7

Objetos e Estruturas Dinâmicas de Dados

Nos capítulos anteriores, em particular nos capítulos 3 e 6, foram abordados objetos de dados escalares ou matriciais, definidos como tipos intrínsecos ou derivados. Todos esses objetos possuem duas características em comum: eles armazenam algum tipo de valor de dado e todos são *estáticos*, no sentido de que o número e os tipos desses objetos são declarados antes da execução do programa. Neste caso, ao se rodar o programa executável, sabe-se de antemão o espaço de memória necessário para tanto.

A memória total destinada ao programa é dividida em diversos segmentos, de acordo com a natureza dos objetos de dados e do espaço de instruções contidas no programa. Especificamente, nas arquiteturas que são tipicamente empregadas atualmente, o espaço destinado aos dados estáticos fica em duas regiões com baixo endereçamento de memória adjacentes: o *data segment* (para dados inicializados) e o *bss* (*basic service set*) *segment* (para dados não inicializados).[1] Ambos os segmentos ocupam um valor fixo de memória.

Um outro segmento existente é o *stack*, o qual contém intervalos variáveis de memória destinados a armazenar as instruções das rotinas chamadas pelo programa e os dados passados e declarados estaticamente nas mesmas. Uma vez que o processamento é retornado ao ponto de onde a rotina foi invocada, esse espaço de memória é liberado.

Porém, existe um outro tipo de objeto de dados que é *dinâmico*, no sentido de que não se conhece de antemão o número e/ou a natureza desses objetos; essas informações somente serão conhecidas no decorrer da execução do programa. Os objetos que são criados dinamicamente durante a execução do programa são armazenados em um outro segmento de memória denominado *heap*, o qual também permite a destinação de intervalos variáveis de memória para os dados dinâmicos, com a posterior liberação desse intervalo quando os objetos forem destruídos.

Um exemplo imediato da necessidade de objetos dinâmicos de memória consiste na interface de um experimento. O equipamento de medida monitora constantemente um determinado processo físico, gerando dados experimentais em intervalos regulares de tempo. Se o tempo total de experimento não é conhecido de antemão, então o número total de amostras medidas pelo equipamento também não é conhecido. Nesta situação, é ideal que os dados obtidos do experimento sejam primeiro armazenados em uma estrutura dinâmica de dados.

O Fortran oferece três recursos básicos para armazenar objetos dinâmicos de dados: *matrizes alocáveis*, *ponteiros* (*pointers*) ou *objetos alocáveis*. Esses recursos serão apresentados neste capítulo.

7.1 Matrizes alocáveis

Matrizes alocáveis são objetos compostos, todos do mesmo tipo e espécie, criados dinâmicamente. O Fortran fornece tanto matrizes alocáveis quanto matrizes automáticas, ambos os tipos sendo matrizes dinâmicas. Usando matrizes alocáveis, é possível alocar e dealocar espaço de memória conforme o necessário. O recurso de matrizes automáticas permite que matrizes locais em uma função ou subrotina tenham forma e tamanho diferentes cada vez que a rotina é invocada. Matrizes automáticas são discutidas na seção 9.2.11.2.

[1] Maiores informações a respeito da organização da memória podem ser obtidos em https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709, https://www.geeksforgeeks.org/memory-layout-of-c-program/ ou https://en.wikipedia.org/wiki/Data_segment.

7.1.1 DEFINIÇÃO E USO BÁSICO

Uma matriz alocável é declarada na linha de declaração de tipo de variável com o atributo ou a declaração ALLOCATABLE. O posto da matriz deve também ser declarado com a inclusão dos símbolos de dois pontos ":", um para cada dimensão da matriz. Ou seja, a matriz tem o posto fixo, mas a sua forma é arbitrária. Este tipo de objeto é denominado uma **matriz de forma deferida** (*deferred-shape array*). Por exemplo, a matriz real de duas dimensões A é declarada como alocável através da declaração:

```
REAL, DIMENSION(:,:), ALLOCATABLE :: A
```

a qual empregou o atributo ALLOCATABLE ou por meio da declaração ALLOCATABLE:

```
REAL :: A
ALLOCATABLE :: A(:,:)
```

Estas declarações não alocam espaço de memória à matriz imediatamente, como acontece com as declarações usuais de matrizes. O status da matriz nesta situação é *not currently allocated*, isto é, correntemente não alocada. Espaço de memória é dinamicamente alocado durante a execução do programa, logo antes da matriz ser utilizada, usando-se o comando ALLOCATE. Este comando especifica os limites da matriz, seguindo as mesmas regras definidas na seção 6.1. Um exemplo de aplicação deste comando, usando expressões inteiras na alocação é:

```
ALLOCATE (A(0:N+1,M))
```

sendo as variáveis inteiras escalares N e M previamente determinadas. Como se pode ver, esta estratégia confere uma flexibilidade grande na definição de matrizes ao programador.

O espaço alocado à matriz com o comando ALLOCATE pode, mais tarde, ser liberado com o comando DEALLOCATE. Este comando requer somente nome da matriz previamente alocada. Por exemplo, para liberar o espaço na memória reservado para a matriz A, o comando fica

```
DEALLOCATE (A)
```

Os comandos ALLOCATE e DEALLOCATE são empregados não somente para alocar espaço de memória para matrizes, mas também para objetos alocáveis em geral, e as suas formas mais gerais serão apresentadas na seção 7.3. Especificamente para matrizes alocáveis, as instruções têm as formas

```
ALLOCATE (<lista-alloc> [, STAT= <status>] [, ERRMSG= <err-mess>])
DEALLOCATE (<lista-alloc> [, STAT= <status>] [, ERRMSG= <err-mess>])
```

onde <lista-alloc> é uma lista de alocações, sendo que cada elemento da lista tem a forma

```
<obj-allocate>(<lista-ext-array>)
```

com <obj-allocate> sendo o nome da matriz e <lista-ext-array> a lista das extensões de cada dimensão da matriz na forma

```
[<limite-inf>:]<limite-sup>
```

sendo <limite-inf> e <limite-sup> expressões inteiras escalares. Como o usual, se <limite-inf> está ausente, é tomado <limite-inf>= 1.

Se a palavra-chave STAT= está presente no comando, <status> status deve ser uma variável inteira escalar, a qual recebe o valor zero se o procedimento do comando ALLOCATE/DEALLOCATE foi bem sucedido ou um valor positivo se houve um erro no processo (*e. g.*, se não houver espaço suficiente na memória para alocar a matriz). Se o especificador STAT= não estiver presente e ocorrer um erro no processo, o programa é abortado.

A palavra-chave ERRMSG também se refere ao status da alocação/dealocação, sendo que o campo <err-msg> contém uma variável de caractere escalar padrão. Se ERRMSG= estiver presente e um erro ocorrer na execução do comando, o sistema gera uma mensagem de erro que é gravada em <err-msg>. Essa mensagem pode ser utilizada para controle de erros na execução do programa.

Um exemplo curto, mas incluindo todas as opções seria:

```
CHARACTER(LEN= 200) :: ERR_MESS ! Ou outro comprimento apropriado
INTEGER :: N, ERR_STAT
REAL, DIMENSION(:), ALLOCATABLE :: X
ALLOCATE(X(N), STAT= ERR_STAT, ERRMSG= ERR_MESS)
IF(ERR_STAT > 0)THEN
   PRINT*, 'Alocação de X falhou:', TRIM(ERR_MESS) ! Função TRIM(): seção 8.7.2
END IF
```

É possível alocar ou dealocar mais de uma matriz simultaneamente na <lista-alloc>. Um breve exemplo do uso destes comandos seria:

```
REAL, DIMENSION(:), ALLOCATABLE :: A, B
REAL, DIMENSION(:,:), ALLOCATABLE :: C
INTEGER :: N
...
ALLOCATE (A(N), B(2*N), C(N,2*N))
B(:N)= A
B(N+1:)= SQRT(A)
DO I= 1,N
   C(I,:)= B
END DO
DEALLOCATE (A, B, C)
```

Matrizes alocáveis satisfazem a necessidade frequente de declarar uma matriz tendo um número variável de elementos. Por exemplo, pode ser necessário ler variáveis, digamos tam1 e tam2, e então declarar uma matriz com tam1 × tam2 elementos:

```
INTEGER :: TAM1, TAM2
REAL, DIMENSION (:,:), ALLOCATABLE :: A
INTEGER :: STATUS
...
READ*, TAM1, TAM2
ALLOCATE (A(TAM1,TAM2), STAT= STATUS)
IF (STATUS > 0)THEN
... ! Comandos de processamento de erro.
END IF
... ! Uso da matriz A.
DEALLOCATE (A)
...
```

No exemplo acima, o uso do especificador STAT= permite que se tome providências caso não seja possível alocar a matriz, por alguma razão.

Como foi mencionado anteriormente, uma matriz alocável possui um *status de alocação* (*allocation status*). Quando a matriz é declarada, mas ainda não alocada, o seu status é *unallocated* ou *not currently allocated*. Quando o comando ALLOCATE e bem sucedido, o status da matriz passa a *alocado* (*allocated)*; uma vez que ela é dealocada, o seu status retorna a *not currently allocated*. Assim, o comando ALLOCATE somente pode ser usado em matrizes não correntemente alocadas, ao passo que o comando DEALLOCATE somente pode ser usado em matrizes alocadas; caso contrário, ocorre um erro.

É possível verificar se uma matriz está ou não correntemente alocada usando-se a função intrínseca ALLOCATED.[2] Esta é uma função lógica com um argumento, o qual deve ser o nome de uma matriz alocável. Se a matriz está alocada, a função retorna .TRUE., caso contrário, retorna .FALSE.. Usando-se esta função, comandos como os seguintes são possíveis:

```
IF (ALLOCATED(A)) DEALLOCATE (A)
```

ou

```
IF (.NOT. ALLOCATED(A)) ALLOCATE (A(5,20))
```

[2] Ver seção 8.2.

Finalmente, há duas restrições no uso de matrizes alocáveis:

1. Matrizes alocáveis não podem ser argumentos mudos de uma rotina e devem, portanto, ser alocadas e dealocadas dentro da mesma unidade de programa (ver capítulo 9).
2. O resultado de uma função não pode ser uma matriz alocável (embora possa ser uma matriz).

O programa-exemplo a seguir ilustra o uso de matrizes alocáveis.

```
program testa_aloc
implicit none
integer, dimension(:,:), allocatable :: b
integer :: i,j,n= 2

print*, "Valor inicial de n:",n
allocate (b(n,n))
b= n
print*, "Valor inicial de B:"
print"(2(i2))", ((b(i,j), j= 1,n), i= 1,n)
deallocate (b)
n= n + 1
print*, "Segundo valor de n:",n
allocate (b(n,n))
b= n
print*, "Segundo valor de B:"
print"(3(i2))", ((b(i,j), j= 1,n), i= 1,n)
deallocate (b)
n= n + 1
print*, "Terceiro valor de n:",n
allocate (b(n+1,n+1))
b= n + 1
print*, "Terceiro valor de B:"
print"(5(i2))", ((b(i,j), j= 1,n+1), i= 1,n+1)
end program testa_aloc
```

Sugestões de uso & estilo para programação

Uma prática útil para evitar erros em códigos extensos: sempre inclua a cláusula `STAT=` nas alocações e sempre verifique o status. Desta maneira, é possível tomar providências caso ocorra algum erro de alocação.

7.1.2 Recursos avançados de alocação/realocação de matrizes

Eventualmente torna-se necessário alterar a forma de uma matriz previamente alocada. O Fortran oferece mais de uma alternativa para realizar essa tarefa.

7.1.2.1 A rotina intrínseca MOVE_ALLOC

A sub-rotina intrínseca `MOVE_ALLOC` realiza a transferência da alocação de uma matriz para outra, ambas previamente alocadas e com os mesmos tipos, espécies e postos. Se não for empregada em coarrays, a rotina é *pura*.[3]

A chamada desta subrotina é realizada pela instrução

```
CALL MOVE_ALLOC(FROM, TO)
```

onde:

[3] A respeito de rotinas puras, ver seção 9.2.17.

FROM é uma matriz alocável de qualquer tipo e espécie. No âmbito da subrotina, FROM possui a *intenção* (*intent*) INOUT.[4]

TO é uma matriz alocável com os mesmos tipo, espécie e posto de FROM. No âmbito da subrotina, TO possui a intenção OUT.

Após a chamada e execução da subrotina, o status de alocação, a geometria e os elementos de TO são aqueles que FROM possuia anteriormente e FROM se torna dealocado. Se TO tem o atributo TARGET, qualquer ponteiro que estava anteriormente associado com FROM passa a se associar com TO; caso contrário, esses ponteiros se tornam indefinidos.

A subrotina MOVE_ALLOC pode ser usada para minimizar o número de operações de cópia requeridas quando se torna necessário alterar o tamanho de uma matriz, uma vez que somente uma operação de cópia da matriz original é necessária, ao invés de duas como aconteceria com o emprego de laços.

O programa abaixo ilustra o uso desta subrotina.

```
! Este programa usa MOVE_ALLOC para aumentar o tamanho
!  do vetor X, mantendo os seus elementos iniciais.
program tes_Move_Alloc
implicit none
integer :: i, n= 5
real, dimension(:), allocatable :: x, y

allocate (x(n), y(2*n))          ! Y é maior que X
! Testa os status de alocação
print'(2(a,g0))', 'X está alocado? ',allocated (X),'  Tamanho de X: ',size(x)
print'(2(a,g0))', 'Y está alocado? ',allocated (Y),'  Tamanho de Y: ',size(y)

call random_number(x)   ! Atribui valores a X
print '(/,a)', 'Elementos originais de X:'
print'(5(g0,x))', x

do i= 1, 5         ! Copia todo X nos primeiros elementos de Y
   y(i) = x(i)     ! Esta é a única cópia de dados necessária
end do
print '(/,a)', 'Elementos de Y:'
print'(10(g0,x))', y

call move_alloc (y, x)   ! X agora tem o dobro do tamanho original, Y foi
                         ! dealocado e os elementos de X foram preservados.

print'(/,a)', 'Após MOVE_ALLOC:'
print'(2(a,g0))', 'X está alocado? ',allocated (X),'  Tamanho de X: ',size(x)
print'(a,g0)', 'Y está alocado? ', allocated (Y)
print'(/,a)', 'Novo vetor X é:'
print'(10(g0,x))', x
end program tes_Move_Alloc
```

7.1.2.2 ALOCAÇÃO/REALOCAÇÃO AUTOMÁTICA

O recurso de *alocação/realocação automática* (ou *alocação na atribuição*) foi introduzido mais recentemente e torna a manipulação de matrizes alocáveis ainda mais conciso e automatizado. Com este recurso, praticamente não há mais a necessidade do comando ALLOCATE ou da subrotina MOVE_ALLOC, exceto em casos muito particulares.

