Série Linguagem de Programação Científica

Fortran 95:
curso básico

Gilberto Orengo
Professor adjunto do Curso de Física Médica
Centro Universitário Franciscano–UNIFRA
http://www.orengonline.com/
orengo@orengonline.com

1ªEdição

Editora XXX

Capa
Lucas Rodrigues dos Santos

Preparação e digitação no LaTeX
Gilberto Orengo

Ilustrações no texto
Gilberto Orengo e Lucas Rodrigues dos Santos

Revisão
CCCC DDDD HHHH

Impressão
Gráfica ZZZZZZZZ
(55) xxxx.xxxx
Santa Maria–RS

Site do Livro
http://www.orengonline.com/fortran95/

Dados Internacionais de Catalogação na Publicação (CIP)
Câmara Brasileira do Livro, SP, Brasil

Orengo, Gilberto, 1961–
Fortran 95: curso básico / Gilberto Orengo. – Santa Maria: Editora da UNIFRA, 2007.

ISBN: XX-XXXX-XXX-X
1. Informática 2. Fortran 3. Linguagem de Programação

YY-XXXX CDD-ZZZ.ZZ

Arquivo LaTeX original: livrof90_v5.tex

Índices para catálogo sistemático:
1. Linguagem de programação
2. Fortran 90/95

[2007]
Espaço reservado à Editora

SUMÁRIO

Apresentação

Nesta parte será transcrito os textos de apresentação de dois pesquisadores da Área de Física e de Matemática, usuários ativos e *seniors* do Fortran.

Prefácio

(Texto provisório) Este livro é o resultado de um longo período de contato com a linguagem Fortran. Tudo começou na graduação do Curso de Física, na Universidade Federal de Santa Maria-RS (UFSM), entre 1984 e 1989, passou pelo Mestrado em Física Aplicada/UFSM, com o auxílio fundamental dos professores Cláudio de Oliveira Graça/UFSM e Manoel Siqueira, da Universidade Federal de Minas Gerais (UFMG). No doutorado em Engenharia Nuclear, pela Universidade Federal do Rio Grande do Sul (UFRGS) o passo final, em que parte da tese foi a construção de uma versão inicial de um código computacional, sob tutela do professor Marco Tulio M. B. de Vilhena/UFRGS. Também contribuíram o curso ministrado na UFRGS, para alunos da Engenharia Mecânica ao lado da professora Cynthia Segatto e, a construção de um material didático para um Minicurso de Fortran 90/95, para a Semana Acadêmica do Curso de Física Médica do Centro Universitário Franciscano (UNIFRA), em junho de 2001. Uma forte contribuição também venho das aulas da disciplina Linguagem de Programação Científica para o Curso de Física Médica/UNIFRA, desde 2002.

A minha formação em Fortran foi baseada principalmente no aprendizado autodidata. Todos os livros citados nas referências tiveram influência sobre a minha visão a respeito da linguagem Fortran e na forma de programar. Portanto, algumas características deles foram incorporadas, subconscientemente, e podem fazer parte do meu texto. Um destes livros que considero uma referência é o *Professional Programmer's Guide to Fortran77*, de Clive G. Page da *University of Leicester*, UK, 1995.

A diferença entre o Fortran 90 e o 95 é sutil. No Fortran 95 foram realizadas pequenas correções do Fortran 90 e introduzidas algumas facilidades de programação paralela, como por exemplo, o procedimento **`FORALL`**. A escolha do título do livro foi pelo Fortran 95, embora não trataremos neste texto a respeito dos avanços do Fortran 95, que fará parte da sequência do livro, numa edição mais avançada. Este livro tem um perfil introdutório e veremos os conceitos mínimos necessários para uma iniciação na linguagem Fortran, introduzindo alguns conceitos mais modernos do Fortran 90/95.

Este livro foi pensado para ser usado por estudantes que não conhecem a linguagem Fortran, bem como por aqueles que já dominam a linguagem. Os programadores que tiveram contato com o antigo e bom FORTRAN 77 terão neste livro uma atualização especialmente nos últimos quatro capítulos. É aconselhável para os iniciantes que leiam os capítulos na sequência em que são apresentados. Já os conhecedores da linguagem podem saltear entre

os capítulos, conforme a necessidade, mas é sugerido que leiam o primeiro capítulo, para entenderem algumas peculiaridades da escrita do livro.

Na página oficial do livro estarão disponíveis atualizações, correções, novidades, exercícios e problemas. Costumo evidenciar que em programação há uma diferença entre exercícios e problemas, ou entre a *solução de problemas* e *prática com exercícios*. O primeiro subentende que o segundo foi realizado exaustivamente e, assim, estaremos, em princípio, habilitados para resolver ou solucionar problemas. O endereço da página é:

http://www.orengonline.com/fortran95/

Também estão disponíveis as respostas e comentários a respeito dos exercícios do livro, no item *exercícios*. E, para verificar se está pronto para resolver problemas, um conjunto deles está disponível no item *problemas*.

METODOLOGIA ...

Nos exemplos de programas as instruções da linguagem Fortran serão destacadas em negrito, e os elementos que não fazem parte da parte da sintaxe da linguagem serão escritos sem negrito. Mas, no texto, para chamar atenção serão também destacados em negrito.

Gilberto Orengo
Santa Maria, 2007

CAPÍTULO 1

CONSIDERAÇÕES INICIAIS: Apresentando o Fortran

Neste capítulo você encontrará:

1.1 INTRODUÇÃO

Seja bem-vindo ao *universo Fortran*.

Neste início de conversa é importante salientar que este texto foi integralmente escrito com o LaTeX [1]–[5]. O LaTeX é um sistema de processamento de textos utilizado especialmente para produzir documentos científicos de alta qualidade tipográfica. É também útil para outros tipos de documentos, desde simples cartas até livros completos. O LaTeX utiliza o TeX como base para seu processamento e a sua distribuição é gratuita, disponível para a maioria dos sistemas operacionais como o UNIX, Linux, Windows, OSMac, Sun e VMS. Nas Universidades é encontrado, previamente instalado, nas redes UNIX/Linux de computadores (exceto redes Windows). É aconselhável, se você ainda não teve contato com LaTeX que o faça. É uma ferramenta computacional muito poderosa para uso no meio acadêmico e científico. No Apêndice E são apresentadas algumas linhas de programação LaTeX, que foram utilizadas na elaboração deste livro.

O LIVRO

É muito importante para compreensão na leitura do livro observar as seguintes convenções adotadas.

