

Programação Estruturada

Estruturas e Funções

Prof. Edson Alves

Faculdade UnB Gama

Sumário

1. Estruturas de controle
2. Tipos de dados compostos
3. Subrotinas e Funções
4. Módulos
5. Programação Procedural

Estruturas de seleção

- ▶ Em Fortran, a principal estrutura de seleção é o construto IF-THEN-ELSE, cuja sintaxe é

```
if (condicao) then
    blocoA
else
    blocoB
end if
```

- ▶ A condicao é uma variável ou expressão lógica
- ▶ Se a condicao for **verdadeira**, o blocoA é executado, e ao fim deste a execução segue para a código que segue o **end if**
- ▶ Caso contrário, o blocoB é executado
- ▶ A cláusula **else** é opcional
- ▶ Se um **if** segue imediatamente um **else**, é criada uma cascata de blocos mutuamente excludentes, sendo executado o primeiro cuja condição associada for verdadeira (ou o último, caso exista uma cláusula **else final**)

Exemplo de uso do construto IF-ELSE

```
1 program IRRF                ! Calcula o imposto de renda mensal
2
3     implicit none
4
5     real :: salario, aliquota, deducacao, imposto
6
7     write(*,*) 'Insira o salário mensal: '
8     read(*,*) salario
9
10    ! Determinar a aliquota e a dedução a partir do salário
11    if (salario <= 1903.98) then
12        aliquota = 0
13    else if (salario <= 2826.65) then
14        aliquota = 0.075
15        deducacao = 142.80
16    else if (salario <= 3751.05) then
17        aliquota = 0.15
18        deducacao = 354.80
```

Exemplo de uso do construto IF-ELSE

```
19  else if (salario <= 4664.68) then
20      aliquota = 0.225
21      deducacao = 636.13
22  else
23      aliquota = 0.275
24      deducacao = 869.36
25  end if
26
27  if (aliquota == 0) then
28      write(*,*) 'Isento'
29  else
30      imposto = salario * aliquota - deducacao
31      write(*,1) imposto
32  end if
33
34 1  format ('Imposto devido: ', F9.2)
35
36 end program IRRF
```

SELECT-CASE

- ▶ Outra estrutura de seleção disponível em Fortran é o construto SELECT-CASE, cuja sintaxe é

```
select case (seletor)
  case (lista de rótulos 1)
    bloco1
  case (lista de rótulos 2)
    bloco2
  ...
  case (lista de rótulos N)
    blocoN
  case default
    bloco_padrao
end select
```

- ▶ O seletor é uma variável ou expressão cujo tipo é **integer**, **character** ou **logical**
- ▶ As listas descrevem os rótulos que compõem cada caso, separados por vírgulas

SELECT-CASE

- ▶ Os rótulos podem ser especificados de quatro maneiras:

```
x  
a : b  
L :  
: R
```

- ▶ Na primeira forma, um único valor `x` é especificado
- ▶ Na segunda forma, são especificados todos os valores no intervalo `[a, b]` (aqui, `a` deve ser necessariamente menor do que `b`)
- ▶ Na terceira forma, são especificados todos os valores maiores ou iguais a `L`; na quarta, todos os valores menores ou iguais a `R`
- ▶ Uma vez determinado o valor do seletor, será executado o primeiro bloco cujo valor está relacionado na lista de rótulos, e em seguida a execução segue para a linha que sucede o **end select**
- ▶ O **case default** é opcional, e seu bloco será executado apenas se o valor do seletor não estiver listado em nenhum **case**

Exemplo de uso do construto SELECT-CASE

```
1 ! Determina a prioridade de atendimento do paciente, de acordo com a idade
2 program priority
3
4     implicit none
5
6     integer :: idade
7     character (len = 6) :: prioridade
8
9     write(*,*) 'Insira a idade do paciente: '
10    read(*,*) idade
11
12    select case (idade)
13        case (0 : 6)
14            prioridade = 'media'
15        case (65 : )
16            prioridade = 'maxima'
17        case (7 : 64)
18            prioridade = 'minima'
```


Exemplo de uso do construto SELECT-CASE

```
19      case default
20          write(*,*) 'Idade inválida!'
21      end select
22
23      ! Imprime a prioridade do paciente
24      write(*,1) idade, prioridade
25
26 1    format (I3, ' anos, prioridade: ', A6)
27
28 end program priority
```