O exemplo a seguir tipifica usos deste recurso:

```
INTEGER, DIMENSION(5) :: A= [ 1, 2, 3, 4, 5]
INTEGER, DIMENSION(:), ALLOCATABLE :: B
```

[4] Acerca da qualificação INTENT, ver seção 9.2.5.

```
REAL, DIMENSION(10) :: RA
REAL, DIMENSION(:), ALLOCATABLE :: RB
B= [ -2, -1, 0 ]  ! B é automaticamente alocado e definido
B= [ B, A ]       ! Agora, B= [-2, -1, 0, 1, 2, 3, 4, 5]
B= [ B, 6, 7, 8 ] ! Agora, B= [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
CALL RANDOM_NUMBER(RA) ; RA= RA - 0.5
RB= PACK(RA, RA > 0.0) ! RB é alocado com os valores positivos de RA
```

Na última linha acima, foi empregada a função `PACK` (seção 8.14.2), a qual irá selecionar somente os elementos positivos de `RA`, automaticamente alocar `RB` com o número de elementos correto e depositar no mesmo esses valores positivos.

O programa 7.1 mostra usos mais sofisticados deste recurso. O recurso da alocação/realocação automáticas fornece uma solução possível ao problema colocado no início desta seção: o armazenamento de um número previamente indeterminado de dados. Esse recurso se torna ainda mais poderoso quando aplicado em matrizes alocáveis de tipos derivados, com componentes também alocáveis, o que é o assunto discutido na seção 7.1.3.

Listagem 7.1: Exemplos de uso de alocação/realocação automáticas

```
program tes_reallocation_on_assignment
use iso_fortran_env, only: dp => real64
implicit none
logical, dimension(:), allocatable :: tes
real(kind= dp), dimension(:), allocatable :: xd, yd, hd
complex(kind= dp), dimension(:), allocatable :: zd
integer :: n, i

write(*, fmt= '(a)', advance= 'no')'Enter n= ' ; read(*,*)n
allocate(yd(n), hd(n))
call random_number(yd)
xd= yd
tes= xd > 0.5_dp
print'(/,a)', 'Array xd:'
write(*, fmt= '(g0, x)') (xd(i), i= 1, n)
print'(/,2(a,g0))', 'size of tes: ', size(tes), '  # of x > 1/2: ', count(tes)
print'(a)', 'Array tes:'
print*, tes

call random_number(yd) ; call random_number(hd)
zd= cmplx(yd, hd, kind= dp)
print'(/,a)', 'Array zd:'
write(*, fmt= '(a,g0,a,g0,a,x)') ('(', zd(i)%re, ',', zd(i)%im, ')', i= 1, n)

xd= pack(xd, tes)
print'(2/,a,g0)', 'Array xd packed: size= ', size(xd)
write(*, fmt= '(g0, x)') (xd(i), i= 1, size(xd))
zd= pack(zd, tes)
print'(/,a,g0)', 'Array zd packed: size= ', size(zd)
write(*, fmt='(a,g0,a,g0,a,x)') ('(',zd(i)%re,',', zd(i)%im,')',i= 1,size(zd))
end program tes_reallocation_on_assignment
```

O uso do recurso da alocação/realocação é padrão na linguagem. Para evitar que o código executável rode com este recurso, é necessário utilizar opções de compilação:

GFortran: `-fno-realloc-lhs`[5]

Intel® fortran: `-assume norealloc_lhs`[6]

Por exemplo, compilando o programa 7.1 com a linha

[5] Ref: https://gcc.gnu.org/onlinedocs/gfortran/Code-Gen-Options.html. Ver também opção `-Wrealloc-lhs`.

[6] Ref: https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-assume.

```
user@machine|dir>gfortran -fno-realloc-lhs tes_auto_reallocation.f90
```

irá gerar um executável, mas a execução do mesmo será abortada, com uma mensagem de erro na primeira tentativa de realizar a alocação automática:

```
user@machine|dir>./a.out
Enter n= 10
Program received signal SIGSEGV: Segmentation fault - invalid memory
    reference.
Backtrace for this error:
#0  0x7f090babbcea
#1  0x7f090babae75
#2  0x7f090b75feff
#3  0x401646
#4  0x40222f
#5  0x7f090b74bf32
#6  0x40117d
#7  0xffffffffffffffff
Segmentation fault (core dumped)
```

7.1.3 ALOCAÇÃO DINÂMICA DE COMPONENTES DE TIPOS DERIVADOS

Obviamente, também é possível declarar matrizes alocáveis de tipos derivados. Adicionalmente, os componentes do tipo derivado também podem ser alocáveis, com o benefício adicional do recurso de alocação na atribuição ou realocação. Isto permite criar objetos dinâmicos com complexidade grande o suficiente para suprir a maior parte dos casos que requerem tais objetos. Um cuidado que precisa ser tomado se refere a inicializações-padrão de componentes do tipo derivado. Uma tentativa de inicialização de um componente alocável não faz sentido e não é permitida pelo padrão.

O programa 7.2 ilustra o uso de matrizes alocáveis de tipos derivados com componentes alocáveis.

Listagem 7.2: Alocação dinâmica de tipos derivados e seus componentes.

```
program tes_derived_type_realloc
implicit none
integer, parameter :: nloop= 5
integer :: i
real, dimension(nloop) :: vr2
type :: t1
   real, dimension(:), allocatable :: vr1
end type t1
type(t1) :: st1                          ! Escalar estático do tipo t1
type(t1), dimension(nloop) :: vt2        ! Vetor estático do tipo t1
type(t1), dimension(:), allocatable :: vt1 ! Vetor dinâmico do tipo t1

print'(a)', 'Allocação automática de componente de um tipo &
           &derivado escalar estático:'
call random_number(vr2)
print'(a,*(g0,x))', 'vr2: ', vr2
st1= t1(vr1= vr2)
print'(a,*(g0,x))', 'st1: ', st1%vr1

print'(/,a)', 'Primeiro laço: alocação automática de componente &
             &de vetor estático de tipo derivado:'
do i= 1, nloop
   call random_number(vr2)
   print'(a,g0,a,*(g0,x))', 'i= ', i, '  vr2: ', vr2
```

```
   vt2(i)= t1(vr1= vr2)
end do
do i= 1, nloop
   print'(a,g0,a,*(g0,x))', 'i= ', i, '  vt2: ', vt2(i)%vr1
end do

print'(/,a)', 'Segundo laço: alocação automática de componente &
             &de vetor dinâmico de tipo derivado:'
allocate(vt1(0)) ! Primeiro, aloque vt1 como uma matriz de tamanho zero
do i= 1, nloop
   call random_number(vr2)
   print'(a,g0,a,*(g0,x))', 'i= ', i, '  vr2: ', vr2
   vt1= [vt1, t1(vr1= vr2)] ! Executa realocação automática
end do

do i= 1, nloop
   print'(a,g0,a,*(g0,x))', 'i= ', i, '  vt1: ', vt1(i)%vr1
end do
end program tes_derived_type_realloc
```

Sugestões de uso & estilo para programação

Quando matrizes alocáveis são empregadas, é conveniente sempre dealocar as mesmas assim que não mais são necessárias. Isto porque matrizes alocáveis, como objetos dinâmicos, são alocadas no *heap*, o qual não é liberado automaticamente com uma saída de rotina, por exemplo. Isto cria os chamados *vazamentos de memória* (*memory leaks*), que podem facilmente esgotar toda a memória disponível.

7.2 Ponteiros (*pointers*)

Um ponteiro (*pointer*) é um outro tipo dinâmico de dado. Até agora, todos os objetos de dados considerados, mesmo as matrizes alocáveis, armazenam valores de algum dos tipos intrínsecos ou de tipos derivados. Um ponteiro, por outro lado, armazena um *endereço de memória*, o qual corresponde à localização da região da memória do computador onde o dado está contido. Como subprogramas também ocupam espaços na memória do computador, um ponteiro também pode ser empregado para armazenar o endereço de um subprograma. Ponteiros também podem ser empregados em conjunto com tipos derivados para estabelecer estruturas dinâmicas de dados mais gerais do que matrizes alocáveis. Exemplos dessas estruturas dinâmicas são listas encadeadas e árvores, entre outras. Ponteiros também oferecem em certos casos métodos para realizar manipulações de um número grande de dados de forma eficiente, pois permitem reduzir o número de operações de leitura/escrita desses dados na memória, diminuindo assim o tempo de processamento.

A figura 7.1 é uma ilustração típica da diferença entre um objeto de dados usual e um ponteiro. Sejam `var = 100` uma variável inteira padrão e o seu valor. Supondo que um inteiro padrão ocupe 4 bytes = 32 bits de memória, o endereço de memória `0x123` que aparece acima da variável corresponde à localização do primeiro destes 32 bits. O ponteiro, `ptr` então "aponta" para `var`, a qual se tornou o "alvo" do ponteiro; mais especificamente, `ptr` passou a armazenar o endereço do primeiro bit ocupado por `var`, como está ilustrado na figura. Como `var` ocupa ao todo 32 bits de memória, é necessário que o programa também tenha essa informação. Por isso, em diversas linguagens tais como Fortran, C ou C++, os ponteiros também têm diferentes tipos, os quais podem ser os tipos intrínsecos ou outros tipos derivados. Portanto `ptr` é um ponteiro do tipo inteiro que armazena o endereço inicial de `var`. Como o ponteiro possui um valor (um endereço), ele também precisa ocupar uma região da memória (o endereço de `ptr` é `0x155`).

A descrição apresentada acima sobre a diferença entre variáveis e ponteiros é aplicável para qualquer linguagem que trabalhe com ponteiros, em particular para C/C++. Contudo, os ponteiros em Fortran podem armazenar também outras informações a respeito de seus alvos, além de seus endereços. Como será visto posteriormente, ponteiros para matrizes também armaze-

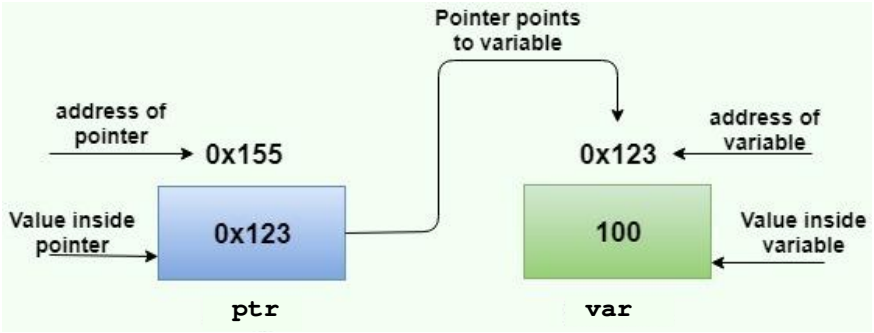

Figura 7.1: O ponteiro `ptr` armazena o endereço de memória da variável inteira `var`.

nam informações a respeito da geometria das mesmas. Além disso, a maneira como o Fortran manipula ponteiros e os emprega em expressões ou atribuições é também ligeiramente diferente da estratégia adotada no C/C++. Um ponteiro somente pode "apontar" para um objeto estático de dados se este último for explicitamente declarado como um **alvo** (*target*).

Pode-se perguntar então: "por que objetos estáticos de dados devem ser declarados como alvos em Fortran, uma vez que outras linguagens, como C/C++, não precisam disso?" A razão está nos métodos usuais empregados por compiladores para a otimização do código. Quando se solicita ao compilador que este gere o código executável mais eficiente possível, o compilador pode realizar modificações próprias no código-fonte criado pelo programador. Essas otimizações envolvem diversas estratégias como o desenrolamento de laços (*loop unwrapping*), substituição de partes do código por algoritmos mais eficientes, inclusão explícita de subprogramas (*inlining*), entre outros. Nesses processos, eventualmente variáveis que foram declaradas e usadas pelo programador são eliminadas ou movidas para outras regiões da memória. Se um ponteiro for previamente associado a uma dessas variáveis modificadas pelo processo de otimização, a referência armazenada no mesmo se torna inválida e o seu uso em expressões irá provavelmente gerar resultados incorretos. Este é um exemplo de criação de um *ponteiro pendente* (*dangling pointer*). Por esta razão, se o objeto estático for declarado como um possível alvo de ponteiros, o compilador será informado que o mesmo não pode ser eliminado ou ter seu endereçamento alterado.

7.2.1 DECLARAÇÕES BÁSICAS DE PONTEIROS E ALVOS

Ponteiros e alvos de ponteiros podem ser respectivamente declarados empregando as palavras-chave `POINTER` e `TARGET` como atributos ou em declarações próprias. Por exemplo, o ponteiro `ptr` da figura 7.1 pode ser declarado um ponteiro inteiro usando o atributo:

```
INTEGER, POINTER :: PTR
```

ou via as declarações:

```
INTEGER :: PTR
POINTER :: PTR
```

Já a variável `var` da figura 7.1 deve ter sido declarada

```
INTEGER, TARGET :: VAR= 100 ! A atribuição do valor pode ocorrer posteriormente
```

ou

```
INTEGER :: VAR
TARGET :: VAR
```

Ponteiros somente podem ser *associados* a alvos que tenham o *atributo de alvo*, adquirido de algumas das duas formas anteriores. Um mesmo objeto não pode ter ambos os atributos de ponteiro e alvo.