NOTAÇÃO DO LIVRO*

As seguintes notações ou convenções são usadas ao longo do livro:

fortran 95 — na linha que aparece este símbolo, como nota de margem, será destacado na mesma cor a instrução referente ao Fortran 95, que não havia no FORTRAN 77.

fundo cinza claro conterá um exemplo de programa Fortran 95. A numeração dos programas seguirá sequencialmente na ordem arábica, antecedida do número do capítulo.

fundo cinza médio conterá a sintaxe de instruções ou procedimentos do Fortran 95.

AaBb — escritas com o tipo `typewriter` negrito identificarão elementos referentes ao Fortran 95.

AaBb — escritas com o tipo **helvética** negrito identificarão exemplos referentes ao LaTeX, disponíveis no Apêndice E.

[ ] — os colchetes *sempre* indicarão que o elemento Fortran 95 é opcional, por exemplo, uma declaração, uma instrução ou um atributo.

[Nx] — em sobrescrito identifica uma nota de rodapé, em que x será o número da nota.

* — notas como esta, em fonte helvética, permitirão inserções informativas, ou explicativas ou ainda de advertências a respeito da programaçao em linguagem Fortran.

O conteúdo sobre Fortran [6]–[10] é extenso para ser tratado num livro que pretende ser introdutório. Sendo assim, veremos os conceitos mínimos necessários para uma iniciação na linguagem Fortran e, na sequência introduziremos alguns conceitos mais modernos do Fortran 90/95, o qual chamaremos somente de Fortran 95. Entre as evoluções sofridas pelo Fortran (relativas ao FORTRAN 77) daremos ênfase a três: as funções intrínsecas fortran 95 `SELECTED_REAL_KIND` e `SELECTED_INT_KIND` (Cap. 7, na pág. 107), que permitem maior portabilidade entre computadores e compiladores Fortran; a declaração de variáveis fortran 95 `ALLOCATABLE` (Cap. 5, na pág. 91), que habilita a alocação dinâmica de memória e; as Declarações e Procedimentos do tipo `MODULE` (no Cap. 8, na pág. 111) que, entre outras funções, substitui com primazia os confusos e perigosos **`COMMON`**, que são utilizados como uma declaração para compartilhar uma mesma área de memória entre duas ou mais unidades de programa, e para especificar os nomes das variáveis que devem ocupar esta área comum. A área de memória suportada pelo **`COMMON`** pode ser entendida como uma seqüência de posições na memória principal. Este assunto será apresentado no Capítulo 8.

Sutilmente ao longo do texto serão apresentados outros pequenos avanços da linguagem, sempre destacados pela convenção acima descrita.

As atualizações do Fortran 95 como o uso de ponteiros (`POINTER`) ou, os tipos de dados definidos pelo usuário (`TYPE`), entre outras, serão tratadas em outra oportunidade. Elas farão parte da sequência deste livro, que tratará de questões mais avançadas relativas ao Fortran 95. Abordagens mais completas a respeito do Fortran podem ser encontradas nos livros citados nas referências bibliográficas. Mas é importante ter em mente que ao usarem um **compilador**[N1] Fortran, uma valiosa fonte de informação encontra-se no Guia (ou Manual) do Usuário (*User's Guide*) e no Manual de Referência da Linguagem (*Language Reference Manual*) do referido compilador.

Será utilizado, como referência, o compilador Fortran G95 desenvolvido, sob a licença GNU[N2], pela comunidade de Software Livre e pode ser obtido gratuitamente no site *http://www.g95.org*, para diferentes plataformas e sistemas operacionais. No Apêndice **??** é apresentado, detalhamente, como obter, instalar e utilizar o compilador Fortran G95, tanto para o sistema operacional Windows como para o Linux e, uma lista de compiladores Fortran é descrita no Apêndice **??**.

As aplicações da linguagem ficarão restridas as áreas da Física, da Matemática e das Engenharias, embora o Fortran possa ser aplicado a outras áreas como por exemplo Economia, Administração, Informática e Biologia.

Este livro poderá ser usado por estudantes que não conhecem a linguagem Fortran, bem como por aqueles que já dominam a linguagem, especialmente se forem oriundos do antigo e bom FORTRAN 77. Para os iniciantes é aconselhável que leiam (estudem) os capítulos na sequência em que se encontram. Os conhecedores da linguagem podem saltear entre os capítulos, conforme a necessidade, mas é interessante que leiam este capítulo, para entenderem algumas peculiaridades da escrita do livro.

Um glossário, disponível no Apêndice C (pág. 123), conterá explicações e definições rápidas sobre conceitos ou elementos do Fortran 95. Assim, contribuirá para uma visão mais global a respeito da linguagem.

No Apêndice **??**, na página **??**, traz um resumo dos capítulos, cujo objetivo é o de um *Guia de Referência*, de rápido e fácil acesso. A disposição do resumo dos capítulos em um único local, é por acreditar que esta metodologia contribua mais para o aprendizado e seja mais didática do que o resumo no final de cada capítulo.

[N1] O termo *compilador* se refere, no jargão da informática, a um *software* que faz a *tradução de um programa fonte codificado em um código legível para a máquina*. Estudaremos o compilador na página 10.
[N2] Informações a respeito da licença GNU podem ser obtidas em *http://www.fsf.org/*.

UM BREVE HISTÓRICO SOBRE O FORTRAN

Para muitos, nos anos 80, o Fortran estava "morto" e "sepultado", porque a sintaxe da linguagem foi e, é até os dias atuais, considerada arcaica por muitos programadores que aprenderam linguagens mais modernas.

O Fortran é uma linguagem de computador para programação técnica e científica, a qual foi criada sob medida para execução eficiente em uma vasta variedade de processadores. A primeira padronização ocorreu 1966, e desde então foram realizadas três revisões (1978, 1991 e 1997). A revisão de 1991 foi a maior das três. Uma quarta revisão foi implementada em acordo com o Encontro da ISO/IEC JTC1/SC22/WG5 em 1997, que considerou todas as sugestões e pedidos dos usuários da linguagem, culminando no Fortran 2003.

Os avanços significativos da revisão de 1991 foram a alocação dinâmica de memória, tipos derivados de dados, manipulação de arranjos (p.ex., matrizes e vetores), módulos, ponteiros, tipos de dados parametrizados e estruturas. Mas o principal impulso dado na linguagem foi em relação as características do HPF (*High Performance Fortran*). A seguir veremos uma breve retrospectiva e os avanços propostos pelo Fortran 2003.