Estruturas de laço

- ▶ Fortran disponibiliza duas estruturas de repetição
- ▶ A primeira delas é a estrutura DO, cuja sintaxe é

```
do variavel = a, b [, delta]  
    bloco  
end do
```

- ▶ A variavel de controle deve ser do tipo **integer**
- ▶ A variavel terá a como valor inicial e b como valor final
- ▶ Após cada execução do bloco, o valor da variável é acrescido do delta
- ▶ Se o delta (passo) for omitido, ele assume o valor 1 (um)
- ▶ O comando **exit**, se executado, encerra o laço imediatamente
- ▶ Já o comando **cycle** finaliza a execução do bloco, seguindo imediatamente para a atualização da variavel

Exemplo de uso do construto DO

```
1 ! Computa o fatorial de n
2 program factorial
3
4     implicit none
5
6     integer :: n, i, Fn = 1
7
8     read(*,*) n
9
10    do i = 1, n
11        Fn = Fn * i
12    end do
13
14    write(*,1) n, Fn
15
16 1   format ('Fatorial de ', I2, I10)
17
18 end program factorial
```

DO-WHILE

- ▶ Fortran possui uma segunda estrutura de repetição: o construto DO-WHILE, cuja sintaxe é

```
do while (condicao)
    bloco
end do
```

- ▶ A condicao é uma variável ou uma expressão do tipo **logical**
- ▶ Se a condicao for verdadeira, o bloco associado será executado
- ▶ Após a execução do bloco, a condição é reavaliada e, se permanecer verdadeira, o bloco é executado novamente
- ▶ Se o bloco não modifica as variáveis que compõem a condição de modo que ela possa eventualmente se tornar falsa, o laço será infinito
- ▶ Os comandos **exit** e **cycle** também podem ser usados neste construto, com o mesmo significado do construto DO

Exemplo de uso do construto DO-WHILE

```
1 program fast_exp                                ! Computa  $a^n$  em  $O(\log n)$ 
2   implicit none
3   integer(16) :: a, n, res = 1, base          ! Inteiros de 128-bits
4   read(*,*) a, n
5
6   base = a
7
8   do while (n > 0)
9       if (iand(n, 1) > 0) then                !  $iand(x, y) = x \& y$ 
10          res = res * base
11       end if
12
13       base = base * base
14       n = ishft(n, -1)                        !  $n = n \gg 1$ 
15   end do
16
17   write(*,*) res
18 end program fast_exp
```

Vetores

- ▶ Fortran tem suporte nativo para **vetores** (*arrays*) de elementos de um mesmo tipo
- ▶ A sintaxe para a declaração de um vetor é

```
tipo_de_dado :: nome(dim1, dim2, ..., dimN)
```

- ▶ O parêntesis que segue o nome da variável e as dimensões listadas determinam a **forma** (*shape*) do vetor
- ▶ A notação de parêntesis também pode ser utilizada para acessar os elementos individuais do vetor
- ▶ Fortran utilizar a indexação matemática, de modo que o primeiro elemento do vetor tem índice 1
- ▶ A palavra-chave **allocatable** pode ser utilizada para declarar vetores dinâmicos
- ▶ A função **allocate()** reserva espaço em memória para tais vetores, e esta memória deve ser liberada após o uso por meio da função **deallocate()**

Exemplo de uso de vetores

```
1 program statistics  ! Computa a média e o desvio padrão dos elementos do vetor xs
2   implicit none
3   integer, allocatable :: xs(:)    ! Vetor dinâmico
4   integer :: n, i                  ! n = dimensão de xs
5   real :: stats(2)                 ! Vetor com duas posições
6
7   write(*,*) 'Insira o número de entradas: '
8   read(*,*) n
9
10  if (n < 1) then
11    return
12  end if
13
14  allocate(xs(n))
15
16  do i = 1, n
17    write(*,*) 'Insira a entrada ', i, ': '
18    read(*,*) xs(i)
19  end do
```