Ponteiros também podem ser criados com objetos compostos e/ou matriciais. Por exemplo,

```
REAL, POINTER, DIMENSION(:) :: X, Y
COMPLEX, POINTER, DIMENSION(:,:) :: Z
```

declara dois ponteiros para vetores do tipo real (`X` e `Y`) e um ponteiro para uma matriz complexa de posto 2 (`Z`). Da mesma forma como acontece durante a declaração de matrizes alocáveis, ponteiros para matrizes devem ser declarados como *matrizes de forma deferida*, isto é, o posto da matriz é determinado, mas não sua forma, o que fica evidenciado pela presença dos caracteres ":" no atributo/declaração `DIMENSION`. Os exemplos abaixo ilustram alvos adequados para os ponteiros definidos acima:

```
REAL, TARGET :: A(100), B(200)         ! Para X e Y
COMPLEX, TARGET, DIMENSION(10,20) :: C ! Para Z
```

Também é possível definir ponteiros para e alvos de tipos derivados. Por exemplo, dado o tipo `aluno_t` definido no programa 3.6, pode-se definir:

```
TYPE(ALUNO_T), POINTER, DIMENSION(:) :: ALUNOS_P
TYPE(ALUNO_T), TARGET, ALLOCATABLE, DIMENSION(:) :: ALUNOS_V
```

Observa-se que `ALUNOS_V` possui tanto o atributo `TARGET` quanto `ALLOCATABLE`, o que permite que o tamanho do vetor seja determinado em tempo de execução. Uma vez que `ALUNOS_V` foi declarado com o atributo `TARGET`, todos os seus componentes automaticamente possuem o mesmo atributo e podem ser individualmente associados a outros ponteiros. Além disso, componentes individuais de tipos derivados podem ter atributos de ponteiro ou alvo, mesmo que a estrutura completa não tenha esse atributo.

A forma geral das declarações de ponteiros e alvos será apresentada posteriormente.

7.2.2 Atribuições de ponteiros e seus usos em expressões

Esta seção explora como ponteiros podem ser associados a alvos e como podem ser subsequentemente empregados em expressões e atribuições.

7.2.2.1 Status de associação

O **status de associação de um ponteiro** indica se o mesmo está ou não apontando para um alvo. Este status pode ser:

Indefinido (*undefined*). Status assumido pelo ponteiro no momento da declaração, como nos exemplos acima.

Definido (*defined*). Status adquirido quando o ponteiro foi associado com algum objeto com atributo de alvo.

Desassociado (*disassociated*). Quando um ponteiro previamente associado perde este status por algum dos mecanismos apresentados abaixo e não é associado com nenhum outro alvo.

Mesmo que um ponteiro seja associado a um alvo (tornando-se assim definido), o alvo *per se* pode ter o status de definido ou indefinido.

Um ponteiro somente pode ser empregado se ele estiver associado a um alvo. Uma ocorrência de uso de um ponteiro indefinido nem sempre pode ser detectada pelo compilador e isto poderá acarretar em um erro durante a execução do programa. Existem circunstâncias nas quais um ponteiro previamente associado pode ser tornar indefinido. Um exemplo disso ocorre com o uso da rotina `MOVE_ALLOC` (seção 7.1.2.1); como foi mencionado, se o objeto identificado pelo qualificador `TO=` tiver o atributo de alvo, então qualquer ponteiro que esteja previamente associado com o objeto em `FROM=` terá sua associação transferida junto com o objeto de dados. Em caso contrário, o ponteiro se torna indefinido.

Para minimizar a possibilidade de uso de um ponteiro indefinido, pode-se empregar a função intrínseca `ASSOCIATED`,[7] a qual retorna um valor lógico indicando o status de associação do ponteiro. Existem duas maneiras de empregar a função `ASSOCIATED`: para testar a associação do ponteiro a *qualquer* alvo ou a um alvo em particular. Por exemplo, para testar se o ponteiro `ptr` está associado a algum alvo, emprega-se

[7] Ver seção 8.2.

```
STATUS= ASSOCIATED(PTR)
```

sendo STATUS uma variável lógica. Se ptr já estiver associado a var, o resultado será STATUS= .TRUE., caso contrário será .FALSE.. Por outro lado, para testar se ptr está associado ao inteiro var, usa-se

```
STATUS= ASSOCIATED(PTR, VAR)
```

se ptr estiver associado a algum outro alvo, o resultado será STATUS= .FALSE..

Contudo, a função ASSOCIATED somente pode testar dois status: definido ou desassociado. Se o ponteiro estiver no estado indefinido, o resultado será ambíguo (pois o padrão não determina qual deve ser o resultado neste caso), o que poderá levar a erros de processamento. Para evitar essa possibilidade, existem dois recursos:

1. Uso do função intrínseca NULL().[8] Esta função confere o status de desassociado a um ponteiro, independente de seu status inicial. O uso deste recurso nos exemplos acima é:

   ```
   INTEGER, POINTER :: PTR => NULL()
   REAL, POINTER, DIMENSION(:) :: X => NULL(), Y => NULL()
   COMPLEX, POINTER, DIMENSION(:,:) :: Z => NULL()
   TYPE(ALUNO_T), POINTER, DIMENSION(:) :: ALUNOS_P => NULL()
   ```

2. Uso do comando NULLIFY. Este comando possui a sintaxe

   ```
   NULLIFY(<lista-obj-pointer>)
   ```

   A execução deste comando confere a todos os objetos na <lista-obj-pointer> o status de desassociado. Esses objetos devem ter o atributo de ponteiro e não podem ser interdependentes, para que o processador possa possa realizar as desassociações em qualquer ordem. Como se trata de um comando, a instrução NULLIFY deve vir após todas as declarações. Para os exemplos acima, o uso pode ser:

   ```
   INTEGER, POINTER :: PTR
   REAL, POINTER, DIMENSION(:) :: X, Y
   COMPLEX, POINTER, DIMENSION(:,:) :: Z
   TYPE(ALUNO_T), POINTER, DIMENSION(:) :: ALUNOS_P
   ⋮ ! Outras declarações
   NULLIFY(PTR, X, Y, Z, ALUNOS_P)
   ```

Sugestões de uso & estilo para programação

Recomenda-se sempre realizar a desassociação de todos os ponteiros, via NULL() ou NULLIFY(...), antes de qualquer uso dos mesmos. Esta prática possibilita a realização de testes de status e evita a ocorrência de erros devidos ao uso de ponteiros indefinidos.

7.2.2.2 ASSOCIAÇÃO DE PONTEIROS COM OBJETOS DE DADOS

Para que um ponteiro possa ser empregado, ele primeiro precisa primeiro estar associado a um objeto de dados, ou seja, deve conter, no mínimo, o endereço de memória do objeto. Isto pode ocorrer de duas maneiras: ou pela *atribuição de ponteiro* (*pointer assignment*) a um objeto que tenha atributos de alvo ou ponteiro ou pela alocação dinâmica de um novo espaço na memória.

ATRIBUIÇÃO DE PONTEIRO. O primeiro processo (*atribuição de ponteiro*) é realizada com o operador "=>" pelas instruções

```
<ponteiro> => <alvo> ou <ponteiro 1> => <ponteiro 2>
```

A atribuição pode ocorrer na inicialização do ponteiro:

[8] Ver seção 8.16.

```
REAL, TARGET :: X        ! Pode ou não ser inicializado.
REAL, POINTER :: PX => X ! Ponteiro PX inicializado com o endereço de X
```

O ponteiro `PX` automaticamente adquire o status de *definido* e é inicializado com o endereço de `X`.

Ou a atribuição pode ocorrer ao longo da seção de instruções do código, onde diversas possibilidades podem ocorrer. O programa 7.3 mostra diversos processos de atribuições entre ponteiros.

Listagem 7.3: Ilustra diversos casos de associação de ponteiros para objetos escalares.

```
1  program tes_pointer
2  implicit none
3  real, target :: x1= 10.0, x2= -17.0
4  real, pointer :: px1 => null(), px2 => null()

6  print'(2(a,g0))', 'px1 associado? ', associated(px1), &
7                   '  px2 associado? ', associated(px2)
8  px1 => x1 ! px1 está associado a x1.
9  print'(3(a,g0))', 'px1 associado? ', associated(px1), &
10                  '  px1= ', px1, '  x1= ', x1
11 px2 => x2 ! px2 está associado a x2.
12 print'(3(a,g0))', 'px2 associado? ', associated(px2), &
13                  '  px2= ', px2, '  x2= ', x2

15 ! Alterações nas associações de ponteiros.
16 px2 => px1
17 print'(/,2(a,g0))', 'px1= ', px1, '  px2= ', px2
18 px1 => x2
19 print'(2(a,g0))', 'px1= ', px1, '  px2= ', px2
20 px2 => px1
21 px1 => null() ! Ou: nullify(px1)
22 print'(2(a,g0))', 'px1 associado? ', associated(px1), '  px2= ', px2
23 end program tes_pointer
```

Neste programa, os ponteiros `px1` e `px2` são inicializados com o status de desassociados. Em seguida, as instruções `px1 => x1` e `px2 => x2` realiza as atribuições indicadas nas linhas 8 e 11. Estas atribuições estão representadas na figura 7.2. Isto significa que `px1` adquiriu o endereço de `x1`, enquanto que `px2` adquiriu o endereço de `x2`.

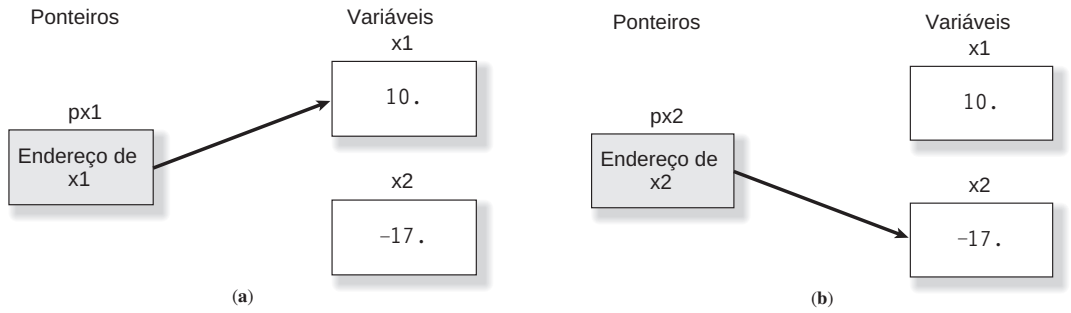

Figura 7.2: atribuições iniciais entre as variáveis ***x1*** e ***x2*** com os ponteiros ***px1*** e ***px2*** no programa 7.3. No painel (a), ***px1*** adquire o endereço de ***x1*** (linha 8), enquanto que no painel (b), ***px2*** adquire o endereço de ***x2*** (linha 11).

Posteriormente, na linha 16, ocorre uma atribuição do tipo `<ponteiro 1> => <ponteiro 2>`. Neste tipo de atribuição, o que ocorre é que o `<ponteiro 1>` adquire o valor do `<ponteiro 2>`, ou seja, o endereço do alvo do `<ponteiro 2>` se este estiver definido ou o seu status em caso contrário. Após a execução desta instrução, *ambos os ponteiros apontam diretamente e de forma independente ao mesmo alvo.* Na linha 16, portanto, o que ocorre é que `px2` copia o endereço armazenado em `px1`, o qual é o endereço de `x1`; ambos os ponteiros apontam agora para o

mesmo alvo, como está ilustrado no painel (a) da figura 7.3. Em seguida, na linha 18, `px1` passa a apontar para `x2`, mas a associação de `px2` não foi alterada (continua associado a `x1`). Esta situação está ilustrada no painel (b) da figura 7.3.

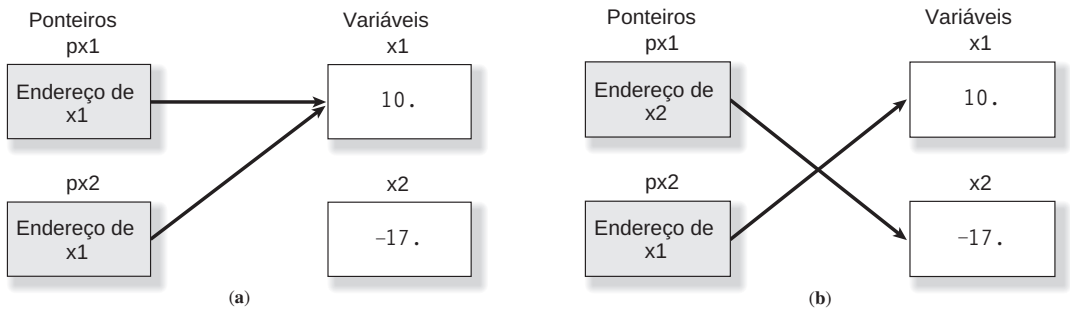

Figura 7.3: atribuições de ponteiros no programa 7.3. Painel (a): *px2* passa a apontar para *x1* (linha 16). Painel (b): *px1* passa a apontar para *x2* (linha 18).

Finalmente, nas linhas 20 e 21, primeiro `px2` adquire o endereço em `px1` (o endereço de `x2`) e logo em seguida `px1` é desassociado. Como os ponteiros retêm os endereços dos alvos de forma independente, a desassociação de `px1` não altera o estado e a associação de `px2`. No programa observa-se também que os valores de `x1` e `x2` podem ser acessados tanto de forma direta quanto de forma indireta, através dos ponteiros associados às variáveis. Expressões e atribuições envolvendo ponteiros serão discutidas na seção 7.2.2.3.

