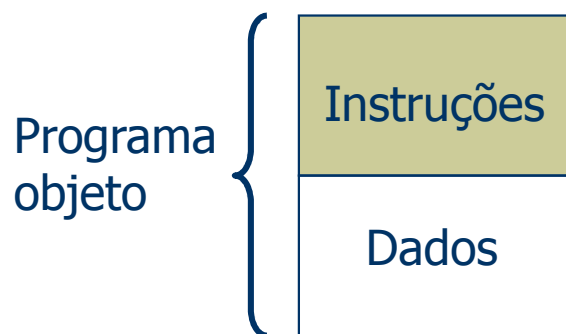


2.4 - Organização de Memória em Tempo de Execução

- ♦ Existe um certo número de elementos aos quais devemos alocar memória para que o programa objeto possa ser executado. A necessidade mais óbvia de memória é para as instruções do programa objeto e para as estruturas de dados usadas no programa.



Além disso, temos a necessidade (menos óbvia) de espaço para:

- endereço de retorno de procedimentos;
- localizações temporárias requeridas para avaliação de expressões;
- "buffers" de entrada e saída; ...

- ♦ Vamos discutir nesse capítulo algumas maneiras de organizar o espaço de dados.

Área de Dados

- ♦ Uma área de dados é um conjunto de células contíguas de memória (bloco de armazenamento) para dados logicamente relacionados. O termo registro de ativação é também usado para denominar uma área de dados.
- ♦ As áreas de dados são classificadas em estáticas e dinâmicas.

Organização de Memória em Tempo de Execução

Área de Dados Estática

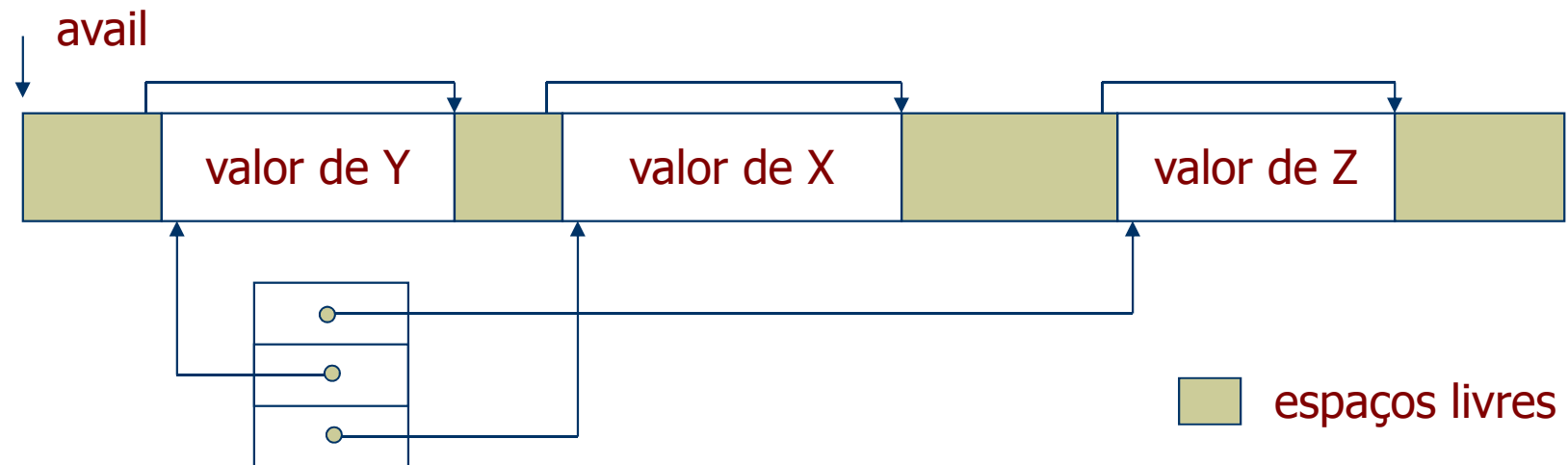
- ♦ Se o tamanho de todo item de dado pode ser determinado pelo compilador (por exemplo, vetor de comprimento ajustável não é permitido na linguagem) e se chamadas recursivas a procedimentos não são permitidas, então o espaço para os dados do programa pode ser alocado em tempo de compilação, ou seja, de forma estática.
- ♦ Nesse caso, então, tem-se um conjunto fixo de células de memória alocadas antes da execução que permanecem alocadas durante toda a execução do programa. Numa área estática seus dados podem ser referenciados por um endereço absoluto.