A linguagem Fortran é a *matriz* de todas as linguagens científicas de computadores. No começo tinha somente a intenção de traduzir equações científicas para códigos de computadores e, foi neste sentido um programa verdadeiramente revolucionário. Antes do Fortran todos os programas de computadores eram lentos e originavam muitos erros. Na primeira versão, o FORTRAN I, lançada entre 1954 e 1957, a linguagem era muito pequena em comparação com as versões mais modernas. Entre outras coisas, continha somente as declarações de variáveis para os tipo inteiro (**`INTEGER`**) e real (**`REAL`**) e também não havia sub-rotinas. Durante este período percebeu-se diversos erros, que forçaram a IBM lançar o FORTRAN II em 1958.

Um avanço ocorreu com o lançamento da versão FORTRAN IV, em 1962, que permitiu os cientistas utilizarem pelos próximos 15 anos. Com isso, em 1966 o Fortran adotou uma padronização, a *American National Standards Institute – "Instituto Nacional Americano de Padronização"*(ANSI) e passou a ser chamado de FORTRAN 66.

A partir desta versão, todas as atualizações seguiram um padrão e o outro avanço veio com o FORTRAN 77, em 1977. Entre as novas características estão as que permitiram escrever e armazenar mais facilmente programas estruturados. Nesta versão foram introduzidas novas estruturas, como o bloco de decisão lógica **`IF...THEN`** e foi a primeira versão que habilitou o manuseio "amigável" de variáveis tipo caracteres (**`CHARACTER`**) ou *strings*.

A atualização mais importante foi a que deu origem ao Fortran 90. Esta versão inclui todo o FORTRAN 77 que serve de base, mas com mudanças significativas, tais como: i) a implementação da alocação dinâmica de memória para os arranjos (ou, em inglês, *arrays*– vetores e matrizes); ii) mudanças nas operações com *arrays*; iii) a parametrização das funções intrínsecas, permitindo assim utilizar mais do que dois tipos de precisão (simples e dupla) para variáveis do tipo real e complexa; iv) capacidade de escrever procedimentos internos e recursivos, como também chamar os procedimentos através de argumentos, sendo estes, opcionais ou obrigatórios; v) a implementação do conceito de ponteiros; vi) melhor portabilidade entre compiladores e processadores; vii) o uso de características especiais do hardware, tais como: *cache* de instruções, *pipeline*[N3] da CPU, disposições de programação paralela (ve-

[N3]*pipeline*: arquitetura de computador em *pipeline* (encadeada); 1. CPU–Unidade Central de Processamento que

torização), entre outras.

Estes novos conceitos da linguagem fizeram do Fortran 90 uma linguagem mais eficiente, especialmente para a nova geração de supercomputadores, e que nos permite dizer: a linguagem Fortran continuará a ser usada com sucesso por muito tempo. Embora o Fortran não seja muito utilizado fora dos campos da Ciência da Computação e da análise numérica, permanece como uma linguagem que desempenhará tarefas da área da computação numérica de alto rendimento. Neste sentido, o Fortran é até hoje superior em relação aos seus competidores, em especial em aplicações científicas computacionalmente intensivas como Física, Matemática, Meteorologia, Astronomia, Geofísica e Engenharias, porque permite a criação de programas que primam pela velocidade de execução.

O padrão mais recente, o Fortran 2003, sofreu a maior revisão, introduzindo várias novas caracteríticas. Esta atualização contempla, entre outros, os seguintes avanços:

- *acréscimos de tipos de dados definidos pelo usuário*: tipos de dados parametrizados, aperfeiçoamento no controle de acessibilidade e de estrutura. Por exemplo, definir o **`KIND`** somente na utilização efetiva do tipo definido e não antes;
- *suporte a programação Orientada a Objetos*: herança e extensões, poliformismo, alocação dinâmica de tipos de dados e, finalizações, cujo objetivo é "limpar" (desalocar) a memória de ponteiros quando deixam de existir.
- *manipulação otimizada de dados*: entre outras, a alocação de componentes; o atributo **`VOLATILE`**, o qual indicará que o dado objeto poderá sofrer alteração ao longo da execução, mesmo por outro programa que esteja rodando em paralelo; transferência de alocações de memória, e acréscimos nos procedimentos intrínsecos.
- *avanços nas Entradas/Saídas*: operações de transferência definidas pelo usuário para dados definidos pelo usuário; controle nas conversões de formatos; constantes nominativas para unidade lógicas pré-conectadas, acesso as mensagens de erro; instrução **`FLUSH`**, que permitará, entre outras coias, ler via teclado caracter por caracter, e também tornará disponível o dado para outro processo. Entrada/Saída assincrôna, que permitirá qie outras instruções podem ser executadas enquanto uma instrução de Entrada/Saída (leitura/escrita) está em execução.
- *procedimento ponteiros*;
- *suporte para as exceções do padrão para pontos flutuantes (IEEE 1989)*;
- *mais interoperabilidade com a linguagem C*: permitindo, por exemplo, acesso a bibliotecas de baixo nível da linguagem C e vice-versa, ou seja, bibliotecas escritas em Fortran podem ser acessadas pela linguagem C.
- *suporte ao conjunto de caracteres internacionais*: acesso aos caracteres ISO 10646 (2000) 4-byte, que é suficiente para cobrir todas as linguagens escritas do mundo. E, a escolha

é construída em blocos e executa instruções em passos em que cada bloco trata de uma parte da instrução e dessa forma acelera a execução do programa; 2. (a) escalar a chegada das entradas no microprocessador quando nada mais está acontecendo e desta forma aumentando a velocidade aparente de processamento; (b) iniciar o processamento de uma segunda instrução enquanto a atual ainda está processando de forma a aumentar a velocidade de execução de um programa. *Fonte: DIC Michaelis UOL, distribuição gratuita em CDROM.*

pelo usuário por ponto ou vírgula para dados numéricos formatados de Entradas/Saídas. Uma nova função intrínseca, para caracteres, a qual retorna o tipo de conjunto de caracteres: **`SELECTED_CHAR_KIND(NAME)`**;

- *integração com sistemas operacionais*: que permitirá acessar os argumentos na linha de comando e a variáveis de ambiente do interpretador, bem como as mensagens de erros do processador.