Quando o ponteiro é de um tipo composto, como matrizes ou tipos derivados, além do endereço do primeiro objeto contido no alvo, um ponteiro em Fortran também armazena informações a respeito da estrutura do objeto composto. Estas informações a respeito do alvo que ficam armazenadas no ponteiro são denominadas em alguns textos como *descritores de ponteiros.*

Quando o alvo é uma matriz, um ponteiro pode adquirir o descritor correspondente à matriz completa ou a uma seção da mesma. A flexibilidade do mecanismo de atribuição de ponteiros para matrizes no Fortran é exemplificada no programa 7.4, no qual estão definidos uma matriz real de posto 2 `table` e um vetor inteiro `sunspot_number`, o qual contém o número de manchas solares observadas entre os anos de 1700 e 2018. O programa define também um ponteiro para matriz real de posto 2 `window` e dois ponteiros para vetores inteiros `ps1` e `psn2`.

Listagem 7.4: Exemplos de associação de ponteiros a matrizes.

```
1  program tes_array_pointer_assignment
2  implicit none
3  integer :: m= 50, n= 130, p= 30, q= 85
4  integer, dimension(1700:2018), target :: sunspot_number
5  real, dimension(200,100), target :: table
6  integer, dimension(:), pointer :: psn1, psn2
7  real, dimension(:,:), pointer :: window

9  ! Associações entre table e window.
10 print'(2(a,*(g0,x)),a)', 'Limites de table:  Inferiores: [ ', &
11                         lbound(table), ']  Superiores: [', ubound(table), ']'
12 window => table
13 print'(2(a,*(g0,x)),a)', 'Limites de window (1): Inferiores: [ ', &
14                         lbound(window), ']  Superiores: [', ubound(window), ']'
15 window => table(m:n, p:q)
16 print'(2(a,*(g0,x)),a)', 'Limites de window (2): Inferiores: [ ', &
17                         lbound(window), ']  Superiores: [', ubound(window), ']'

19 ! Associações entre sunspot_number e psn1 e psn2.
20 print'(/,2(a,*(g0,x)))', 'Limites de sunspot_number: Inferior: ', &
21                           lbound(sunspot_number), '  Superior: ',  &
22                           ubound(sunspot_number)
23 psn1 => sunspot_number
24 print'(2(a,*(g0,x)))', 'Limites de psn1: Inferior: ', lbound(psn1), &
```

```
25                                          '  Superior: ', ubound(psn1)
26 psn2 => sunspot_number(1901:2000)
27 print'(2(a,*(g0,x)))', 'Limites de psn2 (1): Inferior: ', lbound(psn2), &
28                                             '  Superior: ', ubound(psn2)
29 psn2(1901:) => sunspot_number(1901:2000)
30 print'(2(a,*(g0,x)))', 'Limites de psn2 (2): Inferior: ', lbound(psn2), &
31                                             '  Superior: ', ubound(psn2)
32
33 psn2 => sunspot_number(::50)
34 print'(2(a,*(g0,x)))', 'Limites de psn2 (2): Inferior: ', lbound(psn2), &
35                                             '  Superior: ', ubound(psn2)
36 end program tes_array_pointer_assignment
```

O primeiro conjunto de instruções no programa 7.4 (entre as linhas 10 e 17) ilustra diversas atribuições entre as matrizes `table` e `window`. Primeiramente, as funções intrínsecas `LBOUND` e `UBOUND` (seção 8.13) reafirmam os limites das dimensões da matriz `table`. Em seguida, na linha 12 o ponteiro `window` é associado à matriz `table` completa, obtendo assim todos os descritores desta última (endereço inicial e informações da geometria), o que fica demonstrado na impressão dos limites de `window`, realizada na linha 13. Porém, na linha 15 o ponteiro `window` se torna associado a uma seção da matriz `table`, com uma geometria determinada pelas variáveis inteiras `m`, `n`, `p` e `q`. Neste caso, a sintaxe da atribuição segue as mesmas regras para seções de matrizes discutidas na seção 6.3. Na linha 16 a geometria resultante de `window` é impressa na saída padrão, onde se observa que o ponteiro armazenou os descritores para os elementos em 81 linhas (entre as linhas 50 e 130) e 56 colunas (entre colunas 30 e 85) de `table`, exatamente como é determinado pela álgebra de seções de matrizes. Nota-se contudo que os limites de `window` partem de 1 em todas as dimensões, o que é o comportamento esperado na definição das funções `LBOUND` e `UBOUND`. Porém, caso seja necessário ou adequado, é possível alterar também os limites registrados no ponteiro. Este mecanismo será ilustrado a seguir, com os ponteiros `psn1` e `psn2`.

Entre as linhas 20 e 35 ocorrem diversas atribuições entre o vetor `sunspot_number` e os ponteiros para vetores. Na linha 23, `psn1` adquire os descritores do vetor completo. Nota-se que os limites de `sunspot_number` são automaticamente transferidos para `psn1`. Já na linha 26, `psn2` passa a ser um atalho para o número de manchas solares observadas somente durante o século XX. Nota-se que o limite inferior de `psn2` é 1 e o superior é 100, de maneira análoga ao que ocorreu com `window` na linha 15. Porém, na linha 29 a mesma atribuição é realizada, mas agora a sintaxe `psn2(1901:)` determina que a atribuição dos limites de `psn2` ocorra com o limite interior igual a 1901. Ou seja, as regras de seções de matrizes determinadas por tripletos de subscritos podem ser empregadas também em ambos os lados de uma atribuição de ponteiros para matrizes, desde que sejam realizadas de uma forma consistente. Finalmente, na linha 33, `psn2` passa a apontar a uma seção de `sunspot_number` que varre os índices do limite inferior ao superior, porém com um passo igual a 50. Neste caso, `psn2` passou a ser um atalho para o número de manchas solares observadas de 1700 a 2018 a cada 50 anos, totalizando 7 registros.

Os mecanismos de atribuições de ponteiros para matrizes exemplificados acima também podem ser empregados entre objetos de diferentes postos. Neste conjunto de instruções:

```
INTEGER :: N
REAL, DIMENSION(:), ALLOCATABLE, TARGET :: VETOR_BASE
REAL, POINTER :: MATRIZ(:,:), DIAGONAL(:)
ALLOCATE(VETOR_BASE(N*N))
MATRIZ(1:N, 1:N) => VETOR_BASE
DIAGONAL => VETOR_BASE(::N+1)
```

o `VETOR_BASE`, após a alocação de memória, passará a ocupar um intervalo contíguo de espaços de memória contendo N^2 objetos reais. Após a sua atribuição, o ponteiro `MATRIZ` conterá os descritores dos elementos de `VETOR_BASE` na forma de uma matriz $N \times N$, sendo que os descritores serão armazenados em `MATRIZ` na ordem dos elementos de matriz (seção 6.6.2), de forma que esses elementos podem ser referenciados via `MATRIZ` de uma maneira contígua. Finalmente, o vetor `DIAGONAL` é um atalho para os elementos da diagonal de `MATRIZ`.

Quando os objetos envolvidos são tipos derivados, a variadade de diferentes situações é ainda maior. Um objeto de um tipo derivado pode ser declarado com o atributo `TARGET`, em cuja situação todos os seus componentes automaticamente têm o mesmo atributo. Ponteiros desse tipo

derivado podem se associar a uma estrutura-alvo por completo ou ponteiros dos tipos dos componentes da estrutura podem se associar com estes últimos de forma individual. O exemplo abaixo ilustra essa situação:

```
INTEGER :: M, N, P
TYPE :: F3V_T
   REAL :: U, DU(3), D2U(3,3)
END TYPE F3V_T
TYPE(F3V_T), DIMENSION(:,:,:), ALLOCATABLE, TARGET :: MU
REAL, POINTER, DIMENSION(:,:,:) :: PMU
ALLOCATE(MU(M, N, P))
PMU => MU%U
```

Neste exemplo, `F3V_T` é um tipo derivado que armazena o valor de uma função de três variáveis, suas três derivadas parciais de primeira ordem e suas derivadas segundas. A matriz `MU` armazena esses valores em uma grade com `M` $\times$ `N` $\times$ `P` pontos, ao passo que o ponteiro `PMU` possibilita um acesso rápido somente aos valores da função nos pontos de grade. A geometria de `PMU` é igual à geometria de `MU`.

Um componente de um tipo derivado também pode ser declarado de uma forma recursiva como um ponteiro para o mesmo tipo. Toda essa flexibilização permite a definição de novas estruturas complexas de dados, algumas das quais serão discutidas na seção 7.4. Em particular, nessa seção será discutida a construção de uma *lista encadeada*, formada a partir da definição de um tipo derivado semelhante a:

```
TYPE :: ENTRY
   INTEGER :: INDEX
   REAL :: VALOR= 2.0
   REAL, DIMENSION(:), POINTER :: VETOR
   TYPE(ENTRY), POINTER :: NEXT => NULL()
END TYPE ENTRY
```

Antes de discutir o segundo método de associação de ponteiros, é importante mais uma vez frisar que um ponteiro definido adquire os descritores do objeto de dados no momento da associação. Se o ponteiro estiver associado a um objeto alocável, pode ocorrer que posteriormente o estado de alocação do objeto seja alterado via o mecanismo de realocação automática (seção 7.1.2.2). Neste caso, a associação do ponteiro deve ser renovada, para que os novos descritores sejam registrados. No exemplo abaixo, se `V` é um vetor alocável e `PV` um ponteiro a um vetor, o último passo na sequência de instruções

```
PV => V      ! PV adquire os descritores de V
V= [ V, 1 ] ! A extensão de V foi alterada
PV => V      ! PV adquire os novos descritores de V
```

se faz necessário porque a forma de `V` foi alterada via alocação automática. Se houver outros ponteiros associados com `V` ou com uma seção de `V`, estes também deverão ter suas associações renovadas.

Sugestões de uso & estilo para programação

Se ponteiros estão associados a objetos alocáveis, é importante que alterações nos estados de alocação desses objetos sejam realizadas com a rotina `MOVE_ALLOC`, pois assim todos os ponteiros associados a esses objetos terão os descritores automaticamente atualizados.

ALOCAÇÃO DE MEMÓRIA PARA PONTEIRO. O segundo tipo de processo de atribuição de ponteiro ocorre pela alocação dinâmica de um novo espaço de memória, que não estava previamente ocupado por algum objeto de dados. Este procedimento ocorre por meio da instrução `ALLOCATE`, como é exemplificado nas instruções

```
INTEGER :: M, N
REAL, POINTER :: PX1, PV(:), PM(:,:)
ALLOCATE(PX1, PV(10), PM(M, N))
```

A execução da instrução ALLOCATE acima reserva automaticamente espaços de memória e cria objetos de dados sem nomes (*i. e.*, anônimos), com os tipos, espécies e geometrias adequados. Os ponteiros passam a ter status de definidos e os descritores dos correspondentes espaços de memória são atribuídos aos ponteiros, sem haver a necessidade de existência prévia de objetos com atributos de alvo. Uma vez que esses espaços de memória são anônimos, eles somente podem ser acessados pelos ponteiros. Estes espaços de memória podem ser posteriormente liberados por meio da instrução DEALLOCATE.

A forma geral dos comandos ALLOCATE e DEALLOCATE para ponteiros é igual às suas formas para matrizes alocáveis, apresentadas na seção 7.1.1. No caso particular de ponteiros, se o procedimento de alocação de memória falhar, o ponteiro retém o seu status de associação. As ações executadas quando as palavras-chave STAT= e ERRMSG= estão presentes são as mesmas que ocorrem no caso de matrizes alocáveis.

Um ponteiro pode se associar a um novo alvo com qualquer um dos dois métodos discutidos, mesmo se este já estava previamente associado com um alvo. Neste caso a associação prévia é removida. Contudo, se o alvo prévio consistir em uma área de memória reservada para o ponteiro por um comando ALLOCATE, a mesma se tornará inacessível, exceto se um outro ponteiro foi também associado a ela.

Quando o espaço de memória associado a um ponteiro via o comando ALLOCATE não for mais necessário, este espaço pode ser liberado com a instrução DEALLOCATE. Se a operação for bem sucedida, o status do ponteiro muda automaticamente para desassociado. Se a operação falhar, o ponteiro mantém o seu status de associação; contudo, se a cláusula STAT= não for empregada neste caso, a execução do programa será interrompida. É importante ressaltar que o comando DEALLOCATE não deve ser empregado se o ponteiro tiver sido associado a um objeto de dados via atribuição (com o operador =>), como pode ocorrer com objetos estáticos ou matrizes alocáveis.

Exemplos de uso dos comandos ALLOCATE e DEALLOCATE com ponteiros para a construção de estruturas de dados mais complexas serão apresentados na seção 7.4.

7.2.2.3 Uso de ponteiros em expressões ou atribuições

Um ponteiro associado com um objeto de dados contém o descritor desse objeto, isto é, o endereço do objeto na memória e informações sobre a geometria e conteúdo do mesmo, caso este seja um objeto composto. Contudo, um dos principais objetivos de um ponteiro consiste em usar essa informação como um atalho para o *valor* do objeto de dados. Para tanto, deve existir um mecanismo pelo qual esse valor se torne acessível a partir do ponteiro. O processo de acesso ao valor do objeto é denominado a **dereferenciação** (*dereferencing*) de um ponteiro e diferentes linguagens implementam esse mecanismo de diferentes maneiras. No Fortran, a dereferenciação é automática; isto é, o uso do nome do ponteiro em expressões ou atribuições intrínsecas (não em atribuições de ponteiro) automaticamente torna acessível o valor do objeto com o qual o ponteiro está associado. A dereferenciação automática já foi empregada nas instruções PRINT dos programas 7.3 e 7.4 quando os nomes dos ponteiros foram empregados para imprimir os valores das variáveis na tela.

Existem, portanto, duas maneiras distintas de transferir informações entre ponteiros via atribuições. Sendo p1 e p2 ponteiros, a *atribuição de ponteiro* p1 => p2 transfere para p1 o status de associação de p2 e o descritor de um eventual objeto de dados associado a p2; assim, ambos p1 e p2 servem de atalho ao mesmo objeto de dados, cujo valor não foi alterado. Contudo a *atribuição intrínseca* p1 = p2 não transfere o descritor de p2 para p1; o que ocorre é a cópia do valor do objeto associado a p2 para o objeto associado a p1, alterando assim o valor deste último objeto, enquanto que as associações dos ponteiros permanecem inalteradas. As regras usuais de expressões e atribuições discutidas no capítulo 4 continuam válidas neste caso.

