

Programação Estruturada

Conceitos Elementares

Prof. Edson Alves

Faculdade UnB Gama

2020

Sumário

1. Programação Estruturada
2. Fortran
3. Variáveis
4. Entrada e Saída

Principais características

- ▶ A programação estruturada é um paradigma que tem como ideia principal aumentar a clareza e a legibilidade do código através da organização baseada em blocos
- ▶ Outra ideia associada à programação estruturada é evitar o uso de saltos a pontos arbitrários (por exemplo, o comando **goto** da linguagem C/C++)
- ▶ Os saltos, portanto, só poderiam ser feito por meio de construtos pré-definidos
- ▶ Em 1964, Corrado Böhm e Giuseppe Jacopini apresentaram um artigo demonstrando que apenas três estruturas de controle eram necessárias para escrever qualquer programa
- ▶ O teorema proposto e demonstrado tornava o salto incondicional para um ponto arbitrário do código desnecessário

Teorema de Böhm-Jacopini

Teorema da programação estruturada

É possível criar um programa combinando subprogramas de apenas três maneiras diferentes:

1. Executar um subprograma, e em seguida, outro subprograma (**sequência**)
2. Executar um dentre dois subprogramas possíveis, de acordo com uma expressão booleana (**seleção**)
3. Executar um subprograma enquanto uma expressão booleana é verdadeira (**iteração**)

Evolução da programação estruturada

- ▶ Embora o teorema de Böhm-Jacopini tem sido inicialmente ignorado, a carta de Edsger Dijkstra, intitulada "*Go To Statement Considered Harmful*", de 1968, foi um marco na história da computação
- ▶ Dijkstra prega que os programas deveriam ser organizados de uma maneira sistemática, chamada *programação estruturada*
- ▶ Em 1972 o New York Times conclui o primeiro projeto de larga escala, bem sucedido, desenvolvido sobre o paradigma da programação estruturada
- ▶ Este projeto, e as palestras proferidas por Edward Yourdon nos anos 1970 ajudaram a difundir a programação estruturada

Metodologia de desenvolvimento de programas estruturados

Definição

A programação estruturada é um método que reduz a complexidade de um programa por meio de

1. Análise *top-down* para a solução do problema
2. Uso de modularização para a estrutura e organização do programa
3. Uso de código estruturado para os módulos individuais

Metodologia de desenvolvimento de programas estruturados

- ▶ A análise *top-down* inclui a resolução do problema e da listagem das instruções que compõem esta solução
- ▶ Se o problema for muito complexo para ser resolvido diretamente, ele é dividido em subproblemas menores, cujas soluções podem ser combinadas para formar a solução do problema original
- ▶ A modularização permite dividir o programa em partes menores, denominadas **módulos**, **subrotinas** ou **subprogramas**
- ▶ Cada módulo é responsável por uma única tarefa
- ▶ O código estruturado organiza as instruções da solução por meio das **estruturas de controle**
- ▶ Estas estruturas definem a ordem de execução das instruções: sequencialmente, condicionalmente (seleção) ou repetidamente (iteração)

Metodologia de desenvolvimento de programas estruturados

- ▶ As estruturas de controle podem conter seleções e repetições, mas devem ter, externamente, apenas um ponto de entrada e um ponto de saída
- ▶ O módulo principal do programa resolve o problema
- ▶ Caso ele precise realizar uma subtarefa, esta é executada por um módulo, o qual é invocado a partir do módulo principal
- ▶ Uma boa prática em códigos estruturados é de utilizar nomes significativos para variáveis e subrotinas, de modo a ampliar a compreensão da semântica do código
- ▶ Neste sentido, também é recomendado o uso de comentários
- ▶ Por fim, é preferível utilizar tipos agregados de dados que, embora possam ter tipos distintos, representam um mesmo conceito ou ideia, do que listar cada variável individualmente

Subrotinas

Definição

Uma **subrotina** é composta por uma sequência de instruções, agrupadas sob um mesmo nome, e que em conjunto, e na ordem especificada, são capazes de realizar uma tarefa.

