Subprogramas

- Possuem um único ponto de entrada
- A unidade chamadora é suspensa durante sua execução (somente 1 subprograma em execução a cada momento)
  - **Contraexemplo**: unidades concorrentes
- O controle sempre retorna ao chamador quando o subprograma termina
- **Exemplos**: procedimentos, funções e métodos



Subprogramas: conceitos relacionados

- **Como definir um subprograma?**
  - Especificar:
    - **Cabeçalho (interface/protocolo)**:
      - tipo do subprograma + nome + lista de parâmetros
    - **Corpo (ação)**:
      - implementação da ação
      - ponto de entrada da execução
      - define um ambiente de execução
      - código reentrante (mesmo código usado para diferentes ativações)

```
int soma(int a, int b)
{
   int result;
   result = a + b;
   return result;
}
```

Subprogramas: conceitos relacionados

- **Como declarar um subprograma?**
  - Informar seu protocolo/interface
  - Declarações são denominadas de:
    - protótipos (*prototype*) em C
    - encaminhamentos (*forward*) em Pascal

Exemplos de cabeçalhos:

- Fortran:

```
SUBROUTINE somar(parâmetros)
```

- Pascal:

```
PROCEDURE somar(a, b: integer)
FUNCTION soma(a, b: integer):
integer
```

- C:

```
void somar(int, int)
int soma(int, int)
```

Subprogramas: conceitos relacionados

- **A chamada** de um subprograma é uma requisição explícita para que o subprograma seja executado
  - Chamada de função (expressão):

```
menor = x + y - maior(x,y);
```

  - Chamada de procedimento (comando):

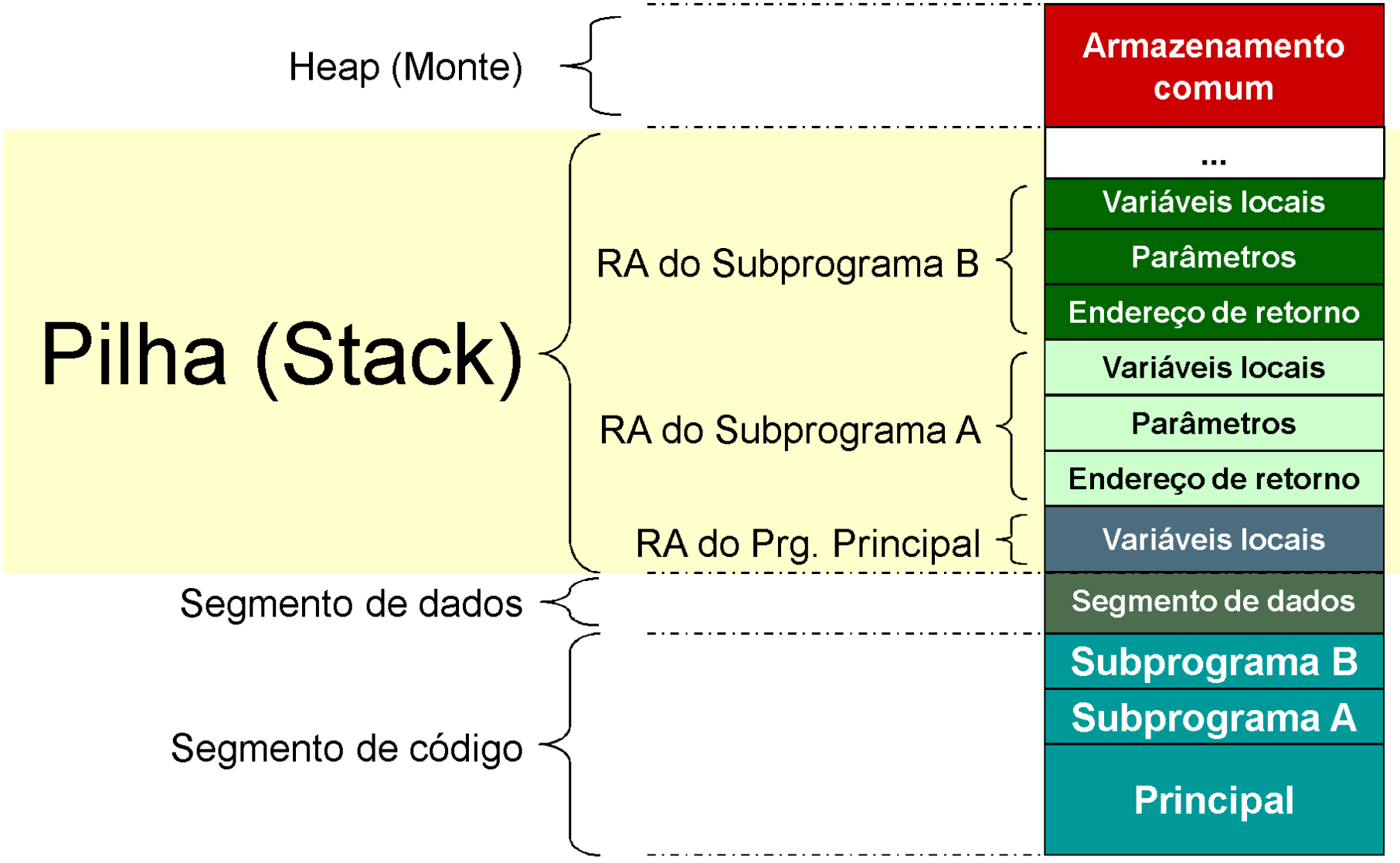
```
if(x[i] < x[j]) troca(x[I], x[j]);
```