O programa 7.5 exemplifica diversas instâncias de emprego de ponteiros em expressões e atribuições envolvendo tanto objetos escalares.

Listagem 7.5: Uso de ponteiros com objetos escalares.

```
1 program tes_pointer_att_scalar
2 implicit none
3 real, target :: x1= 11.0, x2= -25.5, x3
4 real, pointer :: p1 => null(), p2 => null(), p3 => null()
```

```
6  !Realiza associações iniciais: p1 aponta para x1, etc.
7  p1 => x1 ; p2 => x2 ; p3 => x3
8  p3= p1 + p2 ! O mesmo que x3 = x1 + x2
9  print'(a,g0)', 'Valor de x3:           ', x3
10 print'(a,g0)', 'Valor de x3 (via p3): ', p3

12 p2 => p1    ! Agora p2 aponta para x1
13 p3= p1 + p2 ! O mesmo que x3 = x1 + x1
14 print'(/,a,g0)', 'Novo valor de x3: ', x3

16 p3 = p1 ! Atribuição intrínseca: o mesmo que x3 = x1
17 print'(/,2(a,g0))', 'Valores de: x3= ', x3, '  p3= ', p3

19 !Realiza novas associações
20 x3 = 2*x1 + x2 ! p3 ainda está associado a x3
21 p2 => x2 ! Retorna à associação inicial
22 p3 => p2 ! p3 aponta agora para x2.  Ocorreu cópia de descritores, não de dados
23 print'(/,a)', '---> Cópia de descritores: p3 => p2, x3 /= x2:'
24 print'(3(a,g0))', 'Valores nas variáveis:   x1= ', x1, '  x2= ', x2, '  x3= ', x3
25 print'(3(a,g0))', 'Valores pelos ponteiros: p1= ', p1, '  p2= ', p2, '  p3= ', p3

27 !Realiza agora atribuição intrínseca
28 p3 => x3 ! Retorna à associação inicial
29 p3 = p2  ! Houve cópia de dados, não de descritores
30 print'(/,a)', '---> Cópia de dados: p3 = p2 -> x3 = x2:'
31 print'(3(a,g0))', 'Valores nas variáveis:   x1= ', x1, '  x2= ', x2, '  x3= ', x3
32 print'(3(a,g0))', 'Valores pelos ponteiros: p1= ', p1, '  p2= ', p2, '  p3= ', p3

34 end program tes_pointer_att_scalar
```

A execução deste programa gerou os seguintes resultados:

```
user@machine|dir>./a.out
Valor de x3:           -14.5000000
Valor de x3 (via p3): -14.5000000

Novo valor de x3: 22.0000000

Valores de: x3= 11.0000000  p3= 11.0000000

---> Cópia de descritores: p3 => p2, x3 /= x2:
Valores nas variáveis:   x1= 11.0000000  x2= -25.5000000  x3= -3.50000000
Valores pelos ponteiros: p1= 11.0000000  p2= -25.5000000  p3= -25.5000000

---> Cópia de dados: p3 = p2 -> x3 = x2:
Valores nas variáveis:   x1= 11.0000000  x2= -25.5000000  x3= -25.5000000
Valores pelos ponteiros: p1= 11.0000000  p2= -25.5000000  p3= -25.5000000
```

Os resultados que mostram somente cópias de descritores, sem alterações nos dados das variáveis foram gerados pelas instruções entre as linhas 20 e 25 do programa, ao passo que os resultados mostrando cópia de dados mas não de descritores foram gerados nas linhas 28 – 32.

Já o programa 7.6 exemplifica o emprego de ponteiros em expressões e atribuições envolvendo vetores:

Listagem 7.6: Uso de ponteiro com matrizes

```
1  program tes_pointer_att_vector
2  implicit none
3  integer :: i, n
4  real, target,  dimension(:), allocatable :: v1, v2
5  real, pointer, dimension(:) :: pv1 => null(), pv2 => null()

7  write(*, '(a)', advance= 'no') 'Número de elementos no vetor v1: ' ; read(*,*)n
8  allocate(v1(n))
```

```
9  call random_number(v1)
10 print'(a)', '---> Vetor v1 original:'
11 print'(5(g0,x))', v1
12 v2= v1 ! v2 é uma cópia de segurança de v1

14 !Primeiras associações:
15 pv1 => v1        ! pv1 aponta para v1 completo
16 pv2 => v1(2::3) ! pv2 aponta para uma seção de v1
17 print'(/,a)', '---> Dereferências do ponteiro pv1 (todo o vetor v1):'
18 print'(5(g0,x))', pv1
19 print'(/,a)', '---> Dereferências do ponteiro pv2 (seção de v1: v1(2), v1(5), etc):'
20 print'(5(g0,x))', pv2

22 !Cópia de descritores:
23 pv1 => pv2 ! pv1 adquire os descritores de pv2
24 print'(/,a)', '---> Novas dereferências do ponteiro pv1 (1):'
25 print'(5(g0,x))', pv1
26 print'(a)', 'Vetor v1 não é alterado, pois houve cópias de descritores: pv1 => pv2'

28 !Cópia de dados:
29 pv1 => v1 ! Retorna à associação inicial
30 pv1(:size(pv2)) = pv2 ! Agora há cópia de dados
31 print'(/,a)', '---> Novas dereferências do ponteiro pv1 (2) (cópias de dados):'
32 print'(5(g0,x))', pv1
33 print'(/,a)', '---> Novos valores do vetor v1 (1) (houve cópia de dados):'
34 print'(5(g0,x))', v1

36 v1= v2 !Usa v2 para retornar ao vetor v1 original
37 print'(/,a)', '---> Vetor v1 original:'
38 print'(5(g0,x))', v1

40 pv1(3:2 + size(pv2)) = pv2 ! Nova cópia de dados
41 print'(/,a)', '---> Novos valores do vetor v1 (2) (houve cópia de dados):'
42 print'(5(g0,x))', v1
43 end program tes_pointer_att_vector
```

Uma determinada execução deste programa gerou os resultados:

```
user@machine|dir>./a.out
Número de elementos no vetor v1: 10
---> Vetor v1 original:
0.649878383E-01 0.865801036 0.596078634 0.159959137 0.267309129
0.200994670 0.360674262E-01 0.353242338 0.916801214 0.403821468

---> Dereferências do ponteiro pv1 (todo o vetor v1):
0.649878383E-01 0.865801036 0.596078634 0.159959137 0.267309129
0.200994670 0.360674262E-01 0.353242338 0.916801214 0.403821468

---> Dereferências do ponteiro pv2 (seção de v1: v1(2), v1(5), etc):
0.865801036 0.267309129 0.353242338

---> Novas dereferências do ponteiro pv1 (1):
0.865801036 0.267309129 0.353242338
Vetor v1 não é alterado, pois houve cópias de descritores: pv1 => pv2

---> Novas dereferências do ponteiro pv1 (2) (cópias de dados):
0.865801036 0.267309129 0.353242338 0.159959137 0.267309129
0.200994670 0.360674262E-01 0.353242338 0.916801214 0.403821468

---> Novos valores do vetor v1 (1) (houve cópia de dados):
0.865801036 0.267309129 0.353242338 0.159959137 0.267309129
0.200994670 0.360674262E-01 0.353242338 0.916801214 0.403821468

---> Vetor v1 original:
0.649878383E-01 0.865801036 0.596078634 0.159959137 0.267309129
```

```
0.200994670  0.360674262E-01  0.353242338  0.916801214  0.403821468

---> Novos valores do vetor v1 (2) (houve cópia de dados):
0.649878383E-01  0.865801036  0.865801036  0.267309129  0.353242338
0.200994670  0.360674262E-01  0.353242338  0.916801214  0.403821468
```

O vetor `v1` foi alocado com 10 elementos escolhidos aleatoriamente. No intervalo de linhas 15 – 20 do programa, o ponteiro `pv1` é associado com o vetor inteiro, enquanto que o ponteiro `pv2` se torna associado à seção `[ v(2), v(5), v(8) ]`. Em seguida, nas linhas 23 – 26 o ponteiro `pv1` adquire os descritores de `pv2`, sem que ocorra alteração nos valores do vetor.

As expressões seguintes ilustram cópias de dados empregando ponteiros. Nas linhas 29 – 34 os três primeiros elementos de `v1` são alterados pela atribuição intrínseca `pv1(1:3) = pv2` ocorrida na linha 30. Finalmente, nas linhas 36 – 42 o vetor `v1` original é novamente alterado pela atribuição `pv1(3:5) = pv2`, a qual alterou os valores de `v1(3)` – `v1(5)`.

Para certas aplicações, o uso de ponteiros pode ter um impacto significativo na eficiência de um programa. Uma situação típica ocorre quando há necessidade de manipulações de matrizes com um número grande de elementos. Suponha que seja necessário trocar os nomes de duas matrizes `m1` e `m2`, ambas com dimensões $10^4 \times 10^4$. O código para realizar essa manipulação simples é:

```
1  program tes_swap_mat1
2  implicit none
3  integer, parameter :: nlc= 10000
4  real :: start_time, end_time
5  real, dimension(nlc, nlc) :: m1, m2, te

7  call random_number(m1)
8  call random_number(m2)

10 call cpu_time(start_time)
11 te= m1
12 m1= m2
13 m2= te
14 call cpu_time(end_time)
15 print'(a, g0)', 'Tempo de CPU (s): ', end_time - start_time

17 end program tes_swap_mat1
```

A troca propriamente dita ocorre nas linhas 11 – 13 e envolve o uso da matriz temporária `te`. O tempo de CPU necessário para realizar esta manipulação foi medido com a rotina `CPU_TIME` (seção 8.17.2), a qual adquire o valor do tempo (em segundos) do relógio do processador. O tempo total de processamento da troca das matrizes é então dado pela diferença `end_time - start_time`. Compilando este programa sem otimização, o tempo de processamento foi de aproximadamente 0,3 segundos, uma vez que ocorrem três processos de cópias de matrizes com dimensões $10^4 \times 10^4$.

O mesmo tipo de procedimento pode ser realizado de maneira muito mais rápida se forem realizadas cópias dos descritores das matrizes, ao invés dos dados. O programa abaixo implementa o mesmo procedimento com ponteiros:

```
1  program tes_swap_mat2
2  implicit none
3  integer, parameter :: nlc= 10000
4  real :: start_time, end_time
5  real, target,  dimension(nlc, nlc) :: m1, m2
6  real, pointer, dimension(:,:)      :: p1, p2, te

8  call random_number(m1)
9  call random_number(m2)

11 call cpu_time(start_time)
12 p1 => m1 ! p1 associado a m1
```

```
13 p2 => m2 ! p2 associado a m2
14 te => p1 ! te associado a m1
15 p1 => p2 ! p1 agora associado a m2
16 p2 => te ! p2 agora associado a m1
17 call cpu_time(end_time)
18 print'(a, g0)', 'Tempo de CPU (s): ', end_time - start_time

20 end program tes_swap_mat2
```

Agora, o tempo de processamento foi reduzido para cerca de $10^{-6} = 1,0\,\mu s$ apenas. Na verdade, a maior parte do tempo de execução do programa ocorre nas atribuições de valores às matrizes pela rotina `RANDOM_NUMBER`.

7.3 Objetos alocáveis

O conceito de um *objeto alocável* é uma extensão do conceito de uma matriz alocável para qualquer tipo de objeto, escalar ou composto. Observa-se que um objeto alocável, da mesma forma como ocorre com as matrizes alocáveis ou com ponteiros, são objetos que são declarados, mas para os quais não são reservados espaços de memória no momento em que o código é executado da primeira vez. Ao invés disso, o espaço de memória é criado dinâmicamente no momento em que se faz necessário e depois, quando não for mais necessário, esse espaço de memória pode ser disponibilizado de volta ao sistema operacional. Contudo, de forma distinta ao que ocorre com ponteiros, objetos alocáveis armazenam dados, ao invés de descritores.

As matrizes alocáveis, discutidas na seção 7.1, devem ser consideradas como casos particulares de objetos alocáveis em geral. Quando o uso de um objeto alocável se faz necessário, o espaço de memória necessário para armazenar o valor do dado pode ser criado pela instrução `ALLOCATE` e posteriormente liberado pela instrução `DEALLOCATE`. Se o espaço em memória a ser alocado se refere a um objeto escalar, a instrução `ALLOCATE` pode tomar a forma já discutida na seção 7.1.1. Contudo, para certos objetos compostos é necessário empregar a forma mais geral da instrução ALLOCATE que será apresentada mais adiante. Abaixo serão discutidos os diversos objetos alocáveis suportados pelo padrão atual da linguagem.

7.3.1 Objetos com parâmetros de tipo deferido

No padrão do Fortran, uma quantidade ou qualidade é *deferida* quando o seu valor não é conhecido no momento da compilação do código, mas sim durante a execução do programa por meio de um comando `ALLOCATE`, por uma atribuição intrínseca, por uma atribuição de ponteiro ou por *associação de argumento* (seção 9.2). No atual padrão, um *parâmetro de tipo deferido* (*deferred type parameter*) se refere ao valor do parâmetro de comprimento de uma variável de caractere ou *string*.[9]

Uma string de comprimento deferido é declarada através de uma das linhas de declaração abaixo

```
CHARACTER(LEN= :), POINTER[, <outros-atr>] :: <lista-nomes>
CHARACTER(LEN= :), ALLOCATABLE[, <outros-atr>] :: <lista-nomes>
```

onde se observa o caractere "`:`" no lugar de uma constante inteira, indicando que o comprimento dos objetos declarados na `<lista-nomes>` deve ser definido posteriormente, durante a execução do programa.