Área de Dados Dinâmica

- ♦ Uma área dinâmica não está sempre presente durante a execução do programa e seu tamanho poderá, eventualmente, variar. Se a linguagem de programação permite chamadas recursivas a procedimentos ou aceita estruturas de dados de tamanho ajustável, então será necessário algum tipo de gerenciamento dinâmico de memória.
- ♦ Existem dois tipos principais de alocação dinâmica de memória:
 - alocação por pilha
 - alocação por "heap"

Organização de Memória em Tempo de Execução

- ♦ A alocação por pilha é útil para manipular procedimentos recursivos (e também para o tratamento de escopo). Toda vez que um procedimento é chamado (ou quando se entra no escopo) sua área de dados é colocada no topo da pilha e quando o procedimento termina (ou quando se deixa o escopo) essa área é retirada da pilha.
- ♦ A alocação por heap é útil para implementar dados cujo tamanho varia à medida que o programa vai sendo executado. Nesse tipo de alocação, um grande bloco de memória é dividido em sub-blocos de comprimentos variáveis, alguns usados por dados e outros livres. Quando um dado é criado deve-se encontrar um sub-bloco de tamanho suficiente e quando um dado não é mais necessário, sua área é liberada.
- ♦ **Exemplo:**



Organização de Memória em Tempo de Execução

- ♦ Vamos tratar nesse capítulo da alocação por pilha. Esse esquema de alocação de memória, embora não possa ser usado para manipular alocações e liberações arbitrárias (como as necessárias para o tratamento de listas, por exemplo), é muito conveniente para alocação de espaço de uma linguagem estruturada.
- ♦ A área de dados deve conter:
 - espaço para nomes simples e ponteiros para "arrays" e outras estruturas de dados locais ao bloco;
 - espaço para posições temporárias necessárias para avaliação de expressões e passagem de parâmetros de procedimentos;
 - informações relacionadas a atributos de nomes locais e de parâmetros formais quando esses atributos não puderem ser determinados em tempo de compilação (como, os limites de um vetor, por exemplo);
 - o endereço de retorno de procedimentos;
 - um ponteiro para a área de dados seguinte.
- ♦ Quando uma rotina é chamada, invoca-se uma rotina GETAREA que aloca espaço para a sua área de dados. O espaço necessário só é conhecido neste momento. Antes da rotina retornar, invoca-se FREEAREA para devolver o espaço usado e não mais necessário. As referências a estes dados devem ser feitas de forma indireta. Para cada dado estará associada a área de dados a que pertence e o "deslocamento" dentro dessa área de dados.

Organização de Memória em Tempo de Execução

- ♦ Aos dados cujos tamanhos não podem ser determinados até que o procedimento seja chamado são reservadas posições imediatamente acima da área de dados desse procedimento.

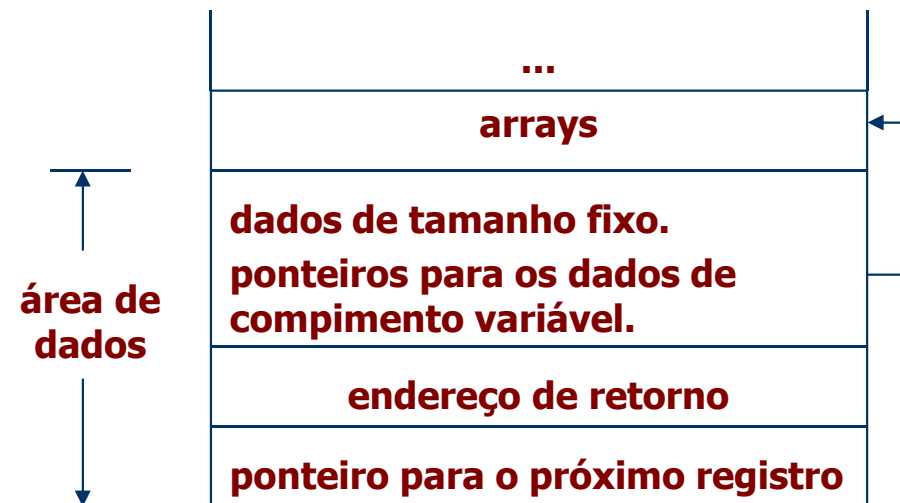
- ♦ Portanto:

endereço de uma variável = (base, deslocamento)

endereço absoluto = conteúdo[registrador base] + deslocamento

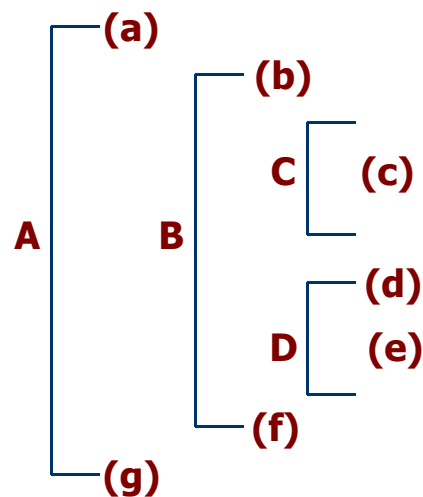
onde:

- base - endereço inicial da área de dados (não é conhecido em tempo de compilação)
- deslocamento - endereço da variável em relação ao começo da área de dados (conhecido em tempo de compilação).