- Estabelece ligação entre o ponto de chamada e o código de execução
- Um subprograma ativo é aquele que iniciou sua execução mas ainda não a terminou

Semântica de chamadas e retornos

- O binding de subprogramas exige armazenamento de informações sobre:
  - estado da execução no ponto de chamada
  - passagem de parâmetros, dependendo do modo
  - ambiente global de execução
  - ambiente local de execução
  - endereço de retorno
- Informações são organizadas em um registro de ativação (RA)
- A cada invocação uma instância do RA é criada na pilha

Visão geral da memória



Registro de Ativação (RA)

- Também conhecido como *frame*
- Estrutura de dados que contém informações para:
  - A execução da unidade requisitada
  - O retorno do fluxo de execução à unidade requisitante
- Seu formato depende da definição/implementação da LP
- Linguagens Algol-*like* devem se preocupar com:
  - Método de passagem de parâmetro (e.g., valor, referência)
  - Alocação dinâmica de variáveis locais
  - Recursão (mais de uma instância, com execução incompleta)
  - Escopo estático para acesso à variáveis não locais

RA típico em linguagens Algol-*like*

Topo da pilha

Aponta para o **topo da instância do RA** de uma ativação **do chamador** (usado na destruição da instância do RA atual, principalmente quando este tem variáveis temporárias derivadas da linguagem de máquina)

Ponteiro para o **segmento de código do chamador** (+deslocamento da instrução seguinte) (PC)

Aponta para a **base da instância do registro de ativação** de uma ativação **do pai estático** (ajuda no acesso à variáveis não locais)

RA: observações

- Compiladores podem utilizar o leiaute que lhe for mais conveniente
- (mas...) Fabricantes de processadores especificam esquemas de leiaute "padrão"
  - *Calling conventions*
- Com isso, o código compilado com um compilador pode chamar funções compiladas por outro

Modelo didático/genérico adotado:

RA: algumas considerações

- A instância de um RA é:
  - criada (colocada na pilha) quando é feita uma invocação
  - destruída (retirada da pilha) quando termina a execução, de forma normal ou abrupta (erro)

  OBS: havendo várias chamadas ao mesmo subprograma, cada chamada cria uma nova instância de RA na pilha, mas o código de execução é o mesmo (único) (exceção para definições do tipo *inline*)

RA: algumas considerações

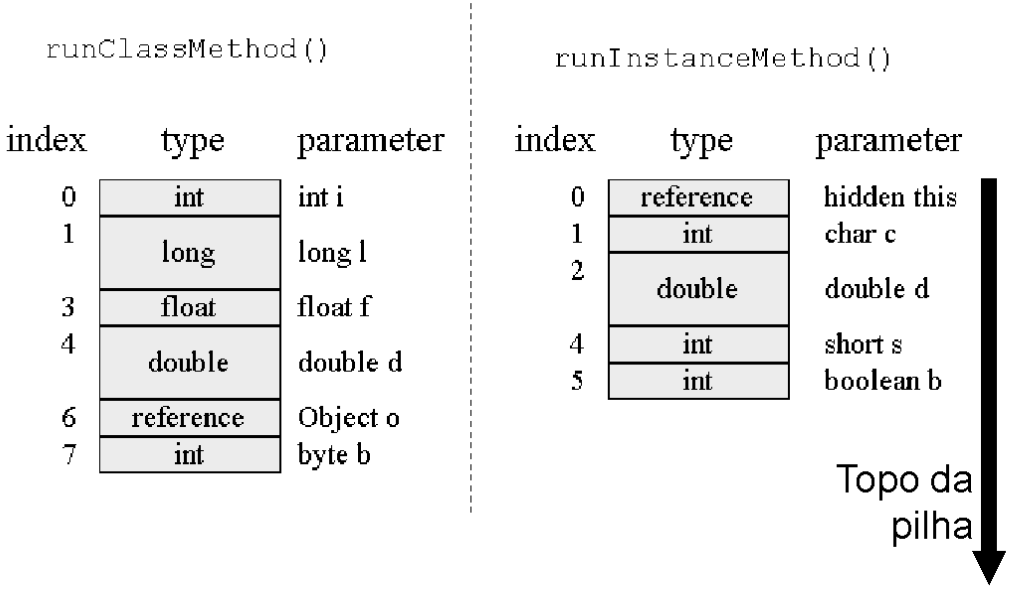
- Sobre variáveis locais:
  - variáveis escalares locais são vinculadas ao armazenamento dentro da instância atual do RA
  - estruturas e objetos (instâncias) são normalmente alocadas em outro lugar (p.ex. Heap) e somente seus descritores e um ponteiro para elas são colocadas no RA
  - variáveis locais são alocadas e possivelmente inicializadas no subprograma gerado

Exemplo em Java

Trecho de código

```
class Example3a {
  public static int
    runClassMethod(int i,
    long l, float f, double d,
    Object o, byte b) {
    return 0;
  }
  public int
    runInstanceMethod(char c,
    double d, short s, boolean b){
    return 0;
  }
}
```

Trecho dos RAs (frames)



Fonte: Venners (1998)

RA: exemplo em Pascal

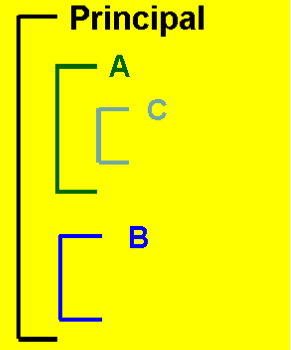
```
Procedure sub(var total: real; parte: integer);
Var lista: array [1..5]: of integer;
    soma: real;
Begin
   { ... }
End;
```