Este recurso possibilita a manipulação dinâmica de strings de comprimentos variáveis em uma miríade de situações distintas. Alguns exemplos são:

```
CHARACTER(LEN= :), POINTER :: VARSTR
CHARACTER(LEN= 100), TARGET :: NOME
CHARACTER(LEN= 200), TARGET :: ENDERECO
VARSTR => NOME
VARSTR => ENDERECO
```

[9] Recursos genéricos para a mudança do tipo um de objeto de dados são usualmente fornecidos por técnicas de programação orientada a objeto. O Fortran possui essa capacidade, mas isso não será abordado nesta Apostila.

Neste exemplo, o comprimento do ponteiro VARSTR é definido por atribuição de ponteiro: primeiro o seu comprimento é 100 (comprimento de NOME) e depois seu comprimento é 200 (ENDERECO).

O recurso da alocação/realocação automática, introduzido na seção 7.1.2.2 para matrizes, também se aplica neste caso. O programa abaixo apresenta alguns casos simples de uso deste recurso. Nota-se que foi empregada a constante iostat_eor, a qual é parte do módulo intrínseco ISO_FORTRAN_ENV (seção 9.3.6).

Listagem 7.7: Alguns exemplos de uso de strings com comprimento deferido para objetos escalares.

```
program tes_deferred_type_scalar
use iso_fortran_env, only: iostat_eor
implicit none
character(len= 1) :: buffer
character(len= :), allocatable :: str1, str2, input
integer :: ion

str1= 'Bom dia!'  ! Alocação automática
print'(3a,g0)', 'String (1): ->|', str1, '|<- Comp. string: ', len(str1)
str2= 'Tudo bem?' ! Alocação automática
print'(3a,g0)', 'String (2): ->|', str2, '|<- Comp. string: ', len(str2)
str1= str1//'  '//str2 ! Realocação automática
print'(3a,g0)', 'String (3): ->|', str1, '|<- Comp. string: ', len(str1)

!Entre com uma frase qualquer no teclado.
input= ''
print'(/,a)', 'Escreva uma frase abaixo:'
do
   read(unit= *, fmt= '(a)', advance= 'no', iostat= ion)buffer
   select case(ion)
   case(0)
      input= input//buffer
   case(iostat_eor)
      exit
   end select
end do
print'(3a,g0)', 'String lida: ->|', input, '|<- Comp. string: ', len(input)
end program tes_deferred_type_scalar
```

Os recursos apresentados no programa 7.7 são suficientes para manipular strings variáveis escalares. Para se trabalhar com *matrizes alocáveis de strings alocáveis* torna-se necessário o uso da forma mais geral do comando ALLOCATE, o que será discutido na próxima seção.

7.3.2 A FORMA GERAL DA INSTRUÇÃO ALLOCATE

A forma mais geral da instrução ALLOCATE é:

```
ALLOCATE([<type-spec> ::] <lista-alloc> [, <spec-alloc>] ...)
```

Nesta instrução, <lista-alloc> é uma lista de alocações, sendo que cada elemento da lista tem a forma

```
<obj-allocate>[(<lista-ext-array>)]
```

e cada <ext-array> na <lista-ext-array> é uma declaração de extensões na forma

```
[<lim-inferior>:]<lim-superior>
```

Quando presente, <spec-alloc> é pelo menos uma das especificações de alocação seguintes:

```
ERRMSG= <err-mess>
STAT= <status>
SOURCE= <type-expr>
MOLD= <type-expr>
```

Os campos ERRMSG= e STAT= são os mesmos já discutidos na forma mais simples do comando, apresentada na seção 7.1.1.

Os especificadores <type-spec>, SOURCE= e MOLD= contêm informações a respeito dos tipos dos objetos de dados sendo alocados. Quando empregados, o uso destes especificadores se dá de forma excludente: ou é empregado somente o especificador <type-spec>, quando então ocorre uma *alocação de tipo* (*typed allocation*), ou somente é empregado um dos especificadores SOURCE= ou MOLD=, quando então ocorre a *alocação de fonte* (*sourced allocation*). Estes distintos tipos de alocação são discutidos a seguir, no contexto dos objetos alocáveis em estudo nesta seção. Contudo, este recurso também é empregado em técnicas de programação orientada a objeto.

7.3.2.1 Alocação de tipo

Na alocação de tipo, as informações sobre o tipo e parâmetros de espécie dos objetos a ser alocados são mencionados explicitamente no campo <type-spec> ::. Este recurso pode ser empregado para alocar espaço de memória para strings escalares de comprimento deferido, como em

```
INTEGER :: COMP
CHARACTER(LEN= :), ALLOCATABLE :: STRV
READ*, COMP
ALLOCATE(CHARACTER(LEN= COMP) :: STRV)
READ'(A)', STRV
```

Neste exemplo, a string STRV é declarada ser de comprimento deferido. Então, a variável inteira COMP é lida, a qual determinará o comprimento de STRV, determinado via alocação de tipo pelo comando ALLOCATE.

A alocação de tipo também pode ser empregada para declarar a forma de uma *matriz alocável* composta por *strings de comprimento deferido*, como no exemplo:

```
INTEGER :: M, N
CHARACTER(LEN= :), DIMENSION(:), ALLOCATABLE :: VSTRV
⋮ ! M e N são determinados
ALLOCATE(CHARACTER(LEN= M) :: VSTRV(N))
```

Neste exemplo, o programa executável irá alocar espaço em memória suficiente para armazenar um vetor VSTRV composto por N elementos, sendo que cada elemento é uma string de comprimento M. Observa-se que neste caso, após a alocação de memória, VSTRV passa a ser um vetor de objetos homogêneos; ou seja, todos os seus elementos têm o mesmo comprimento. Matrizes compostas por strings heterogêneas podem ser implementadas usando tipos derivados com componentes de tipo deferido. Isto será discutido na seção 7.3.3

7.3.2.2 Alocação de fonte

De forma alternativa à alocação de tipo, as informações sobre o tipo e parâmetros de espécie dos objetos a ser alocados podem ser determinadas a partir de objetos já existentes ou de expressões que determinam o tipo e espécie. No primeiro caso, com o qualificador SOURCE=, como em

```
ALLOCATE(OBJ1, SOURCE= OBJ2)
```

o tipo e espécie do objeto alocável OBJ1 (e, portanto, o tamanho de memória requerido) são tomados do objeto OBJ2, o qual por hipótese já existe. Além disso, o valor de OBJ2 é copiado em OBJ1; em essência, OBJ1 é um clone de OBJ2.

Já com o outro qualificador (MOLD=), como em

```
ALLOCATE(OBJ3, MOLD= OBJ4)
```

o tipo e espécie do objeto OBJ3 são determinados a partir de OBJ4, porém não ocorre cópia do valor de OBJ4.

O exemplo a seguir mostra alguns usos simples destes recursos.

```
program tes_typed_sourced_allocation
implicit none
integer :: m
character(len= :), allocatable :: str1               ! Escalar alocável
character(len= :), dimension(:), allocatable :: str2 ! Vetor alocável
character(len= *), parameter :: cst1= 'String 1', &
                                cst2= 'Constante de string 2 (mais longa)'

print'(a)', '===========> Alocações de escalar <==========='
allocate(character(len= 20) :: str1) ! Alocação de tipo: escalar
print*, 'Digite uma frase:'
read'(a)', str1
print'(3a,g0)', 'str1 (1): ->|', str1, '|<- Comp. string: ', len(str1)

! Realocação automática continua sendo possível
str1= 'Esta é uma frase que tem mais de 20 caracteres de comprimento.'
print'(/,3a,g0)', 'str1 (2): ->|', str1, '|<- Comp. string: ', len(str1)

! Alocação de fonte com source=
deallocate(str1) ! Primeiro libere o espaço anterior
allocate(str1, source= cst1)
print'(/,3a,g0)', 'str1 (3): ->|', str1, '|<- Comp. string: ', len(str1)

! Alocação de fonte com mold=
deallocate(str1) ! Primeiro libere o espaço anterior
allocate(str1, mold= cst2)
print'(/,3a,g0)', 'str1 (4): ->|', str1, '|<- Comp. string: ', len(str1)
print*, 'Digite uma frase:'
read'(a)', str1
print'(3a,g0)', 'str1 (1): ->|', str1, '|<- Comp. string: ', len(str1)

print'(2/,a)', '===========> Alocações de vetor <==========='
write(*,'(a)',advance='no')'Nº elementos do vetor (>= 2): ' ; read*, m
allocate(character(len= len(cst2)) :: str2(m)) ! Alocação de tipo
print'(a)', 'Alocação de tipo:'
str2(1)= cst1 ; print'(3a)', 'str2(1)= ->|', str2(1), '|<-'
str2(2)= cst2 ; print'(3a)', 'str2(2)= ->|', str2(2), '|<-'
! Entre com outros elementos...

! Alocação de fonte com mold=
deallocate(str2) ! Primeiro libere o espaço anterior
allocate(str2(m), mold= cst1)
print'(/,a)', 'Alocação de fonte:'
str2(1)= cst1 ; print'(3a)', 'str2(1)= ->|', str2(1), '|<-'
str2(2)= cst2 ; print'(3a)', 'str2(2)= ->|', str2(2), '|<-'
end program tes_typed_sourced_allocation
```

7.3.3 TIPOS DERIVADOS COM COMPONENTES ALOCÁVEIS

Já foi mencionado na seção 7.1.3, no contexto de matrizes alocáveis, que componentes de tipos derivados também podem ser alocáveis. No presente contexto este conceito se estende para qualquer tipo de componente que, individualmente, é um objeto de dados alocável. Dessa forma é possível implementar matrizes cujos elementos são strings heterogêneas, isto é, strings que possuem comprimentos distintos. Uma implementação deste recurso é mostrada no programa abaixo.

```
program tes_derived_type_string
use iso_fortran_env, only: iostat_eor
```

```
implicit none
integer :: n, ion
character(len= 1) :: buffer
character(len= :), allocatable :: str
type :: td
   character(len= :), allocatable :: cs
end type td
type(td), dimension(:), allocatable :: vtd

allocate(vtd(0)) ! Cria vetor tamanho zero.
print'(a)', 'Entre com diversas frases.  Escreva "Fim" para encerrar.'
loop1: do
   write(*, '(a)', advance= 'no')'Frase: '
   str= ''
   loop2: do
      read(*, '(a)', advance= 'no', iostat= ion)buffer
      select case(ion)
      case(0)
         str= str//buffer
      case(iostat_eor)
         exit loop2
      end select
   end do loop2
   if(str == "Fim")exit loop1
   vtd= [ vtd, td(str) ]
end do loop1

print'(2/,a)', 'Frases lidas:'
print'(3a)', ('->|', vtd(n)%cs, '|<-', n= 1, size(vtd))
end program tes_derived_type_string
```

Um componente alocável pode ser de qualquer tipo intrínseco ou derivado. É permitido também que um componente alocável seja do mesmo tipo derivado sendo definido ou de um tipo definido posteriormente na mesma unidade de programa. Este recurso é exemplificado no tipo `MY_REAL_LIST` abaixo:

```
TYPE :: MY_REAL_LIST
   REAL :: VALOR
   TYPE(MY_REAL_LIST), ALLOCATABLE :: NEXT
END TYPE MY_REAL_LIST
```

Uma definição como esta serve para contruir *listas alocáveis*, as quais são discutidas na seção 7.4.1.

7.4 Estruturas dinâmicas de dados

Com os recursos disponíveis para a criação e manipulação de objetos dinâmicos de dados, estruturas extremamente complexas podem ser implementadas, as quais atendem as mais diversas exigências. Nesta seção algumas das estruturas dinâmicas de dados mais comuns serão abordadas, começando por *listas encadeadas.*

7.4.1 Listas encadeadas ou alocáveis

Nesta seção serão discutidas duas implementações de um dos tipos mais simples de estrutura dinâmica de dados: uma estrutura linear, na qual é estabelecida uma coleção linear de uma quantidade indeterminada de objetos de dados. As implementações apresentadas são *listas encadeadas* e *listas alocáveis.*

7.4.1.1 LISTAS ENCADEADAS

Uma *lista encadeada* (*linked list*) é uma coleção linear de objetos de dados de tamanho indeterminado. Cada elemento da lista possui componentes que armazenam valores de dados e pelo menos uma *referência* ou *conexão* (*link*) ao próximo elemento da lista. Em linguagens modernas de programação, listas encadeadas são implementadas por meio de estruturas que contêm componentes de dados e ponteiros que estabelecem a referência ao próximo elemento.

A figura 7.4 ilustra um exemplo básico de lista encadeada. Cada elemento da lista (um *nodo* da lista) é composto por um dado (`valor`) e por um ponteiro (`next`) que pode aponta para o próximo elemento da lista, exceto pelo último, o qual aponta para `NULL()`. Esta ilustração faz lembrar o elos de uma corrente; por esta razão esta estrutura é denominada uma lista encadeada. O início da lista é identificado pelo ponteiro `head`, enquanto que o último elo na lista é localizado por `tail`.

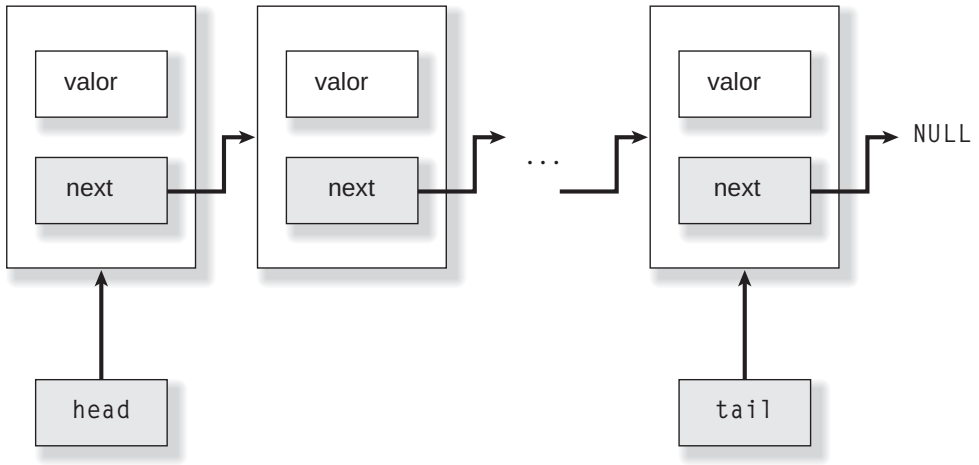

Figura 7.4: Uma lista encadeada típica. Cada ponteiro (*`next`*) aponta para o próximo elemento da lista.