Exemplo de uso de vetores

```
21  stats(1) = 0.0                                ! Média
22
23  do i = 1, n
24      stats(1) = stats(1) + xs(i)
25  end do
26
27  stats(1) = stats(1) / n
28  stats(2) = 0.0                                ! Desvio-padrão
29
30  do i = 1, n
31      stats(2) = stats(2) + (xs(i) - stats(1)) ** 2
32  end do
33
34  stats(2) = sqrt(stats(2)/n)
35  write(*,*) 'Média = ', stats(1)
36  write(*,*) 'Desvio = ', stats(2)
37  deallocate(xs)
38
39 end program statistics
```


Manipulação de vetores

- ▶ Fortran disponibiliza uma série de características úteis para a manipulação de vetores
- ▶ Por exemplo, se a , b e c são vetores de mesma dimensão (N), a expressão “ $c = a + b$ ” equivale a

```
do i = 1, N  
    c(i) = a(i) + b(i)  
end do
```

- ▶ A atribuição “ $xs = k$ ” atribui o valor k a todos os elementos do vetor xs
- ▶ A atribuição também pode ser utilizada para copiar vetores de mesma dimensão
- ▶ Além disso, há várias funções intrínsecas que manipulam vetores diretamente, como `dot_product`, `matmul`, `maxval`, `minval`, `product` e `sum`

Exemplo de manipulação de vetores

```
1 program angle          ! Calcula o ângulo entre dois vetores
2   real, parameter :: pi = acos(-1.0)
3   real :: theta, xlen, ylen
4   real :: xs(2) = (/ 1, 0 /), ys(2) = 1    ! ys = (1, 1)
5   real :: A(2, 2) = reshape((/ 0, 1, -1, 0 /), (/ 2, 2 /))
6
7   xlen = sqrt(dot_product(xs, xs))
8   ylen = sqrt(dot_product(ys, ys))
9   theta = acos(dot_product(xs, ys) / (xlen * ylen))
10  theta = theta * 180 / pi
11
12  write(*,*) 'Ângulo, em graus: ', theta
13
14  ys = matmul(A, ys)
15  theta = acos(dot_product(xs, ys) / (xlen * ylen))
16  theta = theta * 180 / pi
17
18  write(*,*) 'Ângulo após rotação, em graus: ', theta
19 end program angle
```

Tipos de dados de usuário

- ▶ Fortran também permite ao usuário definir novos tipos de dados, denominados dados **derivados** (*derived data types*)
- ▶ Estes dados são compostos pelo agrupamento de dados de tipos primitivos, ou mesmo de outros dados derivados
- ▶ Eles equivalem a uma **struct** da linguagem C
- ▶ A sintaxe para a declaração de um tipo de dado derivado é

```
type nome_do_novo_tipo
  tipo_1 :: nome_var_1
  tipo_2 :: nome_var_2
  ..
  tipo_N :: nome_var_N
end type nome_do_novo_tipo
```

- ▶ Variáveis do nome tipo são declaradas usando a sintaxe

```
type(nome_do_novo_tipo) :: var1, var2, ..., varM
```

- ▶ Os membros do novo tipo são acessados por meio do operador `'%'`

Exemplo de uso de dados derivados

```
1 program pacient      ! Exemplifica a declaração e instânciação de um dado derivado
2   type Paciente
3       character(len=256) :: nome
4       integer           :: idade
5       real              :: peso, altura
6   end type Paciente
7
8   type(Paciente) :: p
9
10  write(*,*) 'Insira o nome do paciente: '
11  read(*,1) p%nome
12  write(*,*) 'Insira a idade, peso e altura, nesta ordem: '
13  read(*,*) p%idade, p%peso, p%altura
14  write(*,2) p%nome, p%idade
15
16 1  format (A10)
17 2  format ('Paciente "', A10, '" (' , I3, ' anos) registrado com sucesso')
18 end program pacient
```

Subrotinas e Funções

- ▶ Em Fortran, uma **subrotina** difere de uma **função** no sentido de que não possui um valor de retorno
- ▶ Ambas podem ser declaradas no próprio arquivo do programa, ou em arquivos separados
- ▶ Funções são invocadas da mesma maneira que as funções intrínsecas da linguagem
- ▶ As subrotinas são invocadas por meio de um comando **call**
- ▶ A comunicação entre o programa e as funções e subrotinas se dá por meio de **argumentos** (ou **parâmetros**) e do **retorno**, no caso das funções
- ▶ Ambas são fundamentais em programas estruturados, no sentido que permite a organização e reuso de trechos de código, formando unidades semânticas