Exercício: como ficaria a estrutura do RA do procedimento acima? (considere que cada tipo ocupa 1 byte/espaço no RA)

RA: exemplo de análise

```
Program Exemplo;
Var P: Real;
  Procedure A(X: integer);
  Var Y: boolean;
     Procedure C(Q: boolean);
     begin          { C }
       { ... }
     end;           { C }
     Begin          { A }
       { ... }
        C(Y);
       { ... }
     End;           { A }
  Procedure B(R: real);
  Var S, T : integer;
  Begin             { B }
       { ... }
        A(S);
       { ... }
  End;              { B }
Begin               { Principal }
  { ... }
   B(P);
  { ... }
End;                { Principal }
```

Estrutura:



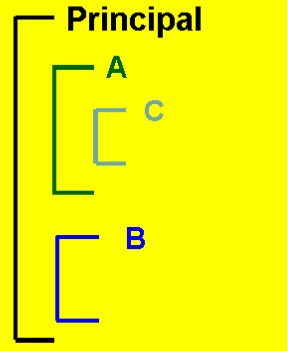
RA: exemplo de análise

```
Program Exemplo;
Var P: Real;
   Procedure A(X: integer);
   Var Y: boolean;
      Procedure C(Q: boolean);
      begin          { C }
        { ... }
      end;           { C }
      Begin          { A }
        { ... }
         C(Y);
        { ... }
      End;           { A }
   Procedure B(R: real);
   Var S, T : integer;
   Begin             { B }
        { ... }
         A(S);
        { ... }
   End;              { B }
Begin                { Principal }
  { ... }
   B(P);
  { ... }
End;                 { Principal }
```

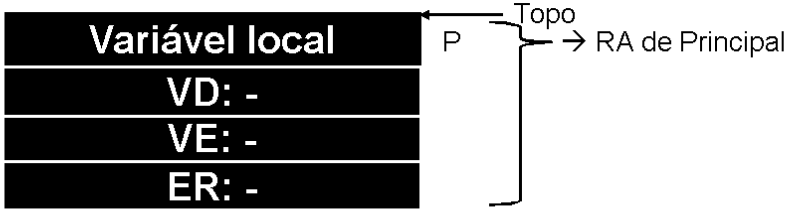
Sequência de chamadas:

- **Principal chama B**

Estrutura:



Ponto 0



RA: exemplo de análise

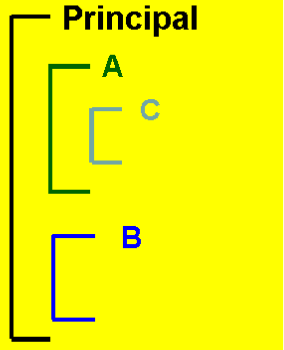
```
Program Exemplo;
Var P: Real;
   Procedure A(X: integer);
   Var Y: boolean;
      Procedure C(Q: boolean);
      begin          { C }
        { ... }
      end;           { C }
      Begin          { A }
        { ... }
         C(Y);
        { ... }
      End;           { A }
   Procedure B(R: real);
   Var S, T : integer;
   Begin             { B }
        { ... }
         A(S);
        { ... }
   End;              { B }
Begin                { Principal }
  { ... }
   B(P);
  { ... }
End;                 { Principal }
```

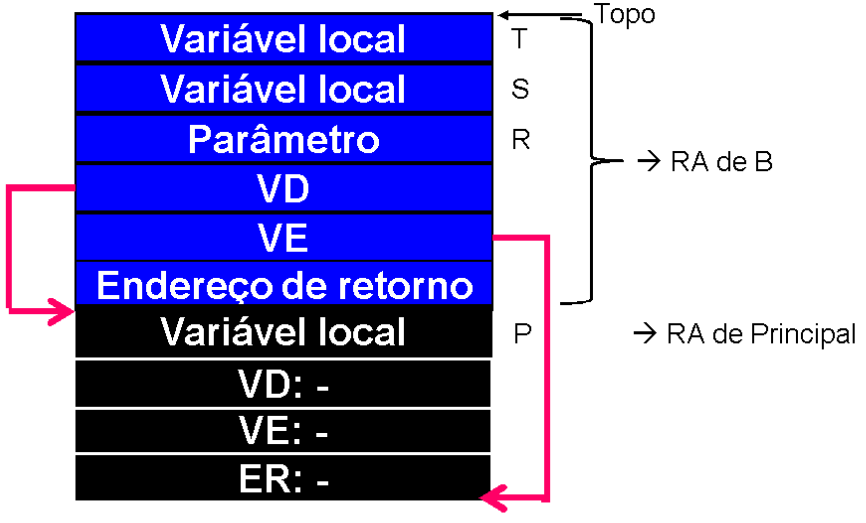
Ponto 1

Sequência de chamadas:

- **Principal chama B**
- **B chama A**

Estrutura:





RA: exemplo de análise

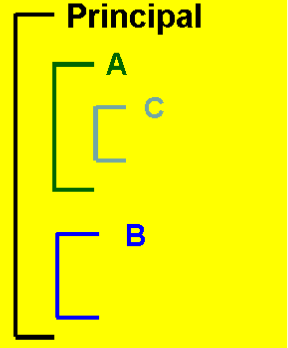
```
Program Exemplo;
Var P: Real;
   Procedure A(X: integer);
   Var Y: boolean;
      Procedure C(Q: boolean);
      begin            { C }
        { ... }
      end;             { C }
      Begin            { A }
        { ... }
         C(Y);
        { ... }
      End;             { A }
   Procedure B(R: real);
   Var S, T : integer;
   Begin               { B }
        { ... }
         A(S);
        { ... }
   End;                { B }
Begin                  { Principal }
  { ... }
   B(P);
```