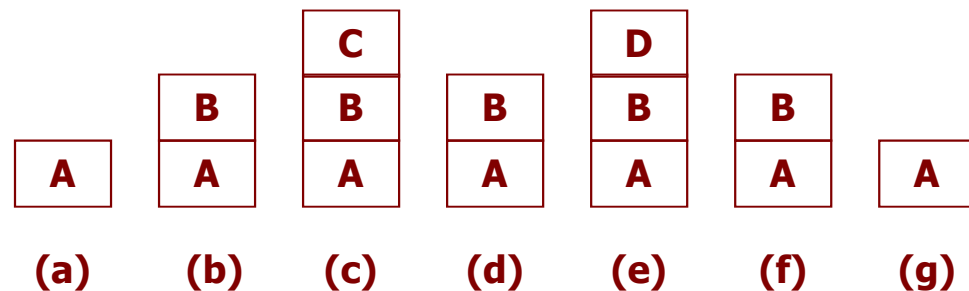


Organização de Memória em Tempo de Execução

- Quando um procedimento P chama um procedimento Q, a área de dados de Q e o espaço para seus dados de tamanho ajustável são empilhados. Quando Q termina, o endereço de retorno é recuperado da área de dados, que é então retirada da pilha.
- Exemplo:



comportamento da pilha de áreas de dados



- Observação: O escopo de um nome é a parte do programa na qual este nome pode ser usado. As regras que determinam o escopo de um nome podem ser estáticas ou dinâmicas. As regras estáticas definem o escopo em termos da estrutura sintática do programa, isto é, pode-se determinar, verificando o programa, a que declaração o uso de um nome se refere. Por exemplo, a regra do "aninhamento mais próximo": a menção a um nome A refere-se à declaração de A no primeiro bloco que inclui esse nome.

Organização de Memória em Tempo de Execução

- ♦ Em algumas linguagens (LISP, por exemplo) as regras de escopo são dinâmicas (a região na qual um nome é válido varia com a execução). É a regra do "bloco iniciado mais recentemente e ainda não finalizado".

Nesse capítulo, vamos discutir as linguagens com regra de escopo estático.

Exemplo:

```
begin
  procedure D;
  begin
    ...(4)...
  end;
  procedure A;
  begin
    procedure B;
    begin
      ...(3)...
      D;
      ...(5)...
    end;
    procedure C;
    begin
      ...
    end;
    ...(2)...
    B;
    ...
  end;
  ...(1)...
  A;
  ...
end.
```

Nos pontos assinalados teremos as seguintes áreas de dados ativas:

- (1) main
- (2) main, A
- (3) main, A, B
- (4) main, D
- (5) main, A, B

Portanto, é necessário (para poder gerenciar o conteúdo do registrador base) manter uma lista das áreas de dados ativas num determinado instante da execução do programa. Isto é feito por uma estrutura denominada "display".

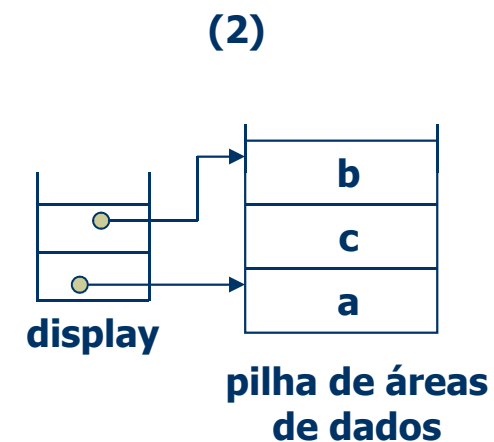
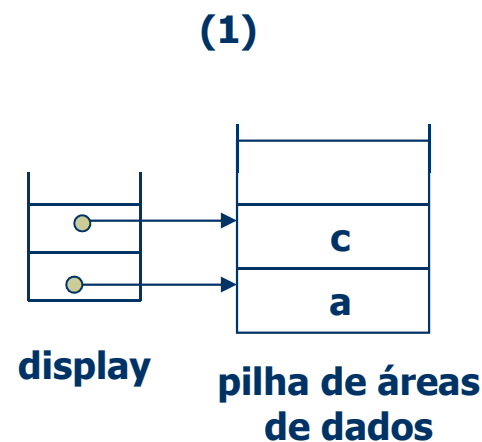
Organização de Memória em Tempo de Execução

"Display"

- Em qualquer ponto, durante a execução do programa objeto, a pilha contém espaço para as variáveis dos blocos que foram ativados e ainda não desativados. Entretanto, somente aqueles conjuntos de variáveis que possuem declarações atualmente válidas serão acessíveis. Os endereços das áreas de dados que contêm as variáveis acessíveis nesse ponto estarão armazenados no "display".
- Exemplo:

```
begin
  real a;
  procedure Q;
  begin
    real b;
    ...(2)...
  end;
  begin
    real c;
    ...
    (1) Q;
    ...
  end;
  ...
  Q;
  ...
end.
```

Nesse caso, a e c são acessíveis quando da chamada a Q (1). Dentro de Q, somente a e b são acessíveis. Note que nesse ponto (2) c continua na pilha mas é inacessível.