O compilador G95, que será utilizado nos exemplos deste livro, já contém as características implantadas no Fortran 2003. Para um conhecimento mais abrangente a respeito desta atualização do Fortran aconselha-se a referência [11], da qual foram extraídas as informações descritas acima.

No Fortran 2008 será revisado o Fortran 2003. Como ocorreu com o Fortran 95 em relação ao Fortran 90, será uma pequena evolução, somente incluindo correções e esclarecimentos do Fortran 2003. Mas será nesta versão a implementação mais efetiva na manipulação de processamento paralelo e os tipos de dados BIT.

A linguagem Fortran é definida de acordo com o Padrão Internacional ISO/IEC 1539. Atualmente, os padrões que conduzem as implementações do Fortran são:

- **ANSI X3.198-1992 (R1997)**. Título: *Programming Language "Fortran" Extended*. É informalmente conhecida como Fortran 90. O padrão é publicado pela ANSI.

- **ISO/IEC 1539-1:1997**. Título: *Information technology - Programming languages - Fortran - Part 1: Base language*. É informalmente conhecido como Fortran 95. Existem mais duas partes deste padrão. A Parte 1 foi formalmente adotada pelo ANSI.

- **ISO/IEC 1539-2:2000**. Título: *Part 2: Varying length strings*. Descreve um conjunto de procedimentos adicionais que fornecem formas de *strings* conterem comprimentos variáveis.

- **ISO/IEC 1539-3:1998**. Título: *Part 3: Conditional compilation*. Descreve um conjunto de recursos adicionais que podem ser utilizados em opções de compilação.

Estas informações, bem como notícias a respeito das atualizações, podem ser visualizadas no endereço: *http://www.nag.co.uk/sc22wg5/IS1539_family.html*.

Está disponível no endereço *http://www.orengonline.com/fortran95/* uma transcrição de uma reportagem do *New York Times*, de 13 de junho de 2001, a respeito dos criadores do Fortran. O título é *Pioneers of the 'Fortran' Programming Language*.

A seguir estudaremos termos cujos significados serão importantes em todo o livro. Veremos também os conceitos relacionados a estrutura de um programa Fortran, tais como os tipos e a declaração de dados. E, criaremos passo-a-passo um programa Fortran, decrevendo e explanando cada etapa.

1.2
OS BITS E BYTES

Para uma aprendizagem mais significativa é necessário e importante conhecermos alguns termos utilizados nesta área da computação. Um ponto forte no uso de computadores é a memória. As memórias dos computadores são compostas de milhões de "interruptores eletrônicos" individuais, cada um assumindo ON ou OFF ("ligado" ou "desligado"), nunca num estado intermediário. Cada um dos "interruptores" representa um **dígito binário** (também conhecido como **bit** – de *binary digit*), em que o estado ON é interpretado como o binário 1 e o estado OFF como o binário 0. Diversos bits agrupados representam o sistema binário de números ou simplesmente *sistema de base dois*, representado pelo conjunto {0,1}. Para comparação, a base do sistema decimal é 10 (0 a 9).

O número de arranjos possíveis para números binários é fornecido por 2^n, em que n é o número de opções possíveis. O menor agrupamento de bits é chamado de **Byte**. Um Byte consiste de um grupo de 8 bits e é a unidade fundamental usada para medir a capacidade da memória de um computador. A partir desta unidade fundamental temos:

1024 Bytes = 1 KByte (1 KiloByte ou 1 KB), devido a base dois temos: $2^{10} = 1024$.

1024 KBytes = 1 MByte (1 MegaByte ou 1 MB = $2^{20} = 1048576$ Bytes).

1024 MBytes = 1 GByte (1 GigaByte ou 1 GB = 2^{30} Bytes).

1024 Gbytes = 1 TByte (1 TeraByte ou 1 TB = 2^{40} Bytes).

Para compreendermos um pouco mais o sistema binário busquemos algo familiar, o nosso sistema decimal, ou sistema de base 10, representado pelo conjunto {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Representamos um número nesta base (por exemplo, o 152) da seguinte forma:

$$152 \equiv 152_{10} = (1 \times 10^2) + (5 \times 10^1) + (2 \times 10^0)\,,$$

O sub-índice 10 é usado para indicar a base numérica. E como representamos um número na base numérica do computador ou base 2[N4]? Bem, fazemos da mesma forma que na base 10. Vejamos o exemplo do número 101:

PRONÚNCIA DOS BINÁRIOS

A pronúncia dos números na base 2 não é igual ao da base 10, isto é, o número 101 é pronunciado UM ZERO UM, enquanto que na base 10 é pronunciado CENTO E UM

$$101 \equiv 101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 5_{10}\,,$$

o qual representa no sistema decimal ao número 5. Observe que os três dígitos binários podem representar oito valores possíveis: do $0_{10}(= 000_2)$, $1_{10}(= 001_2), \ldots,$ até $7_{10}(= 111_2)$. Em geral, como mencionado anteriormente, se n bits são agrupados juntos para formar um número binário, então eles podem representar 2^n valores possíveis. Assim, um grupo de 8 bits (1 Byte) pode representar 256 valores possíveis. Numa implementação típica, metade

[N4]Como curiosidade, se existissem extraterrestres com oito dedos, como você esperaria que fosse a base representativa dos números? Claro, pensando como Ser Humano!!!

destes valores são reservados para representar números negativos e a outra metade para os positivos. No caso 1 Byte (8 bits) é utilizado usualmente para representar números entre -128 e $+127$, inclusive.

Um sistema para representar os caracteres (de linguagens Não-Orientais) deve incluir os seguintes símbolos:

PERMITIDOS NO FORTRAN:

- As 26 letras maiúsculas:
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- As 26 letras minúsculas:
  a b c d e f g h i j k l m n o p q r s t u v w x y z
- Os dez dígitos:
  0 1 2 3 4 5 6 7 8 9
- Símbolos comuns ou caracteres especiais:
  <espaço> ' ( ) , ? ! . : _ ; $ % & "
  - entre eles operadores aritméticos: = + − / *
  - e lógicos: < > == >= <= /=

***NÃO* PERMITIDOS NO FORTRAN:**

- Algumas letras especiais ou símbolos, tais como:
  @ # [ ] { } ~ ^ à ç ë £ entre outros.

O número total de caracteres e símbolos requeridos é menor do que 256, mas mesmo assim é utilizado 1 Byte de memória para armazenar cada caracter. Embora incompleto, este é o sistema de código ASCII (*American Standard Code for Information Interchange*), usado na maioria dos computadores. Atualmente, está no processo de implementação um outro sistema de código mais geral, chamado *Unicode*, que contemplará também as linguagens orientais. Na versão Fortran 2003, este sistema de códigos já está disponível.