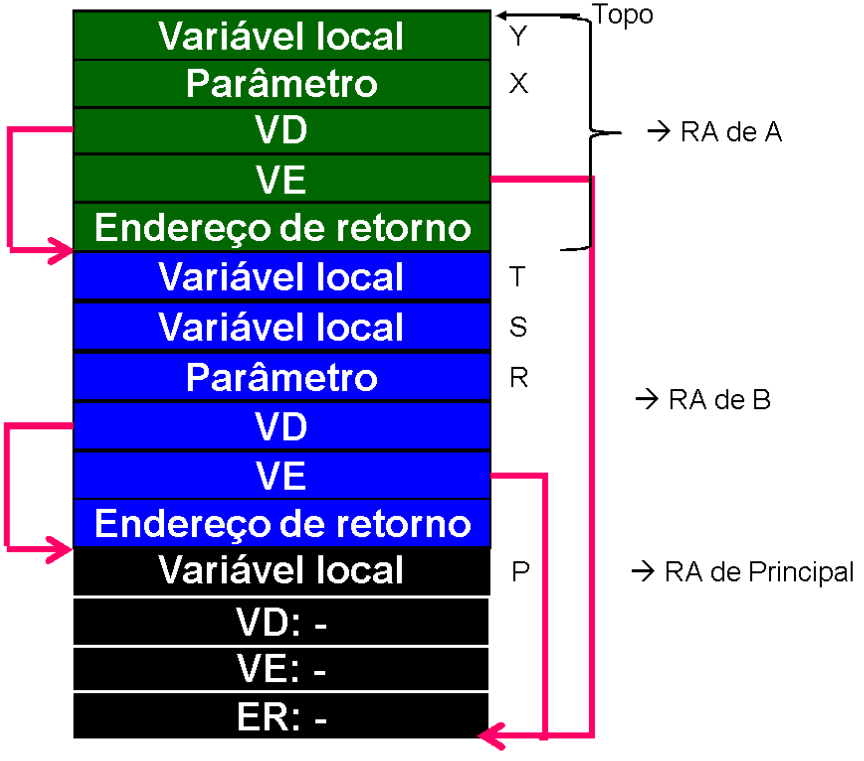
Ponto 2

Sequência de chamadas:

- Principal chama B
- B chama A
- A chama C

Estrutura:





RA: exemplo de análise

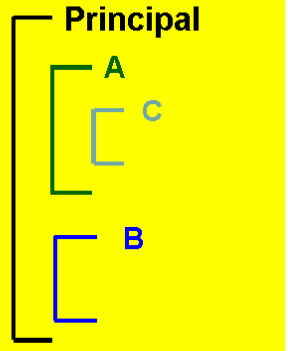
```
Program Exemplo;
Var P: Real;
   Procedure A(X: integer);
   Var Y: boolean;
      Procedure C(Q: boolean);
      begin              { C }
        { ... }
      end;               { C }
      Begin              { A }
        { ... }
         C(Y);
        { ... }
      End;               { A }
   Procedure B(R: real);
   Var S, T : integer;
   Begin                 { B }
        { ... }
         A(S);
        { ... }
   End;                  { B }
Begin                    { Principal }
  { ... }
   B(P);
```

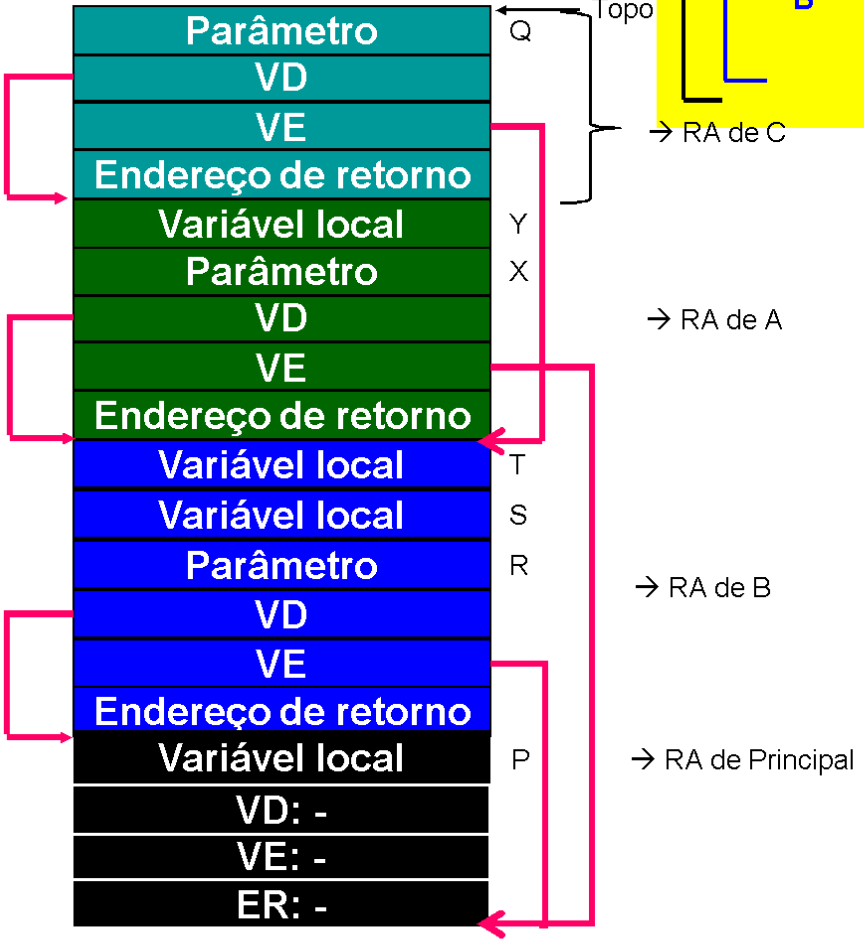
Ponto 3

Sequência de chamadas:

- Principal chama B
- B chama A
- A chama C

Estrutura:





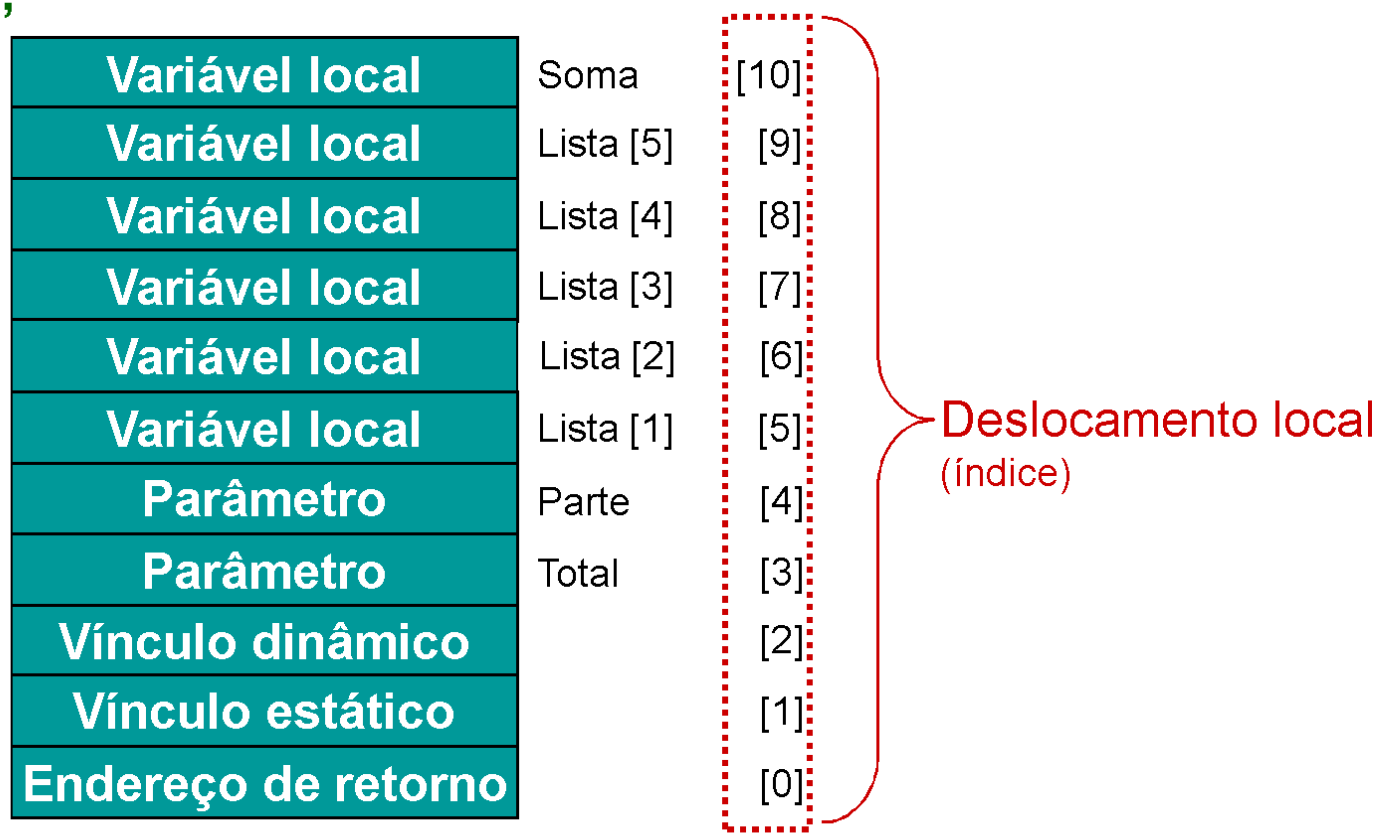
RA: encadeamento dinâmico

- Encadeamento de chamadas
- Conjunto de vínculos dinâmicos presentes na pilha em determinado momento
- Apresenta a história dinâmica de como a execução chegou na posição atual

Referências à variáveis locais podem ser representadas no código como deslocamentos do início do registro de ativação do escopo local (denominado deslocamento_local), que pode ser determinado no momento da compilação

RA: deslocamento local

```
Procedure sub(var total: real; parte: integer);
Var lista: array [1..5]: of integer;
    soma: real;
Begin
   { ... }
End;
```



Subprogramas: sequência de chamada

1. Prólogo (código executado pela sub-rotina chamada, antes da execução propriamente dita)
   - faz alocação de variáveis locais
   - inicializa (variáveis, etc.)
   - atualiza ponteiros para alocação dinâmica na pilha
2. Subrotina (propriamente dita)
3. Epílogo (código executado ao final da sub-rotina, antes de retornar ao chamador)
   - desalocação de área na memória e pilha

Subprogramas: **acesso a dados**

- Acesso direto à variáveis não locais
- Acesso através da passagem de parâmetros
  - Nomes locais ao subprograma
  - Mais flexível, legível, seguro
  - Computações ou expressões como parâmetros

Acesso direto à variáveis não locais

- Variáveis não-locais: aquelas declaradas em outro bloco mas visíveis dentro do subprograma

  - Acesso em dois passos:
    1. Localizar instância (RA) onde a variável foi declarada;
    2. Usar o deslocamento local para acessá-la.

