

# Programação Estruturada

## Estruturas e Funções

**Prof. Edson Alves**

Faculdade UnB Gama

2020

# 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 ! Calcula o imposto de renda mensal
2 program IRRF
3
4     implicit none
5
6     real :: salario, aliquota, deducacao, imposto
7
8     write(*,*) 'Insira o salário mensal: '
9     read(*,*) salario
10
11     ! Determinar a aliquota e a dedução a partir do salário
12     if (salario <= 1903.98) then
13         aliquota = 0
14     else if (salario <= 2826.65) then
15         aliquota = 0.075
16         deducacao = 142.80
17     else if (salario <= 3751.05) then
18         aliquota = 0.15
19         deducacao = 354.80
20     else if (salario <= 4664.68) then
21         aliquota = 0.225
22         deducacao = 636.13
```

## Exemplo de uso do construto IF-ELSE

```
23     else
24         aliquota = 0.275
25         deducacao = 869.36
26     end if
27
28     ! Imprime o imposto a ser pago
29     if (aliquota == 0) then
30         write(*,*) 'Isento'
31     else
32         imposto = salario * aliquota - deducacao
33         write(*,1) imposto
34     end if
35
36 1   format ('Imposto devido: ', F9.2)
37
38 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 de rótulos 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'
19        case default
20            write(*,*) 'Idade inválida!'
21            return
22    end select
```



## Exemplo de uso do construto SELECT-CASE

```
23  
24     ! Imprime a prioridade do paciente  
25     write(*,1) idade, prioridade  
26  
27 1   format (I3, ' anos, prioridade: ', A6)  
28  
29 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, factorial = 1
7
8     read(*,*) n
9
10    do i = 1, n
11        factorial = factorial * i
12    end do
13
14    write(*,1) n, factorial
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 ! Computa a^n com complexidade O(log n)
2 program fast_exp
3
4     implicit none
5
6     integer(16) :: a, n, res = 1, base      ! Inteiros de 128-bits
7     read(*,*) a, n
8
9     base = a
10
11     do while (n > 0)
12         if (iand(n, 1) > 0) then            ! iand(x, y) = x & y
13             res = res * base
14         end if
15
16         base = base * base
17         n = ishft(n, -1)                    ! n = n >> 1
18     end do
19
20     write(*,*) res
21
22 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 ! Computa a média e o desvio padrão dos elementos do vetor xs
2 program statistics
3
4     implicit none
5
6     integer, allocatable :: xs(:)    ! Vetor dinâmico
7     integer :: n, i                  ! n = dimensão de xs
8     real :: stats(2)                 ! Vetor com duas posições
9
10    write(*,*) 'Insira o número de entradas: '
11    read(*,*) n
12
13    if (n < 1) then
14        return
15    end if
16
17    allocate(xs(n))
18
19    do i = 1, n
20        write(*,*) 'Insira a entrada ', i, ': '
21        read(*,*) xs(i)
22    end do
```

# Exemplo de uso de vetores

```
24 stats(1) = 0.0 ! Média
25
26 do i = 1, n
27     stats(1) = stats(1) + xs(i)
28 end do
29
30 stats(1) = stats(1) / n
31
32 stats(2) = 0.0 ! Desvio-padrão
33
34 do i = 1, n
35     stats(2) = stats(2) + (xs(i) - stats(1)) ** 2
36 end do
37
38 write(*,*) 'Média = ', stats(1)
39
40 stats(2) = sqrt(stats(2)/n)
41
42 write(*,*) 'Desvio = ', stats(2)
43
44 deallocate(xs)
45
46 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 ! Calcula o ângulo entre dois vetores
2 program angle
3
4     real, parameter :: pi = acos(-1.0)
5     real :: theta, xlen, ylen
6     real :: xs(2) = (/ 1, 0 /), ys(2) = 1    ! ys = (1, 1)
7     real :: A(2, 2) = reshape((/ 0, 1, -1, 0 /), (/ 2, 2 /))
8
9     xlen = sqrt(dot_product(xs, xs))
10    ylen = sqrt(dot_product(ys, ys))
11    theta = acos(dot_product(xs, ys) / (xlen * ylen))
12    theta = theta * 180 / pi
13
14    write(*,*) 'Ângulo, em graus: ', theta
15
16    ys = matmul(A, ys)
17    theta = acos(dot_product(xs, ys) / (xlen * ylen))
18    theta = theta * 180 / pi
19
20    write(*,*) 'Ângulo após rotação, em graus: ', theta
21
22 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 ! Exemplifica a declaração e instânciação de um dado derivado
2 program pacient
3
4     type Paciente
5         character(len=256) :: nome
6         integer           :: idade
7         real              :: peso, altura
8     end type Paciente
9
10    type(Paciente) :: p
11
12    write(*,*) 'Insira o nome do paciente: '
13    read(*,1) p%nome
14    write(*,*) 'Insira a idade, peso e altura, nesta ordem: '
15    read(*,*) p%idade, p%peso, p%altura
16
17    write(*,2) p%nome, p%idade
18
19 1   format (A10)
20 2   format ('Paciente "', A10, '" (' , I3, ' anos) registrado com sucesso')
21
22 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 da função
- ▶ O bloco de comandos pode ser encerrado prematuramente, 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 ! Calcula o coeficiente binominal (n, m)
2 program binomial
3
4     implicit none
5     integer(8) :: n, m
6
7     write(*,*) 'Insira os valores de n e m: '
8     read(*,*) n, m
9
10    write(*,*) binom(n, m)
11
12 contains
13
14    function factorial(n)
15        integer(8) :: i, n, factorial
16        factorial = 1
17
18        do i = 2, n
19            factorial = factorial * i
20        end do
21
22    end function factorial
```

## Exemplo de declaração e uso de funções