Todas as máquinas tem um "tamanho de palavra"(***wordsize***) – uma unidade fundamental de armazenamento, por exemplo, 8 bits, 16 bits, etc. Esta unidade difere entre as máquinas, em um processador Pentium©, por exemplo, poderá ser de 32 bits (4 Bytes). Isto será importante mais adiante, na pág. 22.

Outro conceito interessante é o **Flop**, que é uma operação de ponto flutuante por segundo. Uma operação de ponto flutuante ocorre quando dois números reais são adicionados. Hoje, se fala de **MegaFlops** ou até mesmo em **GigaFlops**.

Para finalizar, um comentário a respeito de processamento paralelo (ou vetorização). O processamento paralelo ocorre quando duas ou mais CPUs[N5] trabalham **simultaneamente** na solução de um mesmo problema. Assim, obtém-se maior velocidade e volume de processamento computacional. Para fazermos uso deste procedimento é necessário que o compilador Fortran 95 nos habilite tal procedimento e que o programa seja projetado com este objetivo, i.e., implementando as declarações intrínsecas para o processamento paralelo.

[N5] **CPU:** Central Processor Unit ou, no bom português, *Unidade Central de Processamento*.

Como exemplo de compiladores comerciais que habilitam a vetorização temos, entre outros, o *Lahey-Fujitsu Fortran 95 for GNU/Linux* da Lahey Computer Systems, Inc. [7] e o *PGHPF* da Portland Group [8], este por sinal é um excelente compilador. O compilador G95 também permite o processamento paralelo.

1.3 COMO TRABALHA A MEMÓRIA DO COMPUTADOR?

Para entendermos como funciona a memória do computador, usaremos um exemplo hipotético, cujo tamanho de palavra é de 8-bits:

A memória dos computadores é endereçável, i.e., para cada alocação de memória é dado um número específico, o qual é freqüentemente representado em hexadecimal (base 16), por exemplo, **`3F2C`**. Mas, porque usar base 16? Vejamos sucintamente o motivo no quadro abaixo.

SISTEMA HEXADECIMAL

Os computadores trabalham no sistema dos números binários, mas nós "simples mortais" pensamos no "mundo" do sistema de números decimais. Felizmente, podemos programar os computadores para aceitarem os nossos números decimais, convertendo-os internamente para os binários da máquina. Mas, quando os cientistas, técnicos e engenheiros trabalham diretamente com o sistema binário percebem que o mesmo é difícil de manipulá-los. Vejamos, por exemplo, o número 1100_{10} no sistema decimal, que é escrito 010001001100_2 no sistema binário. Para evitar difícil manipulação no sistema binário, uma alternativa é quebrar o número binário em grupos de 3 e 4 bits e com isso obter novas bases, respectivamente, base 8 (**series octal**) e base 16 (**series hexadecimal**). Vejamos última representação, a base 16 ou hexadecimal. Um grupo de 4 bits pode representar qualquer número entre $0(= 0000_2)$ e $15(= 1111_2)$ (lembram do 2^n?). Então, um número hexadecimal tem 16 dígitos: $0, 1, \ldots, 8, 9$ mais de A, B, ..., E, F. Assim, $9_{16} = 9_{10}$; $A_{16} = 10_{10}$; $B_{16} = 11_{10}$; e assim por diante. Podemos quebrar um número binário em grupos de 4 e substituir os dígitos hexadecimais apropriados para cada grupo. Vejamos o nosso número $1100_{10} = 010001001100_2$. Quebrando-o em grupos de 4, temos: $0100|0100|1100_2$. Substituindo cada grupo pelo apropriado hexadecimal, obtemos $44C_{16}$, que representa o mesmo padrão de bits do número binário, mas de maneira simplificada

A CPU está habilitada a ler e escrever numa específica localização (área) de memória. Grupos de áreas de memória são tratados como "informações inteiras" (*não números inteiros*) possibilitando assim armazenar mais informações. Usar a identificação criptográfica hexadecimal para localização de memória é incomum (porque é mais complicado!!), assim o Fortran 95 possibilita substituí-las por nomes (em inglês).

Quando os computadores são ligados, cada localização de memória conterá algum tipo de "valor", e neste caso os valores serão aleatórios (randômicos). Em geral, os valores serão os que permanecem na memória do uso anterior. Por esta razão, é muito importante **inicializar as localizações de memória** antes de começar qualquer manipulação da mesma como cálculos, declaração de variáveis, etc.

Todas as CPU tem um **conjunto de instruções** (ou linguagem própria da máquina) para sua manipulação. De maneira geral, todos os programas Fortran 95 são convertidos (ou compilados) para o conjunto de instruções (ou linguagem de máquina). Grosseiramente falando, todos os processadores têm o mesmos tipos de instruções. Assim, a CPU pode *dizer coisas* como, "busque o conteúdo da área de memória **`3F2C`**" ou "escreva este valor na localização (área) de memória **`3F2A`**". Esta é basicamente a maneira de como os programas trabalham.

Considere a seguinte seqüencia de instruções em **código *assembler***:

`LDA '3F2C'`	$\Rightarrow$	carregue (ou busque) os conteúdos de **`3F2C`**
`ADD '3F29'`	$\Rightarrow$	adicione estes conteúdos em **`3F29`**
`STO '3F2A'`	$\Rightarrow$	armazene o valor resultante na localização **`3F2A`**

Esta seqüencia de instruções, que tem significado somente ilustrativo para os nossos propósitos, efetivamente adiciona dois números e armazena o resultado numa área de memória diferente. Até 1954, quando o primeiro dialeto da linguagem Fortran foi desenvolvido, todos os programas de computador eram escritos usando o código assembler. Foi John Backus e sua equipe, então trabalhando na IBM, que propôs e desenvolveu um método econômico e eficiente de programar. A idéia foi de projetar uma linguagem que possibilitasse expressar fórmulas matemáticas de uma maneira mais natural do que na época era feito somente com a linguagem assembler. Do resultado de suas primeiras tentativas surgiu o FORTRAN (forma abreviada para *IBM Mathematical **FOR**mula **TRAN**slation System*).

Esta nova linguagem possibilitou que as instruções acima fossem escritas de maneira menos criptografada, como por exemplo:

$$K = I + J\,.$$

A seguir veremos a diferença entre compiladores, interpretadores e depuradores.