    - Como localizar a instância do RA?
      - Utilizar um dos seguintes métodos de localização:
        - Encadeamento estático (*static chain*)
        - Display

Acesso direto à variáveis não locais

Observações:

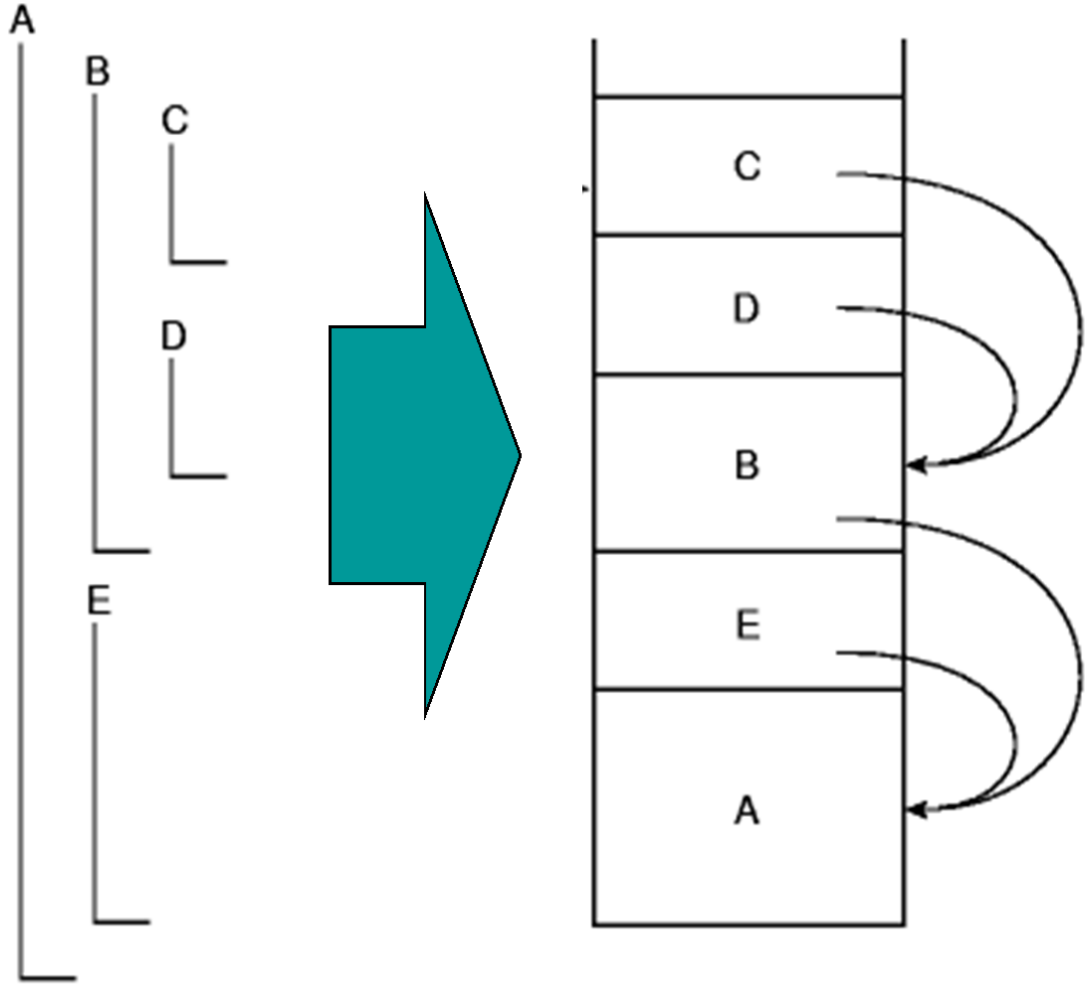
- Somente variáveis declaradas em escopos ancestrais são visíveis
- A existência de instâncias de registros de ativação de todos os ancestrais estáticos é garantida quando uma de suas variáveis for referenciada por um procedimento aninhado (não necessariamente de forma adjacente)
- Um procedimento só pode ser chamado se as suas unidades de programa ancestrais estáticas estão ativas
- A declaração correta de uma variável não-local é a primeira encontrada

Encadeamento estático

- Cadeia de vínculos estáticos que conectam certas instâncias do registro de ativação
- Vincula todos os ancestrais estáticos de um subprograma em execução

- Pode ser usado para acessar variáveis não-locais em linguagens de escopo estático
- Profundidade estática: número associado ao escopo estático que indica quão profundamente ele está aninhado no escopo mais externo
- O compilador pode determinar que uma referência é não-local e calcular o tamanho do encadeamento estático necessário para alcançá-la (diferença entre as profundidades estáticas)

Leiaute da Pilha para *static chain*



Encadeamento estático

- **Estrutura do referenciamento a uma variável (considerando o uso de *static chain*):**