```
23
24  function binom(n, m)
25      integer(8) :: n, m, binom
26
27      if ((n < 0) .or. (m < 0) .or. (n < m)) then
28          binom = 0
29          return
30      end if
31
32      binom = factorial(n) / (factorial(m) * factorial(n - m))
33
34  end function binom
35
36 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 ! Implementa o selection sort
2 program selection
3
4     implicit none
5
6     integer, allocatable :: xs(:)
7     integer :: n
8
9     write(*,*) 'Insira o número de elementos: '
10    read(*,*) n
11
12    allocate(xs(n))
13
14    write(*,*) 'Insira os elementos do vetor: '
15    read(*,*) xs
16
17    call sort(n, xs)
18
19    write(*,*) xs
20
21    deallocate(xs)
22
```

## Exemplo de declaração e uso de subrotinas

```
23 contains
24
25     subroutine sort(n, xs)
26
27         integer :: i, j, k, n, xs(:)
28
29         do i = 1, n - 1
30             j = i
31
32             do k = i + 1, n
33                 if (xs(k) < xs(j)) then
34                     j = k
35                 end if
36             end do
37
38             call swap(xs(i), xs(j))
39         end do
40
41     end subroutine sort
42
```

# Exemplo de declaração e uso de subrotinas

```
43  subroutine swap(x, y)
44
45      integer :: x, y, z
46
47      z = x
48      x = y
49      y = z
50
51  end subroutine swap
52
53  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 mantém seus valores entre as chamadas de uma função ou subrotina

## Exemplo de subrotina em Fortran

```
1 ! Obtém os n próximos números de Fibonacci
2 program fibonacci
3
4     write(*,1) next_fib(8, .false.)
5     write(*,2) next_fib(5, .false.)
6     write(*,3) next_fib(10, .true.)
7
8 1   format ('8 primeiros números de Fibonacci:', 8I3)
9 2   format ('5 próximos números de Fibonacci:', 5I4)
10 3   format ('10 primeiros números de Fibonacci:', 10I3)
11
12 contains
13
14     function next_fib(n, reset)
15         integer :: n, i, next_fib(n)
16         logical :: reset
17         integer, save :: a = 0, b = 1
18
19         if (reset) then
20             a = 0
21             b = 1
22         end if
```

## Exemplo de subrotina em Fortran

```
23
24     do i = 1, n
25         next_fib(i) = b
26         b = a + b
27         a = next_fib(i)
28     end do
29
30     end function next_fib
31
32 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 ! Verifica se o inteiro n é ou não primo
2 program is_prime
3     use primes
4     integer :: n
5
6     write(*, 1, advance="no")
7     read(*,*) n
8
9     if (primalty_check(n)) then
10         write(*, 2) n
11     else
12         write(*, 3) n
13     end if
14
15     write(*,4) get_primes(10)
16
17 1 format('Insira o inteiro n: ')
18 2 format(I7, ' é primo')
19 3 format(I7, ' não é primo')
20 4 format('10 primeiros primos:', 10I4)
21
22 end program is_prime
```