Compilador

Compilador é um programa que, a partir de um código-fonte escrito em uma linguagem de programação (de alto nível), cria um programa semanticamente equivalente porém escrito em outra linguagem, conhecido por código-objeto ou linguagem de máquina (ou ainda, instruções de código *assembler*), o qual chamaremos simplesmente de executável.

As principais numa compilação, a partir de um código-fonte, são:

1. **Análise léxica:** processo que analisa a entrada de linhas de caracteres de um código-fonte, caractere a caractere, e produz uma seqüência de símbolos chamado *símbolos léxicos*, ou simplesmente "símbolos" (*tokens*), que podem ser manipulados mais facilmente, na segunda etapa, por um *parser*, conhecido como leitor de saída.

2. **Análise sintática (*parsing*):** processo que analisa uma sequência de entrada para determinar sua estrutura gramatical segundo uma determinada gramática formal. Nesta etapa é identificada, por exemplo, que o **`WRTE`** está incorreto. O correto é **`WRITE`**, que é uma instrução de escrita.

3. **Análise semântica:** é a fase da compilação em que verifica se as estruturas do programa farão sentido durante a sua execução, ou seja detecta os erros semânticos. Por exemplo, verifica a multiplicação entre tipos de dados diferentes, tal como número inteiro multiplicado por um número real. Nesta etapa é preparada, por meio de coleta as informações, a próxima fase da compilação que é a fase de síntese, por pelo menos mais três etapas.

4. **Gerador de código intermediário:** como o próprio nome traduz, é um passo intermediário, que em alguns casos poderá ser o código-objeto final.

5. **Otimizador de código:** nesta etapa o código intermediário é otimizado em termos de velocidade de execução e espaço de memória.

6. **Gerador do Código-objeto:** nesta fase é gerado o código-objeto, otimizado ou não, dependendo das opções de compilação. Os objetivos desta etapa são, por exemplo, reserva de memória para as variáveis e de dispositivos de entrada e saída de dados. Então, é gerado o executável, em linguagem de máquina, para uma determinada arquitetura de computador, isto é, dependente de máquina.

Para auxiliar na compreensão de algumas funções de um compilador, usaremos uma analogia com a linguagem portuguesa. Seja a seguinte afirmação:

Ela comprar duas febre, e também comprará um cordeiro, ambos amaña.

Inicialmente, numa análise léxica, verifica-se que o caracter *ñ* não pertence ao conjunto de símbolos da língua portuguesa. Para corrigir, os dois caracteres *ña* serão substituídos por *nhã*. Na análise sintática é verificado se a estrutura está em acordo com as regras gramaticais. Neste caso, será detectado dois erros, que serão corrigidos por:

$$\begin{aligned} \textit{comprar} &\Longrightarrow \textit{comprará}, \\ \textit{febre} &\Longrightarrow \textit{febres}. \end{aligned}$$

Na análise semântica é verificado se a frase faz sentido. E, neste exemplo não apresenta significado, porque febre não é objeto venal. Provavelmente houve um engano, e a afirmação correta seja

Ela comprará duas lebres, e também comprará um cordeiro, ambos amanhã.

E por fim, uma otimização pode ser realizada, se o autor desejar, porque a frase está correta e compreensível. Se for o caso teremos, por exemplo:

Ela comprará duas lebres e um cordeiro amanhã.

No Fortran: o compilador é evocado, assim como para outros tipos de compiladores, por uma palavra chave, que depende de compilador para compilador. No nosso caso é o `g95`, do Fortran G95. Ao compilar é anexado ao código, entre outras, instruções matemáticas, instruções de dispositivos de entrada e saída de dados, por exemplo, via teclado e via monitor, respectivamente. Os arquivos executáveis gerados são específicos para cada processador e/ou sistema operacional, isto é, código compilado num computador com processador Intel Pentium ou AMD não será executado numa Estação Sun SPARC e vice-versa. Da mesma forma, um código compilado num computador com sistema operacional Windows não será executado num computador com sistema operacional Linux, UNIX ou MacOS e vice-versa. Assim, quando trocarmos de plataforma (processador e/ou sistema operacional), devemos compilar novamente o código. Observe que um termo novo, **código** ou **código computacional**, foi introduzido e se refere a um programa de computador gerado com uma linguagem de programação.

No Apêndice **??**, na pág. **??**, é apresentado o compilador G95, utilizado como referência neste livro. É indicado como baixá-lo na internet e instalá-lo, tanto para a plataforma Windows como para Linux. Exemplos de uso do compilador estão descritos a partir da página 16.

Existe outra forma de executar um programa de computador gerado por uma linguagem de programação. São os interpretadores. E para corrigir erros em programas podemos recorrer aos depuradores, os quais veremos resumidamente a seguir.

Interpretador

Os interpretadores são programas que lêem um código-fonte de uma linguagem de programação e o executam. Seu funcionamento pode variar de acordo com a implementação e, em muitos casos o interpretador lê linha-a-linha e converte em código-objeto à medida que vai executando o programa. Linguagens interpretadas são mais dinâmicas e apresentam a seguinte sequência:

$$\textit{escrever código} \Longrightarrow \textit{testar} \Longrightarrow \textit{corrigir} \Longrightarrow \textit{escrever} \Longrightarrow \textit{testar} \Longrightarrow \textit{distribuir}.$$

Já os compiladores tem uma sequência dada por:

$$\textit{escrever código} \Longrightarrow \boldsymbol{\textit{compilar}} \Longrightarrow \textit{testar} \Longrightarrow \textit{corrigir} \Longrightarrow \boldsymbol{\textit{compilar}} \Longrightarrow \textit{testar} \Longrightarrow \textit{distribuir}.$$

Mas existem também linguagens que funcionam como interpretadores e compiladores, como por exemplo: Python (somente quando requerido), BASIC, entre outras. Outros exemplos de linguagens interpretadas são: Bash, C#, Perl, PHP, Python, Euphoria, Forth, JavaScript, Logo, entre outras.

Depurador

Depurador, também conhecido por *debugger*, é um programa usado para testar outros programas e fazer sua depuração, ou seja, indica seus problemas ou *bugs*[N6].

Em geral, os depuradores oferecem as seguintes funcionalidades:

[N6] *Bug* significa em inglês qualquer tipo de inseto. Os primeiros computadores de tamanhos de grandes salas paravam os processamentos devido a insetos que se localizavam em seus dispositivos. Para retornar ao trabalho era preciso retirar os insetos, ou seja, fazer um *debugging*. Assim, este termo foi mantido no meio acadêmico e até hoje é utilizado para *erro* e *procura de erros*.