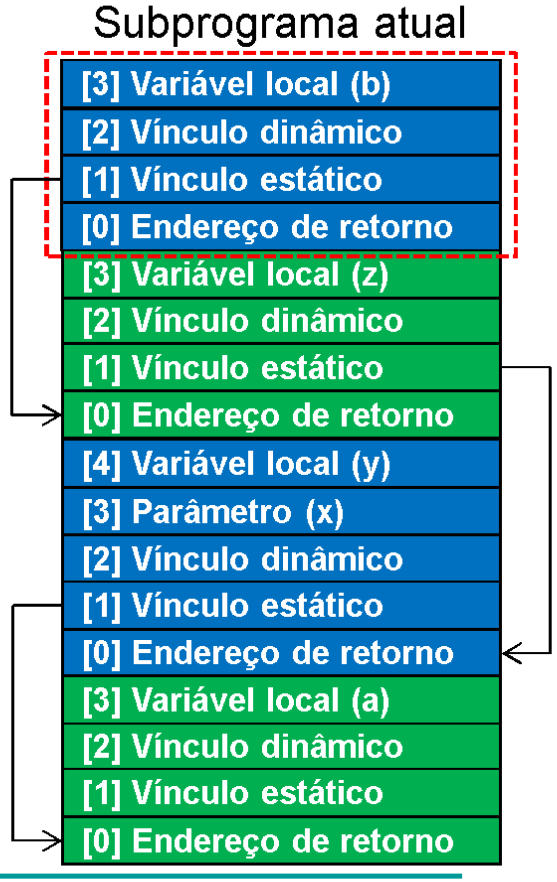
  (<*profundidade_estática*>, <*deslocamento_local*>)

  - **Exemplos:**

    (0, 3) → primeiro parâmetro ou variável do subprograma atual: variável b

    (1, 3) → primeiro parâmetro ou variável do subprograma de nível anterior (pai estático): variável z

    (2, 4) → segundo parâmetro ou variável do subprograma de dois níveis atrás (avô estático): variável y



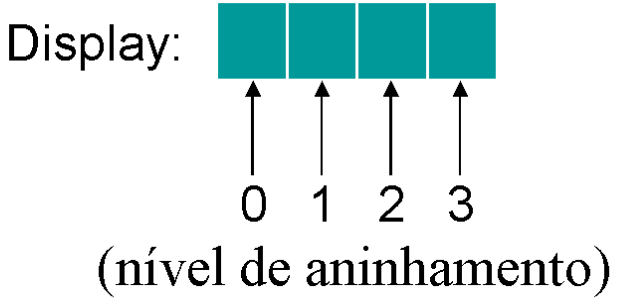
Encadeamento estático

- Problemas:
  - localizar uma variável pode envolver vários acessos, dependendo do nível de profundidade
  - deve-se localizar a instância mais recente do RA do subprograma pai (através do encadeamento dinâmico ou outra técnica), pois ele pode ter várias instâncias na pilha (funções recursivas, por exemplo)

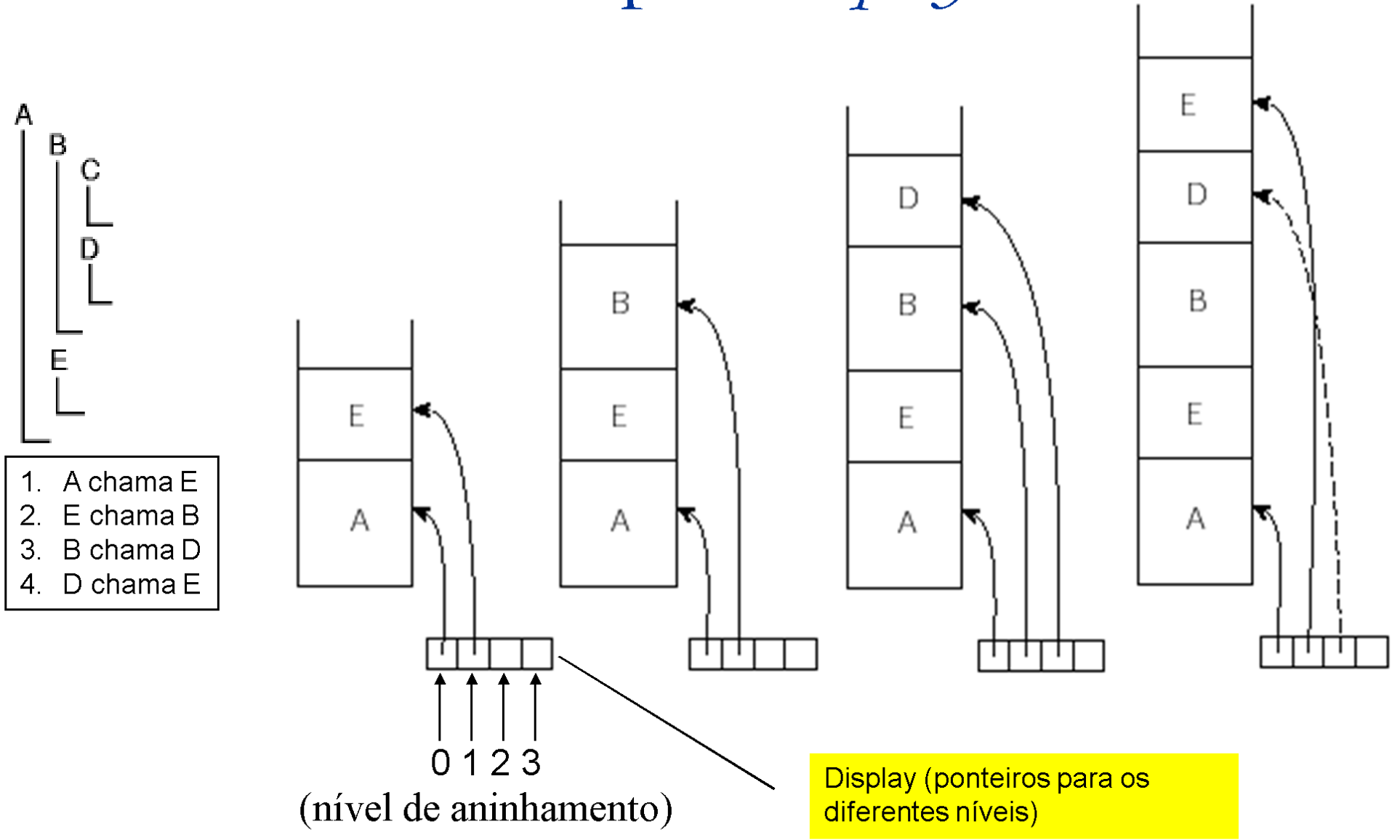
- Por isso, não são tão utilizados atualmente