Organização de Memória em Tempo de Execução

- ♦ Logo, uma variável é acessível ou não dependendo se ela está no bloco em questão ou num bloco envolvente a este. Assim, dado um nível lexicográfico i , os blocos acessíveis têm nível $i, i-1, \dots, 1$. Então, na execução, o endereço de base da área de dados sempre ocupará a mesma posição no display. Com isso, dada uma variável X , podemos simplificar seu endereçamento para:

$$\text{endereço}(X) = \text{display}[\text{nível}(X)] + \text{deslocamento}(X)$$

O par de endereço torna-se, então: (n, d) onde n é o nível lexicográfico e d , o deslocamento dentro da área de dados.

Armazenamento de dados

- elementares
- arrays
- "templates"

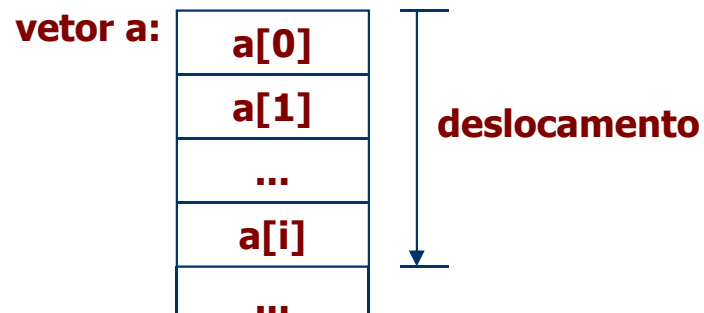
Armazenamento de dados elementares

- ♦ Os tipos de dados devem ser mapeados em células de memória. Para alguns dados (inteiros, ponto flutuante, por exemplo) esse mapeamento pode ser simples. Para outros tipos (strings, por exemplo) várias células de memória podem ser necessárias (e até mesmo uma organização, como estruturas encadeadas).

Organização de Memória em Tempo de Execução

Armazenamento de arrays

- vetores
- matrizes
- arrays multi-dimensionais
- ♦ Vetores: Em geral, os vetores são armazenados em células contíguas de forma ascendente ou descendente.



A indexação $a[i]$, por exemplo, é obtida somando-se (ou subtraindo-se) i ao (do) endereço de base do vetor (caso o limite inferior do vetor seja zero).

Se a declaração do vetor for: $a[\text{INF}:\text{SUP}]$, então o deslocamento de $a[i]$ será $(i - \text{INF})$, ou seja: $\text{endereço}(a[i]) = \text{endereço}(a[\text{INF}]) + \text{deslocamento}$.

- ♦ Matrizes: Usualmente, uma matriz é armazenada (por linha, por exemplo) em posições contíguas.
Seja a matriz $M[1:m, 1:n]$. Vamos imaginar que a matriz é armazenada linha por linha, ou seja:

Organização de Memória em Tempo de Execução

matriz M:

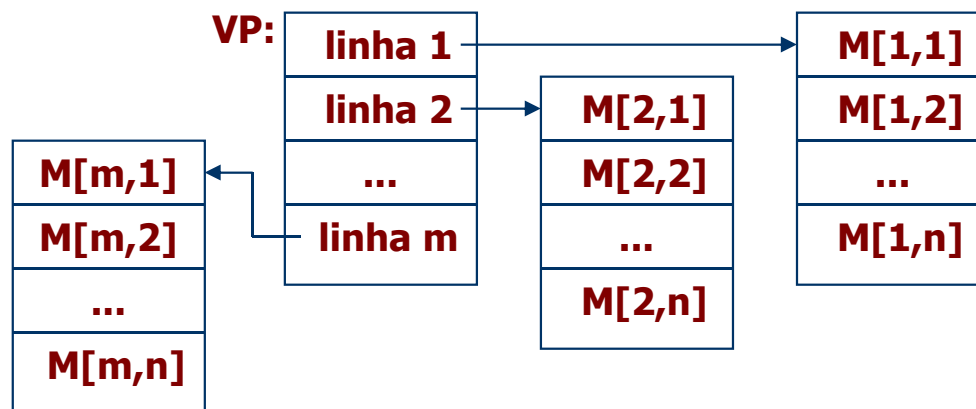
M[1,1]
M[1,2]
...
M[1,n]
M[2,1]
...