Listas encadeadas oferecem uma outra solução ao problema apresentado no início deste capítulo: como armazenar um conjunto previamente indeterminado de objetos de dados. A solução apresentada por uma lista encadeada pode ser equivalente a uma matriz alocável em qualquer circunstância, mas listas encadeadas oferecem mais opções de armazenamento do que matrizes.

A criação de uma lista encadeada simples será exemplificada agora. Primeiro, define-se um tipo derivado que contém um valor real e um componente de ponteiro do mesmo tipo, o qual irá estabelecer a conexão com o nodo seguinte:

```
TYPE :: NODO
   REAL :: VALOR
   TYPE(NODO), POINTER :: NEXT => NULL()
END TYPE NODO
```

Este tipo pode, obviamente, conter outros componentes; por exemplo, pode-se declarar um componente inteiro para servir de contador (ou índice) do número de nodos presentes na lista.

Uma vez definido o tipo, declaram-se ponteiros para compor os nodos da lista. São necessários pelo menos dois ponteiros, denominados `HEAD` e `CURRENT`, os irão armazenar os descritores do início da lista e do nodo corrente, respectivamente. Se for conveniente, pode-se declarar um outro ponteiro, `TAIL`, o qual irá armazenar o endereço do último nodo. Os ponteiros `HEAD` e `TAIL` fornecem acesso imediato aos pontos extremos da lista. Os ponteiros são declarados então:

```
TYPE(NODO), POINTER :: HEAD => NULL(), CURRENT => NULL(), TAIL => NULL()
```

A figura 7.5 ilustra os passos na construção da lista. No painel (a) mostra-se o status dos ponteiros no momento das suas declarações.

Até o momento, todos os ponteiros estão desassociados e nenhum espaço em memória foi alocado. Assim, se não houver valores para serem lidos, basta seguir adiante com o programa

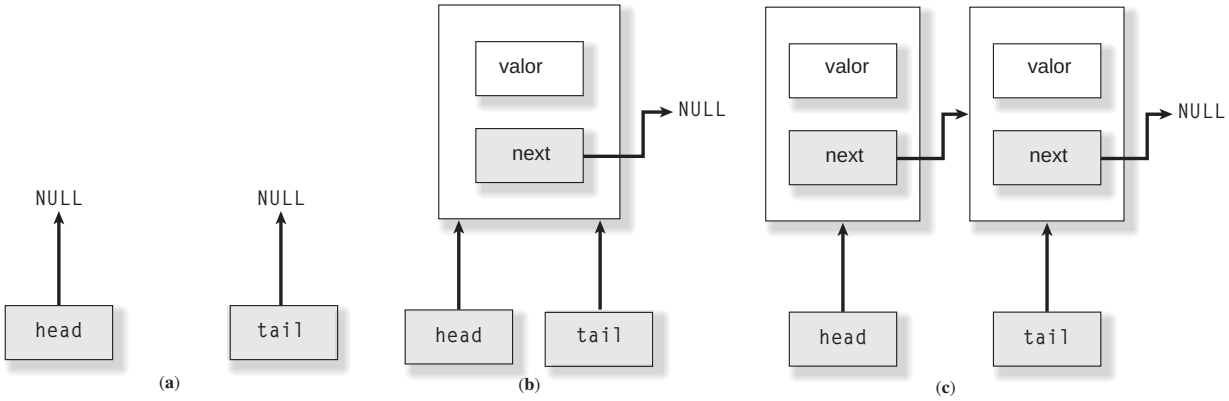

Figura 7.5: Construindo uma lista encadeada: (a) Ponteiros inicialmente desassociados. (b) Situação após a inclusão do primeiro nodo na lista. (c) Situação após a inclusão do segundo nodo na lista.

sem que haja disperdício de recursos. Supondo que exista pelo menos um valor real lido, aloca-se então um espaço anônimo na memória para o componente real, criando também um descritor para o componente ponteiro. Os ponteiros `HEAD` e `TAIL` adquirem então o descritor desse espaço de memória:

```
REAL :: VAR ! Armazena os valores lidos.
⋮ ! O valor real é lido
ALLOCATE(HEAD)
HEAD%VALOR= VAR
TAIL => HEAD
```

Desta maneira, o primeiro nodo da lista encadeada é criado. O painel (b) da figura 7.5 ilustra o status dos ponteiros. Nota-se que os ponteiros `HEAD` e `TAIL` ambos contém o endereço do primeiro bit do espaço de memória alocado. Observa-se também que o seus componentes `NEXT` continuam desassociados, mas os ambos os ponteiros irão conter o descritor do mesmo se `NEXT` for alocado.

Suponha agora que exista pelo menos mais um valor real a ser armazenado. Então, após o mesmo ser lido, um novo nodo da lista deve ser criado. Isto é realizado através dos seguintes passos: *(i)* aloca-se memória para o componente `TAIL%NEXT`. *(ii)* Realiza-se a atribuição `TAIL => TAIL%NEXT`; isto significa que o ponteiro `TAIL` agora contém o descritor (o endereço) do novo espaço de memória, mas como o componente `TAIL%NEXT` era compartilhado com o ponteiro `HEAD`, no momento da alocação o seu componente `HEAD%TAIL` continuou apontando para o início do novo espaço de memória. *(iii)* Finalmente, atribui-se a `TAIL%VALOR` o valor lido. O painel (c) da figura 7.5 ilustra a situação após a inclusão do segundo nodo na lista. A sequencia de comandos é a seguinte:

```
⋮ ! Novo valor de VAR lido
ALLOCATE(TAIL%NEXT)
TAIL => TAIL%NEXT
TAIL%VALOR= VAR
```

Caso existam novos valores a serem incluídos, os passos *(i)* – *(iii)* acima são repetidos até que as entradas se encerrem. A cada novo nodo incluído, o ponteiro `TAIL` estará sempre apontando para o último nodo, enquanto que o ponteiro `HEAD` permanece estoicamente apontando para o primeiro nodo.

Após a lista toda ser lida, pode-se voltar para o início da mesma para armazenar o seus valores em um vetor ou para outros fins quaisquer. Isto é realizado com a atribuição inicial `CURRENT => HEAD`, a qual irá resultar com `CURRENT` contendo o endereço do início da lista. Para acessar os outros nodos, deve-se lembrar que `HEAD%NEXT` contém o endereço do segundo nodo e, portanto, `CURRENT%NEXT` também. Assim, basta realizar a atribuição `CURRENT => CURRENT%NEXT` que o segundo nodo estará acessível. Repetindo esta última atribuição, percorre-se a lista no sentido `HEAD` $\longrightarrow$ `TAIL`. Para verificar se o final da lista foi atingido, basta lembrar que o componente `TAIL%NEXT` está desassociado. Portanto, quando `CURRENT` estiver com os descritores de `TAIL`, a

atribuição CURRENT => CURRENT%NEXT irá levar CURRENT ao status de desassociado. Assim, basta testar o seu status para determinar quando o final da lista foi atingido. A sequencia de comandos fica assim:

```
CURRENT => HEAD
DO
   IF(.NOT. ASSOCIATED(CURRENT))EXIT
   ⋮ ! Faça algo com CURRENT%VALOR
   CURRENT => CURRENT%NEXT
END DO
```

O programa abaixo é uma implementação deste algoritmo. Deseja-se ler da entrada padrão uma quantidade indeterminada de valores reais não nulos. Assim, para verificar o final das entradas, o programa simplesmente testa se o valor é zero. Uma vez que todos os números não nulos foram lidos, o programa imprime os valores acumulados na lista na saída padrão. É importante ressaltar que o ponteiro TAIL pode ser substituído por CURRENT, uma vez que sua presença neste programa não é estritamente necessária.

Listagem 7.8: Exemplo de implementação de uma lista singularmente encadeada.

```
1  program linked_list_01
2  implicit none
3  real :: var
4  type :: nodo
5     real :: valor
6     type(nodo), pointer :: next => null()
7  end type nodo
8  type(nodo), pointer :: head => null(), tail => null(), current => null()
9  ! Procede a atribuição de valores na lista
10 do
11    write(*,'(a)',advance='no')'valor (0.0: saída)= ' ; read*, var
12    if(var == 0.0)exit
13    if(.not. associated(head))then ! Inicia a lista
14       allocate(head)              ! Aloca espaço na memória para head
15       head%valor= var             ! Acumula valor lido em head
16       tail => head                ! Tail aponta para head
17    else                     ! Inicia um novo nodo
18       allocate(tail%next) ! Aloca espaço para o próximo nodo
19       tail => tail%next   ! Tail aponta para o novo nodo
20       tail%valor= var     ! Acumula valor no nodo
21    end if
22 end do
23 !Lista lida.  Agora retorna ao início e imprime valores na tela
24 print'(/,a)', 'Valores lidos:'
25 if(associated(head))then ! Primeiro confirma que lista existe
26    current => head ! Retorna ao início
27    do
28       if(.not. associated(current))then
29          tail => null()
30          exit ! Final da lista
31       end if
32       print'(a,g0)', 'Valor= ', current%valor
33       head => current         ! Move início para o nodo corrente
34       current => current%next ! Aponta para o próximo nodo
35       deallocate(head)        ! Libera espaço do nodo anterior
36    end do
37 end if
38 ! Tudo pronto.  Realize a limpeza final e verifique
39 print'(/,a,g0)', 'Ponteiro head está associado?    ', associated(head)
40 print'(a,g0)', 'Ponteiro tail está associado?    ', associated(tail)
41 print'(a,g0)', 'Ponteiro current está associado? ', associated(current)
```

```
42 end program linked_list_01
```

Uma leitura atenta do programa mostra que há instruções adicionais às que constam no algoritmo, especificamente nas linhas 13, 25, 29, 33 e 35. Se o objetivo do programa for simplesmente ler a lista uma única vez e imprimir os valores na saída padrão, então essas instruções não são necessárias. Contudo, em um programa mais longo, no qual a lista pode ser reconstruída diversas vezes, tanto no programa principal quanto em outras unidades de programa, as instruções citadas são importantes, principalmente nas linhas 29, 33 e 35. Na linha 33, o ponteiro `HEAD` adquire os descritores de `CURRENT` antes que este último aponte para o próximo nodo. Em seguida, `CURRENT` se move para o nodo seguinte e, na linha 35, o comando `DEALLOCATE(HEAD)` libera o espaço em memória ocupado pelo nodo anterior e ao mesmo tempo desassocia `HEAD`. Estas instruções são importantes porque se posteriormente fosse necessário iniciar uma nova lista encadeada, uma nova execução do `ALLOCATE(HEAD)` (linha 14) iria inicar a nova lista em um outro espaço de memória e todos os novos nodos também ocupariam outros espaços. Contudo, os espaços de memória reservados para a lista anterior permaneceriam alocados caso não fossem executadas as instruções nas linhas 33 e 35, e esse espaço permaneceria inacessível e ocioso durante todo o restante da execução do programa. Em uma situação onde a lista é recriada diversas vezes, a existência desses espaços ociosos e reservados de memória pode resultar em uma condição de esgotamento da memória disponível no computador. Este acontecimento é denominado *vazamento ou esgotamento de memória* (*memory leak*). As instruções nas linhas 33 e 35 evitam a ocorrência do vazamento de memória.

Por outro lado, caso a instrução na linha 29 não fosse aplicada, o ponteiro `TAIL` permaneceria com o status de definido e com os descritores de um espaço de memória que foi dealocado, isto é, que foi disponibilizado de volta ao sistema operacional. Nesta situação, o descritores em `TAIL` perdem o sentido de ser, uma vez que o programa pode usar esse espaço de memória para armazenar outros tipos de dados. Um uso posterior de `TAIL` poderia primeiro constatar que o mesmo se encontra associado e assim supor que os descritores contidos no mesmo permanecem válidos. Contudo, se entrementes o programa gravou outro dado nesse espaço, ou em parte desse espaço, os valores acessíveis por `TAIL` serão completamente errôneos. Quando um ponteiro retém o endereço de um espaço de memória que foi dealocado em um outro ponto do programa, este se torna um *ponteiro pendente* (*dangling pointer*). A instrução na linha 29 evita essa ocorrência. As instruções nas linhas 37 – 39 confirmam que todos os ponteiros estão desassociados no final do programa.

Sugestões de uso & estilo para programação

Quando ponteiros são empregados em um programa, é importante sempre tomar providências que evitem a ocorrência de vazamentos de memória (*memory leaks*) ou de ponteiros pendentes (*dangling pointers*).

7.4.1.2 Listas alocáveis

No Fortran existe uma implementação alternativa às listas encadeadas para a formação de uma estrutura linear de dados. Tratam-se das *listas alocáveis* (*allocatable lists*). Na seção 7.3.3 observou-se que componentes alocáveis de tipos derivados podem ser de qualquer tipo, inclusive do próprio tipo sendo definido ou de um tipo que ainda não foi definido.

Com este recurso, pode-se criar uma lista alocável alternativa à lista encadeada mostrada no programa 7.8. O programa abaixo exemplifica o uso deste recurso.