1. a execução passo-a-passo de um programa, chamada *single-stepping*;
2. a suspensão do programa para examinar seu estado atual, em pontos predefinidos, conhecidos como *breakpoints*, ou pontos de parada; e
3. o acompanhamento do valor de variáveis que também o programador pode definir, por meio das *watch expressions*, que podem ser usadas inclusive para gerar uma suspensão, ou ativar um *breakpoint*: por exemplo, em um programa que calcula a raiz quadrada, uma dessas condições seria se a variável que contém o argumento passasse a ser negativa.

Na seção 1.8, pág. 38, trataremos a respeito de erros em Fortran 95. Veremos a seguir como criar e executar o primeiro programa em Fortran. Mas, antes precisamos dos conceitos de unidades de programas. Para obter mais informações a respeito do processo de compilação, aconselha-se o livro descrito na ref. [12].

1.4 AS UNIDADES DE PROGRAMA

Unidades de programa são os menores elementos de um programa Fortran que podem ser compilados separadamente. Existem cinco tipos de unidades de programas:

- Programa Principal (ou *Main Program*)
- Sub-Programa **FUNCTION** (são as funções definidas pelo usuário)
- Sub-Programa **SUBROUTINE** (são as sub-rotinas)
- Unidade de Programa **BLOCK DATA**
- Unidade de Programa **MODULE**

A seguir veremos o primeiro tipo. Os subprogramas **SUBROUTINE** e **FUNCTION** estão descritos no Capítulo 6, na página 97. A unidade de programa **MODULE** é apresentada no Capítulo 8, na página 111. A unidade de programa **BLOCK DATA** não será abordada neste livro. Um **BLOCK DATA** fornece valores iniciais para dados compartilhados por uma ou mais unidades de programas. Uma leitura complementar sobre estas unidades de programa são encontradas nas referências indicadas ou no Manual do Usuário do compilador.

Programa Principal (*Main Program*)

A execução de um programa principal inicia com a primeira declaração ou instrução executável no programa principal e finaliza com a instrução **END** do programa principal ou com uma instrução **STOP**, localizada em qualquer lugar do programa. De fato o que determina a finalização do programa é o **END**. Quando é encontrado um **STOP** ele remete o fluxo de execução incondicionalmente para o **END**, e assim a execução do programa é abortada.

A forma estrutural de um programa principal em Fortran 95 é:

```
[PROGRAM nome_do_programa]
[USE nome_do_use]
[IMPLICIT NONE]
[declaração global dos dados]
   instruções executáveis e não-executáveis
[CONTAINS]
[subprogramas internos]
END [PROGRAM nome_do_programa]
```

em que, os colchetes indicam que o elemento Fortran é opcional[N7] e, a ordem obrigatória está indicada na seqüencia exposta. É uma **boa prática de programação** iniciar o código com a instrução **PROGRAM** seguida do referido nome, tornando claro o início do programa principal. Cada um dos itens da estrutura serão tratados no decorrer do livro.

Um código computacional tem um e somente um programa principal. É a estrutura de referência, isto é, o compilador reconhece somente um **END** – que é obrigatório, ou quando existir, inicialmente, somente um **PROGRAM** – que é opcional. Para as demais unidades de programa não há número mínimo ou máximo de aparições.

Antes de criarmos os dois primeiros programas, é importante e necessário vermos alguns elementos do Fortran 95.

O FORMATO LIVRE

No Fortran 95 podemos escrever em qualquer coluna (posição) na linha e temos 132 posições para ocupar. Este é chamado de *formato livre*. No FORTRAN 77, versão anterior ao Fortran 90/95, se inicia o código a partir da coluna 7 até 72. As colunas 1 até 6 são utilizadas para instruções de controle de fluxo do código e para localização de rótulos, como os utilizados na instrução **FORMAT**. A coluna 6, especificamemte, indica, pela escrita de um caracter qualquer, a continuação da linha anterior. Da coluna 73 até 80 é o campo de identificação do cartão. Esta estrutura, chamada de *formato fixo*, era devido ao cartão de "*digitação*", em que cada linha tinha 80 colunas (posições) para ser perfurado, para posterior manipulação de leitura e processamento pelo computador. O Fortran 95 também suporta o formato fixo.

O CONJUNTO DE CARACTERES

O conjunto de caracteres do Fortran consiste, como visto na página 8, de:

- As 26 letras maiúsculas:
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- As 26 letras minúsculas:
  a b c d e f g h i j k l m n o p q r s t u v w x y z
- Os dez dígitos:
  0 1 2 3 4 5 6 7 8 9

[N7] Lembrando a convenção: neste livro, os colchetes *sempre* indicarão que o elemento (por exemplo, uma declaração, uma instrução ou um atributo) é opcional.

- Símbolos comuns ou caracteres especiais:
  <espaço>[N8] = + − / " < > ' ( ) , ? ! . : _ ; $ % & *

Em Fortram as letras maiúsculas são equivalentes as correspondentes minúsculas, exceto quando estiverem relacionadas com as declarações do tipo **CHARACTER**, que será estudado na seção 1.5.4, na página 24. Isto é, para o Fortran os nomes **TESTE**, **teste**, **Teste** e **TeStE** são equivalentes – *não há distinção entre maiúsculas e minúsculas*, ou seja, não é tipo "*CASE SENSITIVE*".

O caracter ponto-e-vírgula (;) pode ser usado para separar duas ou mais instruções numa mesma linha, como por exemplo:

```
READ(*,*) x; WRITE(*,*) 'Ola amigo ...'
```

cujo significado é, respectivamente, leitura via teclado de um valor para armazenar na variável **x** e escrita da frase **Ola amigo...**, que está entre as aspas. Ambas instruções serão estudadas no Capítulo 2 e *variável* é um elemento que armazenará algum valor e será estudada na página 25. Outros exemplos:

```
a = 23 ; b = 150 ; c = 23.76
```

que significa, atribuição de valores às variáveis **a**, **b** e **c**. A atribuição de valores às variáveis será estudada na página 27.