$$\text{endereço}(M[i,j]) = \text{endereço}(M[1,1]) + (i-1)*n + (j-1)$$

ou seja:

$$\text{endereço}(M[i,j]) = \underbrace{(\text{endereço}(M[1,1]) - n - 1)}_{\text{constante}} + i*n + j$$

- Assim, para endereçar um elemento da matriz serão necessárias 1 multiplicação e 2 somas.
- Um outro método usado para armazenar matrizes é reservar uma área de dados separada para cada linha e usar um vetor de ponteiros para as áreas de dados das linhas:



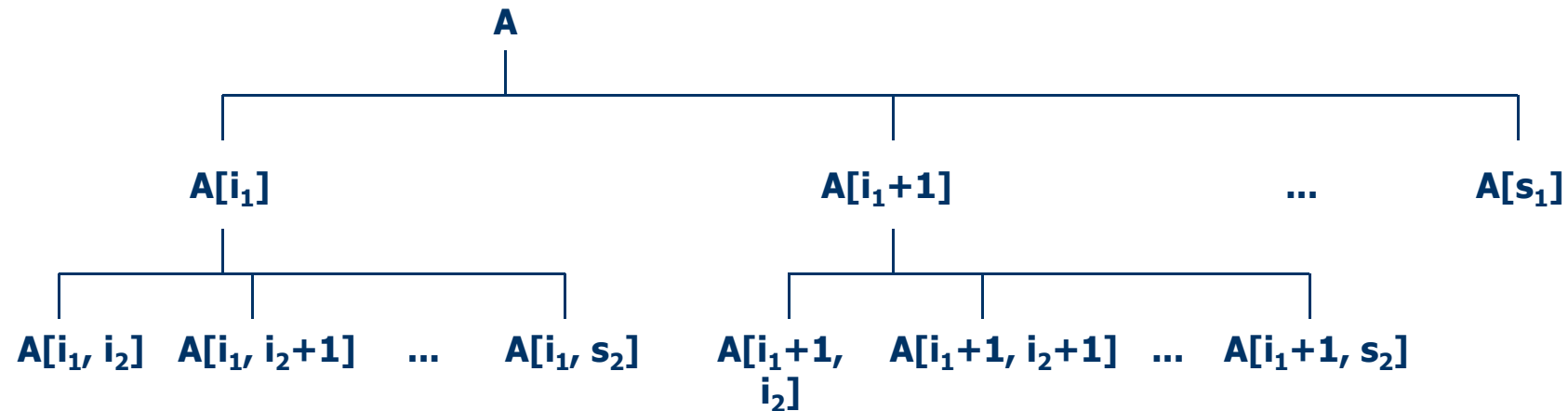
Nesse caso: $\text{endereço}(M[i,j]) = \text{conteúdo}(\text{endereço}(\text{VP}) + (i-1)) + (j-1)$

ou seja: nenhuma multiplicação é necessária (outra razão para usar esse método é que nem todas as linhas precisam estar na memória ao mesmo tempo).

Organização de Memória em Tempo de Execução

- Arrays multi-dimensionais

Seja $A[i_1:s_1, i_2:s_2, \dots, i_n:s_n]$. Se fixarmos um elemento teremos um "array" de dimensão $n-1$:



- Para referenciar um elemento $A[k_1, \dots, k_n]$ seja: $d_j = (s_j - i_j + 1)$ ($j = 1, \dots, n$). O início do sub-array $A[k_1, *, \dots, *]$ é:

$$\text{baseloc} + (k_1 - i_1) * d_2 * \dots * d_n$$

onde baseloc é o endereço do primeiro elemento: $A[i_1, i_2, \dots, i_n]$.

O início de $A[k_1, k_2, *, \dots, *]$ é, portanto:

$$\text{baseloc} + (k_1 - i_1) * d_2 * \dots * d_n + (k_2 - i_2) * d_3 * \dots * d_n$$

Portanto, a posição de $A[k_1, k_2, \dots, k_n]$ será:

$$\text{baseloc} + (k_1 - i_1) * d_2 * \dots * d_n + (k_2 - i_2) * d_3 * \dots * d_n + (k_n - i_n)$$

Organização de Memória em Tempo de Execução

- ♦ Posição de $A[k_1, k_2, \dots, k_n]$:

$$\text{baseloc} + (k_1 - i_1) * d_2 * \dots * d_n + (k_2 - i_2) * d_3 * \dots * d_n + (k_n - i_n)$$

Nessa expressão temos uma parte constante (que pode ser calculada uma só vez):

$$\text{conspart} = \text{baseloc} - ((\dots (i_1 * d_2 + i_2) * d_3 + i_3) * d_4 + \dots + i_{n-1}) * d_n + i_n)$$

e uma parte variável:

$$\text{varpart} = (\dots ((k_1 * d_2 + k_2) * d_3 + k_3) * d_4 + \dots + k_{n-1}) * d_n + k_n$$

que pode ser calculada como:

$$\text{varpart} = k_1$$

$$\text{varpart} = \text{varpart} * d_2 + k_2$$

...

$$\text{varpart} = \text{varpart} * d_n + k_n$$

- ♦ Portanto, dada a natureza iterativa do cálculo de varpart, não é difícil gerar código para o cálculo de endereços de variáveis indexadas usando esse esquema, qualquer que seja a dimensão do array.