Funções

- ▶ A sintaxe para a declaração de uma função é a seguinte:

```
function nome_da_funcao(par1, par2, ..., parN)
    ! Declaração dos tipos dos argumentos
    ! Declaração das variáveis locais da função
    ! bloco de comandos
end function [nome_da_funcao]
```

- ▶ O retorno da função deve armazenado em uma variável local de mesmo nome
- ▶ O bloco de comandos pode ser encerrado por meio do comando **return**
- ▶ As variáveis são passadas por referência
- ▶ A primeira implicação deste fato é que os parâmetros devem ter o mesmo tipo da variável passada como parâmetro na chamada
- ▶ A segunda implicação é que, caso um parâmetro seja modificado na função, esta mudança será feita na variável original
- ▶ As funções devem ser declaradas a partir do ponto marcado pela palavra-chave **contains**

Exemplo de declaração e uso de funções

```
1 program binomial           ! Calcula o coeficiente binominal (n, m)
2   implicit none
3   integer(8) :: n, m
4
5   write(*,*) 'Insira os valores de n e m: '
6   read(*,*) n, m
7   write(*,*) binom(n, m)
8 contains
9   function factorial(n)
10      integer(8) :: i, n, factorial
11      factorial = 1
12
13      do i = 2, n
14         factorial = factorial * i
15      end do
16
17   end function factorial
```

Exemplo de declaração e uso de funções

```
19  function binom(n, m)
20      integer(8) :: n, m, binom
21
22      if ((n < 0) .or. (m < 0) .or. (n < m)) then
23          binom = 0
24          return
25      end if
26
27      binom = factorial(n) / (factorial(m) * factorial(n - m))
28
29  end function binom
30
31 end program binomial
```


Subrotinas

- ▶ A sintaxe para a declaração de subrotinas é semelhante à declaração de funções:

```
subroutine nome_da_subrotina(par1, par2, ..., parN)
    ! Declaração dos tipos dos argumentos
    ! Declaração das variáveis locais da subrotina

    ! bloco de comandos
end subroutine [nome_da_subrotina]
```

- ▶ Assim como as funções, as subrotinas recebem os valores de seus argumentos por referência, o que permite a modificação destes parâmetros
- ▶ Não há retorno em subrotinas
- ▶ As subrotinas também devem ser declaradas após a palavra-chave **contains**, e encerradas a qualquer momento por meio do comando **return**

Exemplo de declaração e uso de subrotinas

```
1 program selection                ! Implementa o selection sort
2   implicit none
3   integer, allocatable :: xs(:)
4   integer :: n
5
6   write(*,*) 'Insira o número de elementos: '
7   read(*,*) n
8
9   allocate(xs(n))
10
11  write(*,*) 'Insira os elementos do vetor: '
12  read(*,*) xs
13
14  call sort(n, xs)
15
16  write(*,*) xs
17
18  deallocate(xs)
```

Exemplo de declaração e uso de subrotinas

```
20 contains
21     subroutine sort(n, xs)
22         integer :: i, j, k, n, xs(:)
23
24         do i = 1, n - 1
25             j = i
26
27             do k = i + 1, n
28                 if (xs(k) < xs(j)) then
29                     j = k
30                 end if
31             end do
32
33             call swap(xs(i), xs(j))
34         end do
35
36     end subroutine sort
```

Exemplo de declaração e uso de subrotinas

```
38  subroutine swap(x, y)
39      integer :: x, y, z
40
41      z = x
42      x = y
43      y = z
44  end subroutine swap
45
46 end program selection
```

Observações sobre funções e subrotinas

- ▶ No caso em que um dos parâmetros é um vetor `xs` de tamanho desconhecido, as dimensões deste pode ser obtido por meio da função intrínseca `size()`
- ▶ Para tal, na declaração do tipo de parâmetro esta dimensão desconhecida deve ser indicada (por exemplo, `integer :: xs(:)`)
- ▶ Como um subrotina pode usar um parâmetro tanto para entrada como para saída, o uso de cada parâmetro pode ser explicitado por meio do atributo `intent`
- ▶ O parâmetro atributo é um dentre três valores possíveis: `in`, `out` e `inout`
- ▶ Além de melhorar a legibilidade, este atributo previne que um parâmetro de entrada seja modificado
- ▶ O atributo `save` pode ser utilizado para marcar variáveis locais que mantêm seus valores entre as chamadas de uma função ou subrotina