Características das subrotinas

- ▶ Subrotinas podem ser definidas no próprio programa, ou em arquivos separados (por exemplo, em bibliotecas)
- ▶ A depender da linguagem, as subrotinas podem ser chamadas de **procedimentos**, **funções**, **rotinas**, **métodos** ou **subprogramas**
- ▶ O termo geral é **unidade invocável** (*callable unit*)
- ▶ As subrotinas se comportam como programas independentes, mas são codificadas de modo que seja possível invocá-las quantas vezes forem necessárias
- ▶ Elas podem invocar ou serem invocadas por outras subrotinas
- ▶ Uma vez finalizada a subrotina, a execução do programa segue para a instrução que sucede a chamada da subrotina
- ▶ O **corpo** da subrotina contém o código que será executado a cada invocação

Características das subrotinas

- ▶ O código que invoca a subrotina pode ser comunicar com esta por meio de **parâmetros**, que são uma lista de variáveis, definidas pela rotina, necessárias para sua execução
- ▶ Os valores atribuídos, em uma chamada, para cada parâmetro, são denominados **argumentos**
- ▶ Subrotina podem ou não retornar valores após a sua execução: este valor, se existir, é o **retorno** da subrotina
- ▶ Algumas linguagem denominam **função** uma subrotina com retorno, e **procedimentos** as subrotinas sem retorno
- ▶ Um subrotina que invoca a si mesma é denominada rotina **recursiva**
- ▶ Uma convenção comum é que o nome de uma subrotina deve ser um **verbo**, que indique a tarefa que será realizada
- ▶ Idealmente, uma subrotina deve executar uma única tarefa e depender o mínimo possível de outras subrotinas

Blocos

Definição

Um **bloco** é uma seção de código delimitada, que consiste de declarações, comandos e expressões.

Características dos blocos

- ▶ As estruturas de controle tem blocos associados às suas ações
- ▶ Os blocos permitem que grupos de comandos correlacionados sejam entendidos como um único comando maior, que representa a ação coletiva de todos os seus comandos
- ▶ Eles também restringem o escopo das variáveis, procedimentos e funções, evitando conflitos de nomes
- ▶ O início e o final de um bloco pode ser delimitado por palavras-chave ou por símbolos
- ▶ Por exemplo, algumas linguagens delimitam os blocos por meio das palavras-chave **begin** e **end**
- ▶ Outras linguagens marcam os blocos por caracteres (por exemplo, '{' e '}') ou por indentação

Fortran

- ▶ Fortran (*IBM Mathematical FORMula TRANslation System*) é uma linguagem de programação desenvolvida na década de 1950
- ▶ Até os dias atuais é uma das principais (ou a principal) linguagem utilizada em programação científica
- ▶ O primeiro compilador foi desenvolvido na IBM, por uma equipe liderada por John W. Backus, nos anos de 1954 a 1957
- ▶ O ISO/IEC 1539-1:1997 contém o padrão Fortran 95, um dos mais populares da linguagem
- ▶ Fortran apresenta notável performance em computação numérica, o que levou a sua adoção em pesquisas científicas e aplicações computacionalmente intensivas, como meteorologia, física, engenharia, etc

GFortran

- ▶ O projeto GNU Fortran (GFortran) consiste em um *front-end* de compilador e bibliotecas de *run-time* para o GCC que dêem suporte à linguagem Fortran
- ▶ Ele é totalmente compatível com o padrão Fortran 95 e inclui suporte legado ao formato Fortran 77
- ▶ Em distribuições Linux com suporte ao apt, ele pode ser instalado com o comando

```
$ sudo apt-get install gfortran
```

- ▶ Para testar a instalação, insira o seguinte comando no terminal:

```
$ f95 -v
```

Hello World!

```
1 ! Implementação do Hello World em Fortran
2 program hello
3
4     write(*,*) 'Hello, World!'
5
6 end program hello
```


Compilação, linkedição e execução

- ▶ Para compilar um código Fortran (extensões .f90) é preciso invocar o GFortran, utilizando a *flag* -c:

```
$ f95 -c hello.f90
```

- ▶ No processo de linkedição é preciso indicar, os código-objetos que comporão o executável e, opcionalmente, o nome deste executável (opção -o):

```
$ f95 hello.o -o hello
```

- ▶ É possível executar ambas etapas em um só comando:

```
$ f95 hello.f90 -o hello
```

- ▶ Para rodar o executável criado, basta usar os mesmos mecanismos disponíveis em Linux para invocar um programa como, por exemplo, indicar seu caminho:

```
$ ./hello
```