Organização de Memória em Tempo de Execução

"Templates" (molde)

- ♦ Se o compilador conhece todos os atributos das variáveis, então é possível gerar todo o código necessário para referenciar essas variáveis. Entretanto, em certas linguagens de programação, algumas informações referentes aos dados são conhecidas somente em tempo de execução.
- ♦ Para resolver esse problema, o compilador aloca espaço para um molde que irá descrever os atributos a serem determinados em tempo de execução. Para os "arrays", esses moldes são conhecidos como vetor dopado ("dope vector") ou vetor de informação. Esse vetor terá um tamanho fixo, conhecido em tempo de compilação, na área de dados à qual o "array" estiver associado. O espaço para o próprio "array" só poderá ser alocado em tempo de execução. Assim, quando ativa-se o bloco em que o "array" está declarado, os limites do "array" são calculados, o vetor dopado é preenchido, o espaço de memória é calculado e o "array" é alocado.
- ♦ Formato do vetor dopado para $A[i_1:s_1, i_2:s_2, \dots, i_n:s_n]$:

i_1	s_1	d_1
i_2	s_2	d_2
...
i_n	s_n	d_n
n	conspart	
baseloc		

Notar que n (dimensão do array) s_1, s_2, \dots, s_n (limites superiores para os valores dos índices) são incluídos no vetor dopado para ser possível gerar código de verificação de "índice inválido".

Organização de Memória em Tempo de Execução

Correspondência entre parâmetros formais e parâmetros reais

- chamada por referência
- chamada por valor
- argumento "dummy"

Chamada por referência

- ◆ Quando um parâmetro é passado por referência (também conhecido como chamada por endereço) seu endereço é fornecido ao procedimento (a área de dados do procedimento, portanto, deverá ter espaço para armazenar esse endereço). No caso do parâmetro real ser uma expressão, deve ser reservada uma localização temporária (para o valor da expressão), que é então passada como parâmetro.
- ◆ **Exemplo:**

```
procedure swap(integer x,y);  
begin  
  integer temp;  
  temp := x;  
  x := y;  
  y := temp;  
end;
```

Chamada por referência de swap(i,a[i]):

- colocar os endereços de i e de a[i] em posições conhecidas (exemplo: arg1 e arg2)
- temp := conteúdo(arg1)
- i := conteúdo(arg2)
- conteúdo(arg2) := temp

Organização de Memória em Tempo de Execução

Chamada por valor

- ◆ Nesse caso, o procedimento chamado possui uma localização especial para o parâmetro formal em sua área de dados. Em tempo de execução é passado o valor do parâmetro real.
- ◆ **Exemplo:**

```
procedure swap(integer x,y);  
begin  
  integer temp;  
  temp := x;  
  x := y;  
  y := temp;  
end;
```

Chamada por valor de swap(i,a[i]):

- colocar os valores de i e de a[i] em posições conhecidas (exemplo: t1 e t2)
- temp := t1
- t1 := t2
- t2 := temp

Argumento "dummy"

- ◆ Se um argumento constante é passado por endereço a um procedimento p(x) (p(3), por exemplo) e se no procedimento p existe uma atribuição a x (por exemplo, x := x + 5), então a execução do procedimento irá alterar o valor da constante!
- ◆ Uma forma de evitar o problema é a linguagem não aceitar constantes passadas por referência.
- ◆ Outra forma é reservar uma posição temporária (conhecida como argumento "dummy") para conter o argumento real. O endereço dessa posição temporária será então passado para o procedimento.

Organização de Memória em Tempo de Execução

Observação: Quando o parâmetro real é um:

- ♦ nome de array: é fornecido ao procedimento o endereço do vetor dopado;
- ♦ nome de procedimento: é fornecido o endereço da primeira instrução executável.