Exemplo de subrotina em Fortran

```
1 program fibonacci      ! Obtém os n próximos números de Fibonacci
2   write(*,1) next_fib(8, .false.)
3   write(*,2) next_fib(5, .false.)
4   write(*,3) next_fib(10, .true.)
5
6 1   format ('8 primeiros números de Fibonacci:', 8I3)
7 2   format ('5 próximos números de Fibonacci:', 5I4)
8 3   format ('10 primeiros números de Fibonacci:', 10I3)
9
10 contains
11
12   function next_fib(n, reset)
13     integer :: n, i, next_fib(n)
14     logical :: reset
15     integer, save :: a = 0, b = 1
16
```

Exemplo de subrotina em Fortran

```
17      if (reset) then
18          a = 0
19          b = 1
20      end if
21
22      do i = 1, n
23          next_fib(i) = b
24          b = a + b
25          a = next_fib(i)
26      end do
27
28  end function next_fib
29
30 end program fibonacci
```

Organização do programa em módulos

- ▶ Os **módulos** são uma importante característica da programação estruturada, permitindo o agrupamento lógico de trechos de código semanticamente relacionados
- ▶ As linguagens de programação que suportam o paradigma estruturado, em geral, permitem a separação de módulos em arquivos distintos
- ▶ Os módulos também podem oferecer controle, completo ou parcial, de acesso às variáveis, funções e subrotinas definidas no módulo
- ▶ A possibilidade de um módulo importar outros módulos favorece o reuso de código e a construção de bibliotecas de funções e subrotinas
- ▶ Além disso, no caso de linguagens compiladas, os módulos podem ser pré-compilados, acelerando o processo de compilação do programa e facilitando a depuração e manutenção

Declaração e implementação de módulos em Fortran

- ▶ A sintaxe para a declaração de um módulo em Fortran é a seguinte:

```
module nome_do_modulo
    ! comandos
    [contains ! subrotinas e funções do módulo ]
end module [nome_do_modulo]
```

- ▶ O bloco do programa, as funções e as subrotinas podem acessar o módulo por meio do comando **use**:

```
use nome_do_modulo
```

- ▶ A declaração de funções e subrotinas é opcional
- ▶ Cada módulo deve estar em um arquivo separado
- ▶ Os módulos devem ser compilados com a *flag* '-c'
- ▶ O acesso às variáveis do módulo pode ser controlado por meio dos atributos **private** e **public**

Exemplo de uso de módulos em Fortran

```
1 program is_prime      ! Verifica se o inteiro n é ou não primo
2   use primes
3   integer :: n
4
5   write(*, 1, advance="no")
6   read(*,*) n
7
8   if (primality_check(n)) then
9       write(*, 2) n
10  else
11      write(*, 3) n
12  end if
13
14  write(*,4) get_primes(10)
15 1  format('Insira o inteiro n: ')
16 2  format(I7, ' é primo')
17 3  format(I7, ' não é primo')
18 4  format('10 primeiros primos:', 10I4)
19 end program is_prime
```

Exemplo de uso de módulos em Fortran

```
1 module primes      ! Módulo com funções relacionadas a números primos
2   implicit none
3   integer, parameter :: max_value = 10 ** 7
4   logical, private :: sieve(max_value), ready = .false.
5
6   ! Define a visibilidade das funções e subrotinas
7   public :: primality_check, get_primes
8   private :: erasthotenes
9
10 contains
11   function primality_check(n)
12     integer :: n
13     logical :: primality_check
14
15     if (n > max_value) then
16       write(*,*) 'Max value exceeded!'
17       call exit(-1)      ! Aborta o programa com erro
18     end if
```

Exemplo de uso de módulos em Fortran

```
20      if (n < 1) then
21          primality_check = .false.
22          return
23      end if
24
25      if (.not. ready) then
26          call erasthotenes()
27      end if
28
29      primality_check = sieve(n)
30  end function primality_check
31
32  subroutine erasthotenes()      ! Implementa o crivo de Erastótenes
33      integer(8) :: i, j, step = 4
34
35      sieve = .true.
36      sieve(1) = .false.
37
```