Variáveis em Fortran

- ▶ Em Fortran, as variáveis simbolizam regiões de memória, as quais podem ser lidas ou escritas
- ▶ Cada variável é identificada por um **nome**, que deve iniciar com um caractere alfabético e conter apenas caracteres alfanuméricos ou o símbolo '_'
- ▶ Em Fortran não há distinção entre caracteres maiúsculos e minúsculos
- ▶ Assim como nas linguagens imperativas, uma variável identifica tanto o endereço da região de memória quanto o valor armazenado
- ▶ Qual dos dois valores será utilizado depende do contexto (se é um *l-value* ou um *r-value*)

Declaração de variáveis e tipos de dados

- ▶ Uma variável pode ser declarada em Fortran usando a seguinte sintaxe

```
tipo_do_dado :: nome_da_variavel [= valor_inicial]
```

- ▶ Os principais tipos de dados em Fortran são: **real**, **integer**, **complex** e **character**
- ▶ O valor inicial é opcional
- ▶ Strings podem ser declaradas indicando-se o número de caracteres que a compõe

```
character (len = N) :: s      ! string de N caracteres
```

- ▶ Para declarar **constantes**, isto é, variáveis com permissão para leitura apenas), é utilizada a palavra-chave **parameter**:

```
complex, parameter :: pi = 3.1415
```

- ▶ No caso de constantes, o valor inicial é mandatório
- ▶ A expressão **implicit none** determina que todas as variáveis devem ser declaradas antes de seu uso, e é boa prática sempre utilizá-la no início dos programas

Exemplo de declaração e uso de variáveis em Fortran

```
1 ! Computa a área de um círculo de raio r
2 program area
3
4     implicit none
5
6     real, parameter :: pi = 3.141592      ! Declaração de constante
7     real :: A                             ! Declaração de variável real
8     integer :: r                          ! Declaração de variável inteira
9
10    r = 8                                  ! Define um valor para r por meio de atribuição
11
12    A = pi * r ** 2                        ! Área do círculo
13
14    write(*,*) 'Area = ', a
15
16 end program area
```

Operadores aritméticos e funções intrínsecas

- ▶ Sendo uma linguagem voltada para computação científica, Fortran tem suporte para uma série de operadores aritméticos
- ▶ No caso das expressões com mais de um operador, o operador de menor precedência é computado antes do de menor precedência
- ▶ Além disso, há um bom número de funções *intrínsecas* da linguagem, disponíveis sem a necessidade de importar arquivos ou bibliotecas externas
- ▶ Boa parte destas funções são relacionadas às funções matemáticas e manipulação numérica

Operador	Precedência	Operação
**	1	Expoenciação
*	2	Multiplicação
/	2	Divisão
+	3	Adição
-	3	Subtração

Funções intrínsecas úteis

Função	Retorno
abs(a)	Valor absoluto de a
sin(w)	Seno de w
cos(w)	Cosseno de w
tan(x)	Tangente de w
sqrt(x)	Raiz quadrada de x
conjg(z)	Conjugado complexo de z
log10(x)	Logaritmo em base 10 de x
mod(r1, r2)	Resto da divisão de r1 por r2
max(r1, r2, ...)	Maior dentre todos os argumentos
min(r1, r2, ...)	Menor dentre todos os argumentos

Legenda: r: real ou inteiro, z: complexo, w: real ou complexo, x: real, a: qualquer tipo.

Exemplo de uso de funções intrínsecas e operadores aritméticos

```
1 ! Computa a forma polar do complexo c
2 program polar
3
4     implicit none
5
6     complex :: c = complex (0.5, sqrt(3.0)/2)
7
8     real :: p, theta      ! Parâmetros da forma polar
9
10    write(*,*) 'c = ', c
11
12    ! Converte c para a forma polar c = p(cos(theta) + isen(theta))
13    p = sqrt(real(c) ** 2 + aimag(c) ** 2)
14    theta = atan(aimag(c)/real(c))
15
16    ! p = 1, theta = pi/3 = 60°
17    write(*,*) 'p = ', p, ' theta = ', theta
18
19 end program polar
```