Esquema de Randell & Russell (linguagens estruturadas em bloco)

Neste esquema:

- não há diferença entre bloco "puro" e bloco de procedimento.
- o "display" precisa ser feito tanto na entrada como na saída do bloco.
- formato da área de dados:

dados e parâmetros, inclusive endereço de retorno
cadeia dinâmica
cadeia estática

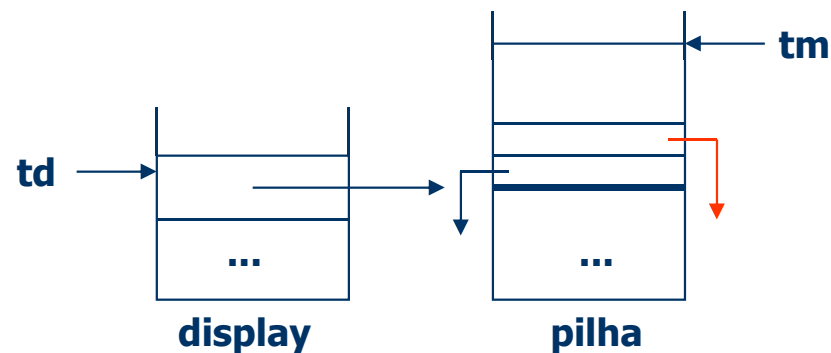
- ♦ Durante a execução do programa-objeto é necessário armazenar a posição inicial de cada área de dados empilhada. Para o bloco que está sendo usado (ou seja, o bloco que está no topo da pilha) essa posição é armazenada em PP ("procedure pointer").
- ♦ A cadeia estática forma a lista de blocos acessíveis (essa lista aparece duplicada no "display") e a cadeia dinâmica forma a lista de todas as áreas de dados empilhadas (usada para atualizar PP)

Organização de Memória em Tempo de Execução

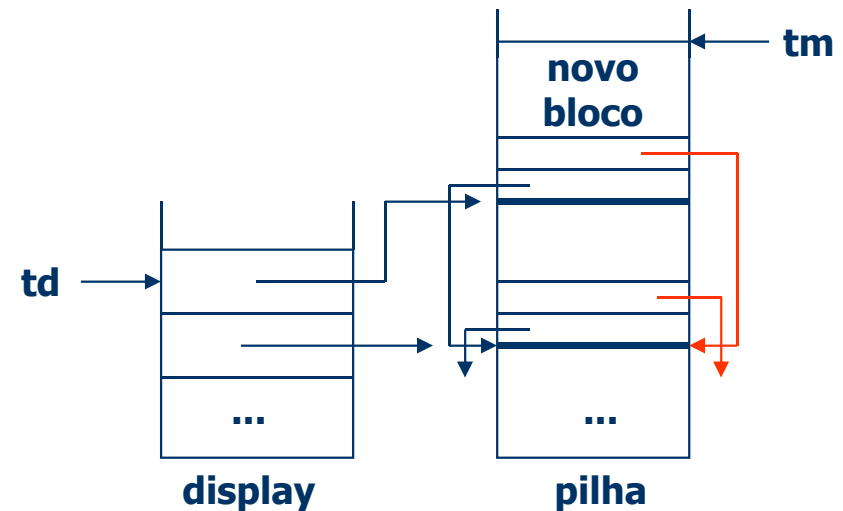
Entrada de bloco. Seja n o nível lexicográfico do bloco.

```
cd := display[td]
td := n;
display[td] := tm;
mem[tm] := display[td-1];    (* cadeia estática *)
mem[tm+1] := cd;            (* cadeia dinâmica *)
tm := tm + tamanho(bloco);
```

antes de entrar no bloco



após a área de dados ter sido alocada



Organização de Memória em Tempo de Execução

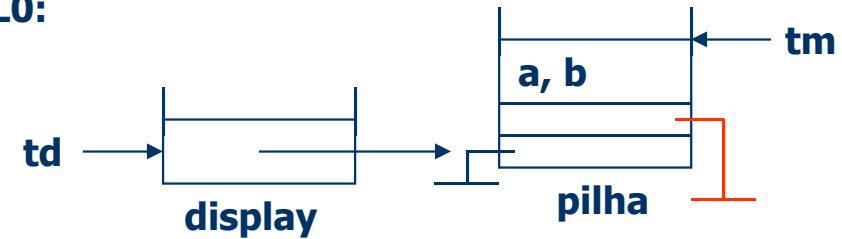
Saída de bloco.

```
tm := display[td]  
restaurar o display (por meio da cadeia estática);
```

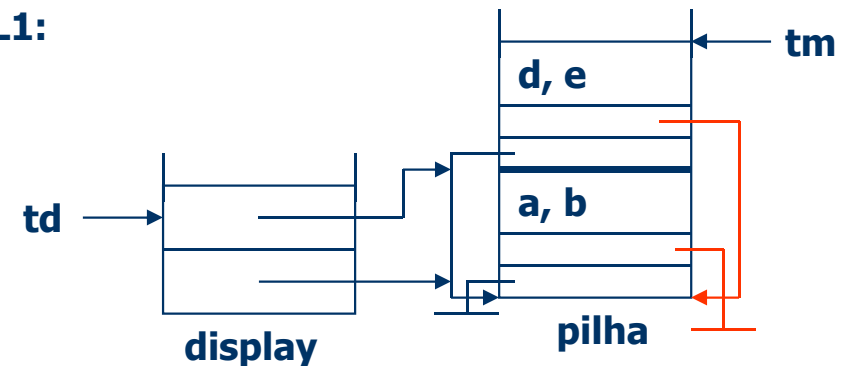
Exemplo:

```
(1) begin  
    int a,b[1:10];  
    procedure Q(integer x);  
(2)  begin  
        integer c[1:x];  
        ...  
(2)  end;  
        ...  
        L0: ...  
        ...  
(2)  begin  
        integer d[1:5],e;  
        ...  
        L1: ...  
        ...  
        L2: Q(10);  
        ...  
        L3: ...  
        ...  
(2)  end;  
        ...  
        L4: ...  
        ...  
(1) end;
```