Exemplo de uso de módulos em Fortran

```
38      do i = 4, max_value, 2
39          sieve(i) = .false.
40      end do
41
42      do i = 6, max_value, 3
43          sieve(i) = .false.
44      end do
45
46      do i = 5, max_value, step
47
48          if (sieve(i)) then
49              do j = i*i, max_value, 2*i
50                  sieve(j) = .false.
51              end do
52          end if
53
54          step = 6 - step
55      end do
56  end subroutine erasthotenes
```

Exemplo de uso de módulos em Fortran

```
58  function get_primes(n)
59      integer :: i, n, total = 0, get_primes(n)
60
61      if (.not. ready) then
62          call erasthotenes()
63      end if
64
65      do i = 1, max_value
66          if (sieve(i)) then
67              total = total + 1
68              get_primes(total) = i
69          end if
70
71          if (total == n) then
72              return
73          end if
74      end do
75  end function get_primes
76 end module primes
```

Programação Procedural

- ▶ A ideia que originou a programação procedural surgiu por volta de 1958, antes do paradigma estruturado estar completamente estabelecido
- ▶ O objetivo era diminuir a complexidade dos programas, dividindo-os em unidades menores
- ▶ Cada unidade, denominada **procedimento**, era responsável por uma única tarefa
- ▶ Estes procedimentos seriam equivalentes aos verbos nas linguagens naturais
- ▶ Este paradigma se desenvolveu como uma evolução do paradigma estruturado, sendo comum usar ambos termos combinados (programação estruturada/procedural) ou mesmo como sinônimos

Exemplo de linguagem procedural: C

- ▶ A linguagem C foi desenvolvida em 1972 por Ken Thompson e Dennis Ritchie
- ▶ Ela combina construtos de alto nível com elementos de baixo nível (ponteiros, `goto`, etc)
- ▶ A partir do padrão estabelecido em 1988, os códigos escritos em C se tornaram portáteis para todas as plataformas que tivessem um compilador C
- ▶ C é uma linguagem adequada para programação de sistemas operacionais, compiladores, jogos e aplicações comerciais, e tem sido amplamente utilizada desde sua criação
- ▶ Por exemplo, na linguagem C, o programa é representado pela função

```
int main(int argc, const char *argv[])  
{  
    // comandos do programa  
}
```


Exemplo de linguagem procedural: C

- ▶ O retorno da função `main()` é capturado pelo sistema operacional
- ▶ Zero significa que o programa finalizou sua execução com sucesso; qualquer outro valor representa um possível erro na execução
- ▶ Os dois parâmetros (os quais podem ser omitidos) representam o número de parâmetros (`argc`) e os parâmetros (`argv`) passados em linha de comando
- ▶ As subrotinas são declaradas sem retorno:

```
void nome_da_subrotina(tipo1 arg1, ..., tipoN, argN)
{
    // comandos da subrotina
}
```

- ▶ As funções e subrotinas podem invocar outras funções e subrotinas, ou mesmo a si próprias
- ▶ Uma função/subrotina que invoca a si mesma é chamada **recursiva**
- ▶ Em C, os parâmetros de uma função/subrotina são passados por **cópia**, de modo que parâmetros de saída devem ser ponteiros

Exemplo de linguagem procedural: C

- ▶ Cada função/subrotina deve ser definida/implementada em um único ponto, mas pode ser declarada em um arquivo diferente
- ▶ A sintaxe para a declaração de uma função é

```
tipo_do_retorno nome_da_funcao(tipo1 par1, ..., tipoN parN);
```

- ▶ Para a definição da função, a sintaxe é

```
tipo_do_retorno nome_da_funcao(tipo1 par1, ..., tipoN parN)
{
    // comandos da função
}
```

- ▶ O tipo do retorno e dos parâmetros é um dos tipos primitivos (**char**, **int**, **float** e **double**), ou ponteiros para estes tipos (ou para o tipo **void**), ou tipos definidos pelo usuário (**struct** ou **union**)
- ▶ O valor a ser retornado deve seguir o comando **return**
- ▶ Para referenciar uma função implementada em outro arquivo, a declaração deve ser antecedida pela palavra reservada **extern**