## Exemplo de uso de módulos em Fortran

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

## Exemplo de uso de módulos em Fortran

```
23
24     if (n < 1) then
25         primality_check = .false.
26         return
27     end if
28
29     if (.not. ready) then
30         call erasthotenes()
31     end if
32
33     primality_check = sieve(n)
34
35 end function primality_check
36
37 ! Implementa o crivo de Erastótenes
38 subroutine erasthotenes()
39
40     integer(8) :: i, j, step = 4
41
42     sieve = .true.
43     sieve(1) = .false.
44
```

## Exemplo de uso de módulos em Fortran

```
45     do i = 4, max_value, 2
46         sieve(i) = .false.
47     end do
48
49     do i = 6, max_value, 3
50         sieve(i) = .false.
51     end do
52
53     do i = 5, max_value, step
54
55         if (sieve(i)) then
56             do j = i*i, max_value, 2*i
57                 sieve(j) = .false.
58             end do
59         end if
60
61         step = 6 - step
62
63     end do
64
65     end subroutine erasthotenes
66
```

## Exemplo de uso de módulos em Fortran

```
67  function get_primes(n)
68
69      integer :: i, n, total = 0, get_primes(n)
70
71      if (.not. ready) then
72          call erasthotenes()
73      end if
74
75      do i = 1, max_value
76          if (sieve(i)) then
77              total = total + 1
78              get_primes(total) = i
79          end if
80
81          if (total == n) then
82              return
83          end if
84      end do
85
86  end function get_primes
87
88  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**

## Exemplo de linguagem procedural: C

- ▶ Cada função/subrotina deve ser definida/implementada em um único ponto, mas pode ser declarada ou invocada quantas vezes forem necessárias

- ▶ 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 utilizando as moedas disponíveis
2  * em coins. O sistema utilizado não é canônico */
3 #include <stdio.h>
4 #include "coin_change.h"
5
6 int main() {
7     const int coins[] = { 1, 4, 5, 9, 14, 19 }, N = 6;
8     int C, xs[N], i;
9
10    printf("Insira o valor do troco: ");
11    scanf("%d", &C);
12
13    coin_change(xs, C, N, coins);
14
15    printf("Troco para %d centavos:\n", C);
16
17    for (i = 0; i < N; ++i)
18        if (xs[i])
19            printf("%d moeda(s) de %d centavo(s)\n", xs[i], coins[i]);
20
21    return 0;
22 }
```

## Exemplo de código procedural escrito em C

```
1 /* As diretivas de pré-processador ifndef, define e endif permitem
2  * a múltipla inclusão deste arquivo via #include sem erro de
3  * duplicidade de declarações */
4 #ifndef COIN_CHANGE_H
5 #define COIN_CHANGE_H
6
7 /* Declaração da função */
8 extern void coin_change(int *xs, int C, int N, const int *coins);
9
10 #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;
19
20     /* Troco pendente, sem opções de moedas */
21     if (i == 0)
22         return oo;
```

## Exemplo de código procedural escrito em C

```
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  {
34      /* Escolhe uma moeda com valor coins[i] */
35      int r = dp(c - coins[i - 1], i, coins) + 1;
36
37      if (r < res)
38      {
39          res = r;
40          ps[c][i] = 1;
41      }
42  }
43
44  /* Memorização */
45  st[c][i] = res;
```

## Exemplo de código procedural escrito em C

```
46
47     return res;
48 }
49
50 void coin_change(int *xs, int C, int N, const int *coins)
51 {
52     int p, i = N;
53
54     /* Inicializa as tabelas e o vetor xs */
55     memset(st, -1, sizeof st);
56     memset(ps, -1, sizeof ps);
57     memset(xs, 0, N * sizeof(int));
58
59     /* Resolve o problema */
60     dp(C, N, coins);
61
62     /* Resgata as moedas utilizadas */
63     p = ps[C][N];
64
65     while (i > 0 && p != -1)
66     {
67         if (p == 0)
68             i--;
```

## Exemplo de código procedural escrito em C

```
69     else
70     {
71         xs[i - 1]++;
72         C -= coins[i - 1];
73     }
74
75     p = ps[C][i];
76 }
77 }
```



# 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, Department of Physics, 2007.
7. **SHALOM**, Elad. *A Review of Programming Paradigms Throughout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
8. **SHENE**, C. K. [SELECT CASE Statement](#), acesso em 04/02/2020.