Listagem 7.9: Exemplo de implementação de uma lista alocável.

```
1 program allocatable_list_02
2 implicit none
3 real :: var
4 type :: alloc_list
5    real :: valor
6    type(alloc_list), allocatable :: next
7 end type alloc_list
8 type(alloc_list), allocatable, target :: lista
9 type(alloc_list), pointer :: nodo => null()
```

```
10 ! Procede a atribuição de valores na lista
11 do
12    write(*,'(a)',advance='no')'valor (0.0: saída)= ' ; read*, var
13    if(var == 0.0)exit
14    if(.not. associated(nodo))then ! Inicia a lista
15       allocate(lista, source= alloc_list(valor= var)) ! Inicia lista
16       nodo => lista                ! Nodo aponta para início da lista
17    else                      ! Inicia um novo nodo
18       allocate(nodo%next, source= alloc_list(valor= var)) ! Cria novo nodo
19       nodo => nodo%next    ! Aponta para o novo nodo
20    end if
21 end do ! Nodo continua associado ao final da lista.
22 !Lista lida.  Agora retorna ao início e imprime valores na tela
23 print'(/,a)', 'Valores lidos:'
24 if(allocated(lista))then ! Primeiro confirma que lista existe
25    nodo => lista ! Retorna ao início da lista
26    do
27       if(.not. associated(nodo))exit ! Final da lista
28       print'(a,g0)', 'Valor= ', nodo%valor
29       nodo => nodo%next ! Aponta para o próximo nodo
30    end do
31 end if ! Nodo resulta desassociado ao final da lista
32 ! Tudo pronto.  Realize a limpeza final e verifique
33 deallocate(lista) ! Dealoca todos os nodos da lista
34 print'(/,a,g0)', 'A lista está alocada? ', allocated(lista)
35 print'(a,g0)', 'O ponteiro nodo está associado? ', associated(nodo)
36 end program allocatable_list_02
```

A implementação realizada no programa 7.9 possui algumas diferenças importantes em relação ao programa 7.8. Como os nodos são agora objetos alocáveis ao invés de objetos anônimos, pode-se empregar alocação de fonte (seção 7.3.2.2) juntamente com um construtor de estrutura. Isto é realizado nas linhas 15 e 18, onde na mesma instrução aloca-se o espaço e realiza-se a atribuição de valor do nodo.

Mas a diferença mais importante se revela na parte final do programa. Por ser um objeto alocável, a instrução `deallocate(lista)` na linha 33 irá automaticamente liberar o espaço de memória reservado para *toda a lista*. Neste caso não é necessário dealocar nodo por nodo como foi preciso com o uso de ponteiros. Uma outra vantagem tem impacto na optimização do código. Por exigência do padrão, objetos alocáveis são alocados de forma *contígua*. Por outro lado, tal exigência não é imposta a listas encadeadas formadas por ponteiros; o sistema operacional pode escolher onde reservar espaço para um determinado nodo, sem que este seja necessariamente contíguo em relação aos nodos vizinhos.

7.4.1.3 LISTAS ENCADEADAS CIRCULARES

Uma das limitações de listas encadeadas lineares como as criadas nos programas 7.9 e 7.8 está no fato de que em cada vez que for necessário procurar um determinado valor armazenado na mesma, deve-se iniciar a busca sempre a partir do início da lista, o que terá um impacto no tempo de processamento da busca. Existem diversas estratégias que reduzem o tempo necessário para a realização da busca no interior de uma lista. Uma dessas estratégias consiste no estabelecimento de uma *lista encadeada circular*.

Listas lineares, como aquelas descritas nas seções 7.4.1.1 e 7.4.1.2, são listas *abertas*, isto é, no final da lista a referência ao próximo modo é sempre indefinida (um ponteiro para `NULL` ou objeto não alocado). Por outro lado, uma lista circular é estabelecida quando, no término da incorporação de dados à lista, executa-se a atribuição de ponteiro

```
TAIL%NEXT => HEAD
```

Assim, durante a varredura da lista, ao se chegar ao seu final (`TAIL`) o próximo nodo será imediatamente o início (`HEAD`), conforme está representado na figura 7.6. Uma outra estratégia consiste no estabelecimento de uma *lista duplamente encadeada*, discutida na seção 7.4.2.

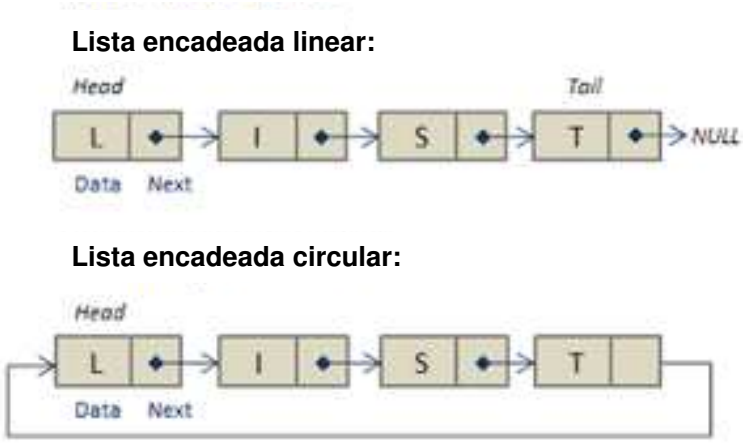

Figura 7.6: Diferença entre uma lista linear e uma lista circular.

7.4.2 Listas duplamente encadeadas e árvores binárias

Serão rapidamente discutidos agora dois tipos comuns de estruturas dinâmicas de dados contendo dois ponteiros em cada nodo: *listas duplamente encadeadas* e *árvores binárias*.

Listas duplamente encadeadas

As listas encadeadas apresentadas na seção 7.4.1 são exemplos de listas *simplesmente* encadeadas. Uma vez que essas listas possuem somente um ponteiro em cada nodo, elas somente podem ser percorridas no sentido `HEAD` $\longrightarrow$ `TAIL`. Um processo de busca no interior da lista poderia ser executado mais rapidamente se a mesma pudesse ser percorrida em ambos os sentidos. Uma *lista duplamente encadeada* permite justamente isso. A definição básica de um tipo derivado que contém um nodo de uma lista duplamente encadeada é :

```
TYPE :: DLNODO
   REAL :: VALOR
   TYPE(DLNODO), POINTER :: PREVIOUS => NULL()
   TYPE(DLNODO), POINTER :: NEXT => NULL()
END TYPE DLNODO
TYPE(DLNODO), POINTER :: HEAD => NULL(), CURRENT => NULL(), TAIL => NULL()
```

O início da lista será novamente determinado por

```
REAL :: VAR ! Armazena os valores lidos.
⋮ ! O valor real é lido
ALLOCATE(HEAD)
HEAD%VALOR= VAR
TAIL => HEAD
```

mas os próximos nodos serão estabelecidos por:

```
⋮ ! Novo valor de VAR lido
CURRENT => TAIL
ALLOCATE(TAIL%NEXT)
TAIL => TAIL%NEXT
TAIL%VALOR= VAR
TAIL%PREVIOUS => CURRENT
```

Desta forma, a cada novo nodo incluído na lista, o componente `PREVIOUS` irá sempre apontar para o nodo anterior, enquanto que o componente `NEXT` permanecerá desassociado. A estrutura final da lista é representada na figura 7.7.

Esta lista se torna uma *lista duplamente encadeada circular* se o componente `HEAD%PREVIOUS` apontar para `TAIL`, ao mesmo tempo em que o componente `TAIL%PREVIOUS` apontar para `HEAD`.

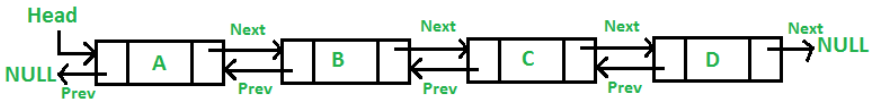

Figura 7.7: Uma lista duplamente encadeada linear.

ÁRVORES BINÁRIAS

Uma *árvore binária* é uma outra estrutura dinâmica cujos nodos possuem dois ponteiros. Contudo, ao invés de estabelecer uma estrutura linear com os mesmos, pode-se estabelecer uma estrutrura que se assemelha às raízes de uma árvore partindo do tronco ou aos galhos de uma árvore invertida. A figura 7.8 ilustra uma árvore binária com diversos nodos estabelecidos.

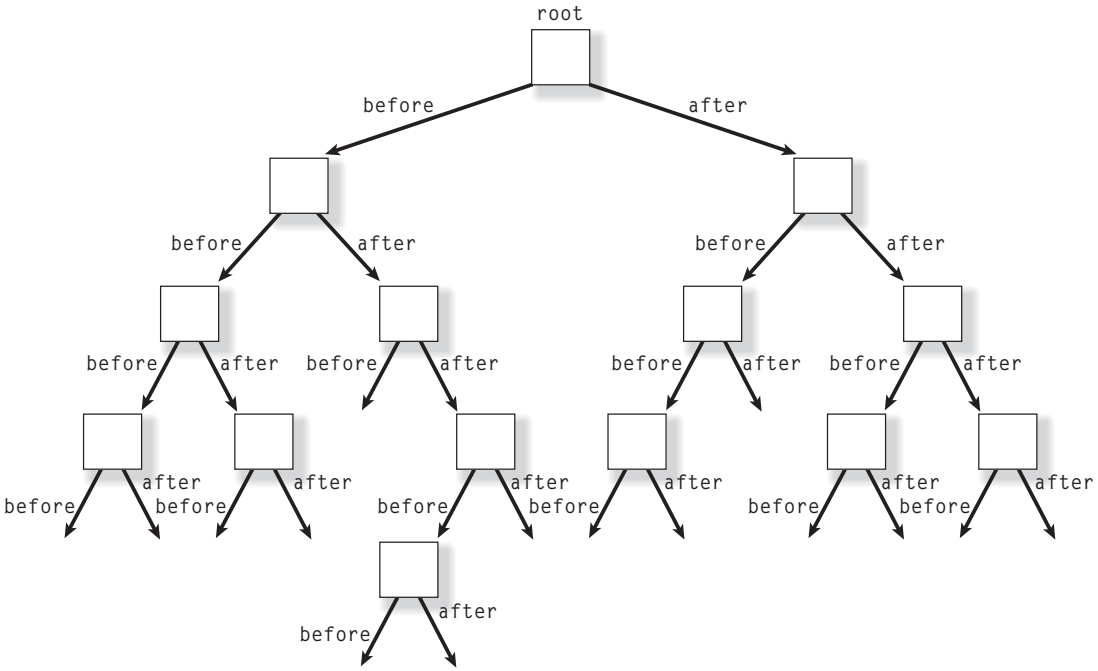

Figura 7.8: Uma árvore binária cujos nodos finais não estão associados.

7.4.3 MATRIZES DE PONTEIROS E LISTAS MULTIPLAMENTE CONECTADAS

Ao longo de toda a seção 7.2, sempre que uma referência foi feita envolvendo ponteiros e matrizes, a menção era referente a *ponteiros para matrizes* (*pointers to arrays* ou *array pointers*). Ou seja, o ponteiro sempre apontava para um alvo que no final das contas era um objeto de dados matricial.

Um tipo de estrutura distinto é uma *matriz de ponteiros* (*array of pointer* ou *pointer array*), isto é, matrizes cujos elementos são ponteiros. No Fortran, uma matriz de ponteiros pode ser criada a partir de um tipo derivado que contém componentes que são ponteiros para matrizes. Por exemplo,

```
TYPE :: ARR_PTR
   REAL, DIMENSION(:), POINTER :: P
END TYPE ARR_PTR
```

estabelece um tipo cujo componente é um ponteiro a um vetor. Declarando matrizes do tipo ARR_PTR, cada elemento dessas matrizes são ponteiros a vetores de tamanhos arbitrários. Esta é uma maneira para se criar *matrizes irregulares* (*jagged arrays*).

O programa abaixo implementa uma matriz irregular VPTR na qual o número de linhas, o número de colunas em cada linha e os valores dos elementos são todos determinados aleatoriamente.

```
program jagged_array_01
implicit none
integer, parameter :: mnelem= 2 ! Menor nº de elementos no vetor
integer :: nelem, i, j
real :: temp
integer, dimension(:), allocatable :: tams ! Contém os tamanhos dos vetores
real, dimension(:), pointer :: vtemp
type :: vec_ptr
   real, dimension(:), pointer :: p
end type vec_ptr
type(vec_ptr), dimension(:), allocatable :: vptr

do
   call random_number(temp)
   nelem= int(20.*temp)
   if(nelem >= mnelem)exit
end do
print'(a,g0)', 'Nº elementos do vetor de ponteiros: ', nelem
allocate(tams(nelem), vtemp(nelem))
call random_number(vtemp)
tams= int(20.*vtemp)
print'(/,a,*(g0,x))', 'Tamanhos dos elementos: ', tams

allocate(vptr(nelem))
do i= 1, nelem
   allocate(vtemp(tams(i)))
   call random_number(vtemp)
   vptr(i)%p => vtemp
end do

print'(2/,a)', 'Valores nos elementos de vptr:'
do i= 1, nelem
   print'(/,2(a,g0))', 'Elemento ', i, ':  Nº de elementos: ', size(vptr(i)%p)
   print'(5(g0,x))', (vptr(i)%p(j), j= 1, size(vptr(i)%p))
end do
end program jagged_array_01
```

Estruturas dinâmicas de extrema complexidade podem ser construídas desta maneira. Por exemplo, o par de tipos derivados definidos abaixo:

```
TYPE :: PTR
   TYPE(ENTRY), POINTER :: POINT
END TYPE PTR
TYPE :: ENTRY
   INTEGER :: INDEX
   REAL :: VALOR
   TYPE(PTR), DIMENSION(:), POINTER :: CHILDREN
END TYPE ENTRY
```

estabelece que cada elemento do vetor CHILDREN é um ponto (POINT) a partir do qual um novo vetor CHILDREN de tamanho arbitrário pode ser definido. Isto permite a criação de *listas multiplamente conectadas* ou *encadeadas* (*multiply linked lists*), nas quais cada nodo está conectado com um número arbitrário de outros nodos.

7.5 RECURSOS ORIENTADOS À COMPUTAÇÃO DE ALTA PERFORMANCE

[EM CONSTRUÇÃO]