Exemplo de código procedural escrito em C

```
1 /* Calcula o troco mínimo de C centavos. O sistema utilizado não é canônico */
2 #include <stdio.h>
3 #include "coin_change.h"
4
5 int main() {
6     const int coins[] = { 1, 4, 5, 9, 14, 19 }, N = 6;
7     int C, xs[N], i;
8
9     printf("Insira o valor do troco: ");
10    scanf("%d", &C);
11    coin_change(xs, C, N, coins);
12
13    printf("Troco para %d centavos:\n", C);
14    for (i = 0; i < N; ++i)
15        if (xs[i])
16            printf("%d moeda(s) de %d centavo(s)\n", xs[i], coins[i]);
17
18    return 0;
19 }
```

Exemplo de código procedural escrito em C

```
1 /* As diretivas de pré-processador ifndef, define e endif permitem a múltipla inclusão
2  * deste arquivo via #include sem erro de duplicidade de declarações */
3 #ifndef COIN_CHANGE_H
4 #define COIN_CHANGE_H
5
6 /* Declaração da função */
7 extern void coin_change(int *xs, int C, int N, const int *coins);
8
9 #endif
```

Exemplo de código procedural escrito em C

```
1 #include <string.h>
2 #include "coin_change.h"
3
4 #define MAX 1000
5 #define oo 1000000001
6
7 /* st[c][i] = mínimo de moedas para troco c usando as i primeiras moedas
8    de coins; ps[C][i] marca se a moeda coins[i] foi escolhida ou não */
9 static int st[MAX][MAX], ps[MAX][MAX];
10
11 /* Solução do coin change usando programação dinâmica */
12 static int dp(int c, int i, const int *coins)
13 {
14     int res;
15
16     /* Caso base: troco vazio */
17     if (c == 0)
18         return 0;
```

Exemplo de código procedural escrito em C

```
20  /* Troco pendente, sem opções de moedas */
21  if (i == 0)
22      return oo;
23
24  /* Consulta a resultados já computados */
25  if (st[c][i] != -1)
26      return st[c][i];
27
28  /* Não escolhe a moeda coins[i] */
29  res = dp(c, i - 1, coins);
30  ps[c][i] = 0;
31
32  if (coins[i - 1] <= c) {
33      /* Escolhe uma moeda com valor coins[i] */
34      int r = dp(c - coins[i - 1], i, coins) + 1;
35
36      if (r < res) {
37          res = r;
38          ps[c][i] = 1;
39      }
40  }
```

Exemplo de código procedural escrito em C

```
42     /* Memorização */
43     st[c][i] = res;
44
45     return res;
46 }
47
48 void coin_change(int *xs, int C, int N, const int *coins)
49 {
50     int p, i = N;
51
52     /* Inicializa as tabelas e o vetor xs */
53     memset(st, -1, sizeof st);
54     memset(ps, -1, sizeof ps);
55     memset(xs, 0, N * sizeof(int));
56
57     /* Resolve o problema */
58     dp(C, N, coins);
59
60     /* Resgata as moedas utilizadas */
61     p = ps[C][N];
```

Exemplo de código procedural escrito em C

```
61     p = ps[C][N];
62
63     while (i > 0 && p != -1)
64     {
65         if (p == 0)
66             i--;
67         else
68         {
69             xs[i - 1]++;
70             C -= coins[i - 1];
71         }
72
73         p = ps[C][i];
74     }
75 }
```


Referências

1. **annefou**. [Why Derived Data Types?](#), acesso em 10/02/2020.
2. **CHEUNG**, Shun Yan. [Loops \(DO, DO WHILE, EXIT, CYCLE\)](#), acesso em 04/02/2020.
3. GNU Fortran. [IAND – Bitwise logical and](#), acesso em 04/02/2020.
4. GNU Fortran. [ISHFT – Shift bits](#), acesso em 04/02/2020.
5. GNU Fortran. [Exit the program with status](#), acesso em 10/02/2020.
6. **PADMAN**, Rachael. [Computer Physics: Self-study guide 2 – Programming in Fortran 95](#), University of Cambridge, Departament of Physics, 2007.
7. **SHALOM**, Elad. *A Review of Programming Paradigms Throughtout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
8. **SHENE**, C. K. [SELECT CASE Statement](#), acesso em 04/02/2020.