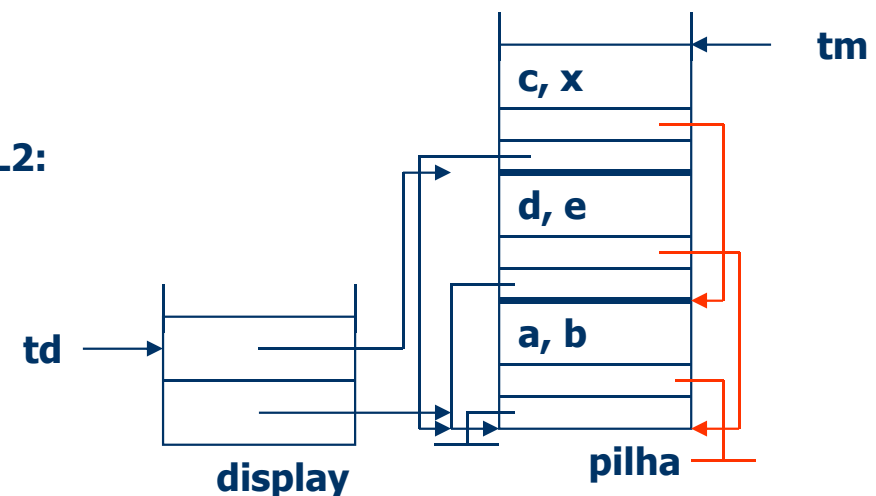
L0:



L1:



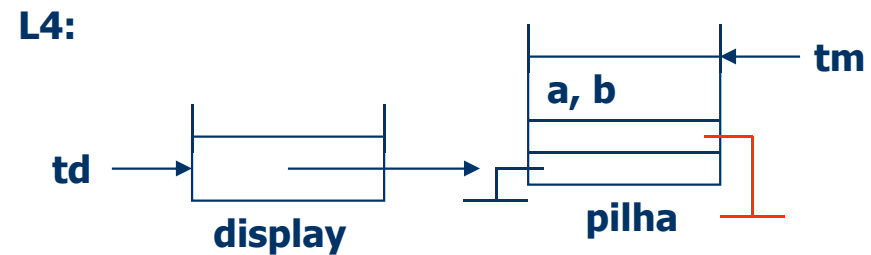
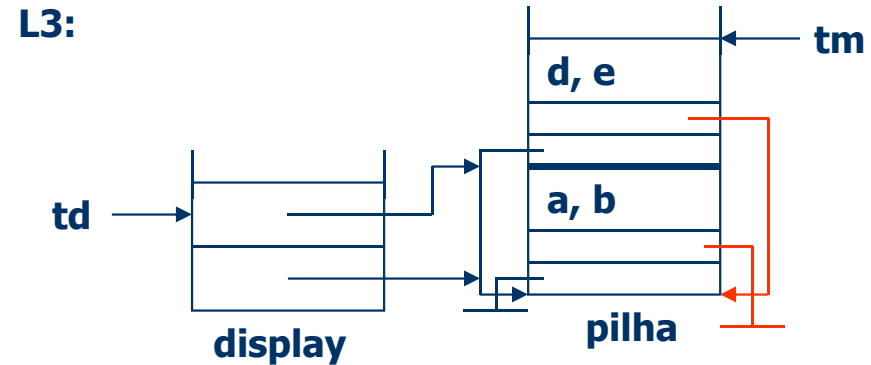
L2:



Organização de Memória em Tempo de Execução

Exemplo:

```
(1) begin
    int a,b[1:10];
    procedure Q(integer x);
(2)   begin
        integer c[1:x];
        ...
(2)   end;
    ...
    L0: ...
    ...
(2)   begin
        integer d[1:5],e;
        ...
        L1: ...
        ...
        L2: Q(10);
        ...
        L3: ...
        ...
(2)   end;
    ...
    L4: ...
    ...
(1) end;
```



Observe que nesse esquema de memória, desvios para fora de blocos ("bad goto's") acarretam chamadas ao procedimento de saída de bloco.