Operadores lógicos e relacionais

- ▶ Fortran também tem suporte para variáveis booleanas, cujos valores possíveis são verdadeiro (`.TRUE.`) e falso (`.FALSE.`)
- ▶ As variáveis booleanas são declaradas com o tipo **logical**:
`logical :: T = .true., F = .false.`
- ▶ Variáveis booleanas ou expressões que resultem em valores booleanos podem ser combinadas com os operadores lógicos **e** (`.and.`), **ou** (`.or`) ou **não** (`.not`)
- ▶ Os operadores relacionais são apresentados em duas formas
- ▶ A primeira delas é a em notação simbólica: `<`, `<=`, `==`, `>=`, `>`, `/=`
- ▶ A segunda é por meio de com operadores semelhantes aos operadores lógicos: `.lt.`, `.le.`, `.eq.`, `.ge.`, `.gt.`, `.ne.`

Exemplo de uso de operadores relacionais

```
1 ! Verifica se um competidor pode ou não participar da Maratona
2 ! no ano de 2020
3 program maratona
4
5     implicit none
6
7     integer :: inicio = 2017, nascimento = 1995
8     logical :: primeira_graduacao = .true.
9     logical :: ok
10
11     ok = nascimento >= 1997 .or. (primeira_graduacao .and. inicio > 2015)
12
13     write(*,*) 'Pode participar? ', ok
14
15 end program maratona
```

Saída de dados

- ▶ A função **write** permite a escrita de uma lista (*list*) de dados em um fluxo (*stream*), de acordo com a formatação dada em *label*:

```
write(stream, label) list
```

- ▶ O fluxo pode ser um número associado a um arquivo, uma variável do tipo **character** ou o símbolo *, que indica o valor padrão (em geral, o terminal)
- ▶ O rótulo (*label*) é o inteiro identificador do formato, ou * para formato livre
- ▶ A sintaxe para a declaração de um rótulo é

```
label format (format_descriptors)
```

- ▶ Os descritores de formato são uma lista de itens, separados por vírgula, que determinam como a saída deve ser apresentada

Exemplos de descritores de formato

Descritor	Efeito
nIw	Imprime os próximos n inteiros, com tamanho de w caracteres cada
$nFw.d$	Imprime os próximos n números complexos ou reais, em ponto fixo, com w caracteres, e d dígitos na parte decimal
$nEw.d$	Imprime os próximos n números complexos ou reais, em ponto flutuante, com w caracteres, e d dígitos na parte decimal
Aw	Imprime a variável não-numérica A , com w caracteres de tamanho

Leitura de dados

- ▶ A função **read** permite a leitura de uma lista (`list`) de dados de um fluxo (`stream`), com os tipos especificados em `label`:

```
read(stream, label [, end=end_label][, err = err_label]) list
```

- ▶ O fluxo e as especificações de formato são as mesmas apresentadas para a função **write**
- ▶ O rótulo indicado para o parâmetro `end` indica o ponto do código para o qual a execução deve prosseguir caso a entrada seja exaurida prematuramente
- ▶ No caso do rótulo associado ao parâmetro `err` marca o ponto que inicia o tratamento de algum possível erro na leitura

Exemplo de leitura/saída de dados em Fortran

```
1 ! Calcula o valor da parcela P a ser para um financiamento de R reais,  
2 ! em N meses, com taxa de juros de j por cento ao mês, sem entrada  
3 program financiamento  
4  
5     implicit none  
6  
7     real :: R, j, P  
8     integer :: N  
9  
10    ! Leitura dos dados  
11    read(*,*) R, j, N  
12  
13    ! Cálculo da prestação  
14    P = R*(j/(1 - 1/((1 + j) ** N)))  
15  
16    ! Saída formatada  
17    write(*,1) P  
18  
19 1 format ('Prestacao: ', F9.2)  
20  
21 end program financiamento
```

Referências

1. **BURKARDT**, John. [Source Codes in Fortran 90](#), acesso em 31/01/2020.
2. **PADMAN**, Rachael. [Computer Physics: Self-study guide 2 – Programming in Fortran 95](#), University of Cambridge, Department of Physics, 2007.
3. GNU Fortran. [Welcome to the home of GNU Fortran](#), acesso em 29/01/2020.
4. OC3030 Fortran Modules. [FORTRAN: Input/Output \(I/O\)](#), acesso em 03/02/2020.
5. Matemática Didática. [Cálculo de prestações - Financiamento](#), acesso em 03/02/2020.
6. **SHALOM**, Elad. *A Review of Programming Paradigms Throughout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
7. **SHENE**, C. K. [LOGICAL Type and Variables](#), acesso em 31/01/2020.
8. Wikipédia. [Fortran](#), acesso em 29/01/2020.