Displays

- Ao invés de ficarem nos registros de ativação, os vínculos ficam em uma estrutura específica: o display
- Display é um *array* que armazena a lista de endereços das instâncias de um registro de ativação na pilha, na ordem em que estão aninhados
- Podem ser armazenados em registradores (seu tamanho é reduzido/limitado)



Leiaute da Pilha para *display*



Displays

- Acesso à variáveis não locais sempre custa dois passos (independente do número de níveis):
  1. buscar vínculo para o RA pelo deslocamento no display (nível atual é o índice no vetor)
  2. o deslocamento local (posição da variável dentro do RA) é computado como no deslocamento estático (processadores possuem instruções de acesso com deslocamento)

Displays

- Problema: toda chamada e todo retorno exigem que o display seja modificado para refletir a nova situação

  - Chamada:
    - Salvar ponteiro do nível atual no RA da função chamada
    - Nível atual do display aponta para função chamada

  - Saída:
    - Ponteiro RA volta para display
    - RA sai da pilha

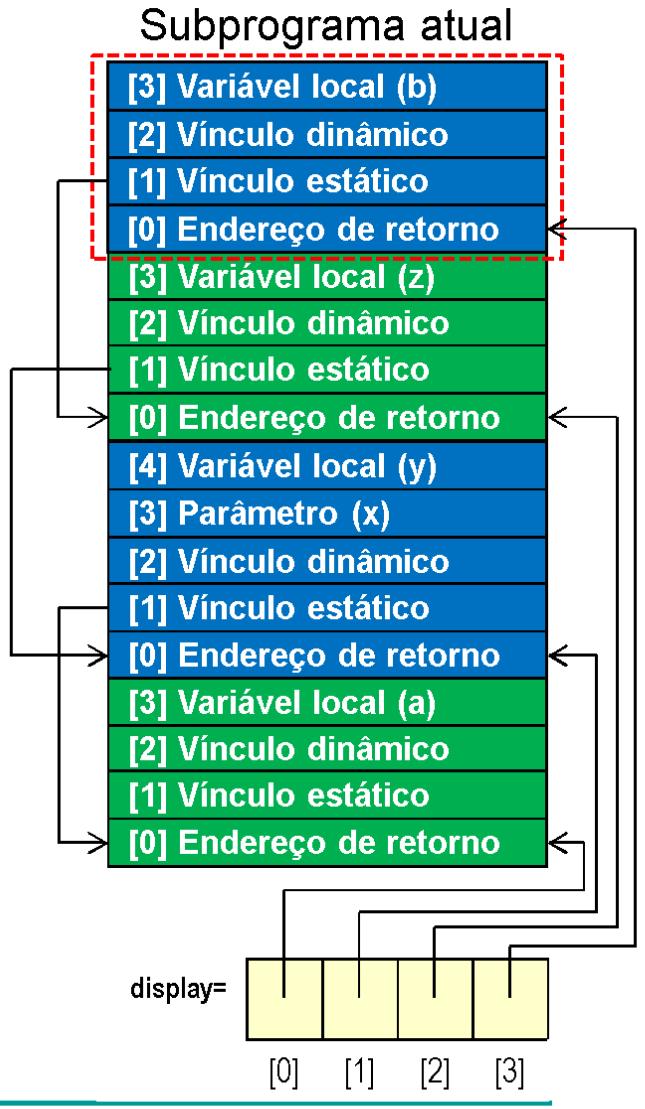
Displays

- Para fins didáticos, podemos utilizar o nível do RA como índice para o *display*:

(0, 3) → primeiro parâmetro ou variável do subprograma de nível 0: variável a

(1, 4) → segundo parâmetro ou variável do subprograma de nível 1: variável y

(2, 3) → primeiro parâmetro ou variável do subprograma de nível 2: variável z



Leitura fortemente recomendada

- Sebesta, R. W. Subprogramas (capítulo 9). In: Sebesta, R. W. **Conceitos de Linguagens de programação**. 5a edição. Porto Alegre: Bookman, 2003.
- Sebesta, R. W. Implementando Subprogramas (capítulo 10). In: Sebesta, R. W. **Conceitos de Linguagens de programação**. 5a edição. Porto Alegre: Bookman, 2003.
- **Call Stack (Wikipedia)**. http://en.wikipedia.org/wiki/Call_stack
- Venners; Bill. The Java Virtual Machine (chapter 5). In: **Inside the Java Virtual Machine.** McGraw-Hill, 1998.