OS NOMES SIMBÓLICOS EM FORTRAN

Os nomes simbólicos são utilizados em Fortran para se referir a vários elementos como nomes de programa e de variáveis. Um nome inicia, necessariamente, com uma letra e terá no máximo até 31 caracteres, os quais poderão ser letras, dígitos e *underscore* – "traço-baixo" (_). É proibído o uso de caracteres especiais, de caracteres acentuados ou cedilhados, tais como: <espaço> = + − / " < > ' ( ) , ? ! . : ; $ % & * @ # [ ] { } ~ ^ à ç ë £. São chamados de simbólicos, como mencionado anteriormente (pág. 10), porque substituem aquelas instruções de máquina de localização de elementos na memória.

Exemplos de nomes válidos em Fortran:

t_2006	**t2006**	**primeiro_programa**
Metodo_LTSN	**AaBb**	**Calculo_integral_definida**
soma	**SOMA**	**Soma**

Exemplos de nomes **inválidos** em Fortran:

2_t_2006	**primeiro programa**
calculo_do_π	**ilegal_@_caracter**
ação_distância	**nome_muito_longo_nao_pode_ser_usado**

Dê nomes as variáveis e as unidades de programa que representem, mais próximo possível, o que elas processarão. Por exemplo, se uma variável armazenará o valor de pi, então é conveniente chamá-la de **pi**. Outro, se uma expressão calculará o salário de um funcionário, após o seu aumento, é interessante chamá-la de **salario_final**.

[N8] <espaço> significa um espaço em branco.

OS RÓTULOS EM FORTRAN

Os procedimentos ou instruções em Fortran podem ter suas posições identificadas no programa. Essa identificação é conhecida como *rótulo* e é descrito por um (1) até cinco (5) algarismos, sendo o primeiro algarismo não nulo. É localizado sempre a esquerda na linha, antes da instrução ou procedimento. Os rótulos são importantes, por exemplo, na instrução não executável **`FORMAT`**, que será estudada na seção 2.6 do Capítulo 2, na página 56.

Exemplos de rótulos válidos em Fortran:

```
200          1234          20000          10          00010
```

Os dois últimos rótulos são equivalentes. Um mesmo rótulo não poderá identificar duas ou mais linhas na mesma unidade de programa. O rótulo (e por consequência instruções ou procedimentos) poderá ser invocado (chamado) quantas vezes for necessário.

O PRIMEIRO PROGRAMA

Veremos um exemplo simples de programa escrito na linguagem Fortran 95: *O primeiro programa*. Como é de praxe em qualquer livro de linguagem de programação, o primeiro programa apresentado sempre ensina como escrever uma mensagem na tela do monitor. Isto tem um significado importante: *todos os procedimentos realizados por um programa (ou código computacional), como por exemplo cálculos de expressões matemáticas, precisam ser apresentados para o usuário e isso só é possível por intermédio de uma saída de informação. Caso contrário, todas as informações manipuladas ficarão "aprisionadas" na memória do computador sem que saibamos seus valores, e se perderão ao desligar o computador.*

Como mencionando anteriormente, no Apêndice **??** é apresentado com detalhes, como obter, instalar e utilizar o compilador Fortran G95, tanto para o sistema operacional Windows como para o Linux. No Apêndice **??** é apresentada uma lista de outros compiladores Fortran.

Para criar um programa é necessário seguir os três seguintes passos[N9]:

1. ***Digite o código fonte em um editor de texto e salve-o com a extensão*** **`.f90`.**

   O termo código fonte será utilizado para o arquivo contendo as instruções Fortran 95 em modo texto.

[N9] Existem compiladores Fortran que apresentam uma área de trabalho totalmente integrada, isto é, é possível num mesmo ambiente digitar o código fonte, compilá-lo para gerar o executável e executá-lo. É o caso, por exemplo, dos compiladores Lahey, PowerStation e Portland.

O EDITOR DE TEXTO

Para elaboração do programa Fortran é possível utilizar qualquer editor, com exceção daqueles que ao salvar o arquivo anexem instruções de controle do próprio editor. Assim, ao compilar o programa contido neste arquivo poderá ocorrer um erro de compilação devido a estas instruções (muitas vezes caracteres) de controle. Assim, evite o WORD e o WORDPAD, no caso do Windows. Utilize o NOTEPAD (o mesmo que BLOCO DE NOTAS) para digitar os programas, e salve-o com a extensão "`.f90`" ou "`.f95`". Importante: habilite no "Salvar como" a opção "Todos". Assim será evitado o indesejável acréscimo da extensão "`.txt`" após a extensão "`.f90`" ou "`.f95`". No Linux, em princípio, pode-se utilizar qualquer um dos editores de texto disponíveis.

2. ***Gere o programa executável usando o comando `g95` do compilador G95.***

   Importante:
   Antes de seguirmos, uma advertência: será dado preferência por instruções utilizadas no GNU/Linux[13]. Para usuários do Windows, a transferência de instruções é quase automática para uso no terminal DOS. Por exemplo, para acionar o compilador G95, já instalado, é utilizado o comando **`g95`** tanto para o sistema operacional GNU/Linux[N10] como para o Windows, isto é, no terminal do DOS ou do GNU/Linux[N11] digitamos esta palavra chave seguida do nome do programa em Fortran 95.

   Assim, para gerar um executável, digite no terminal DOS ou num terminal Linux:

   ```
   g95 nome_programa.f90
   ```

   Este comando:

   1°) verifica, no programa (código fonte), as sintaxes das instruções Fortran,

   2°) na sequência, gera um código objeto com nome **`nome_programa.o`**,

   3°) e repassa o código objeto para o *"linkador"*[N12], que anexa bibliotecas (sistema, E/S (Entrada/Saída de dados), etc.) e gera um executável, com um nome *default* chamado **`a.out`** no Linux e **`a.exe`** no Windows.

   Para alterar o nome do arquivo executável é possível utilizar uma opção do compilador, a **`-o`** seguida do nome que se deseja para o executável. Assim, temos para o exemplo acima:

   ```
   g95 -o nome_de_saida nome_programa.f90
   ```

[N10] Para exemplificar o uso de outro compilador, por exemplo, o da *Lahey/Fujitsu for GNU/Linux*, compila-se com o comando **`lf95`**.

[N11] **Conselho:** se você não trabalha com o sistema operacional GNU/Linux, experimente !!! É gratuito e não por isso ineficiente, pelo contrário, é altamente estável e confiável.

[N12] Infelizmente, na falta de uma palavra apropriada em Português, para a ação de quem faz um *link (ligação, vínculo, elo)*, que reforce a idéia em computação, estou usando *"linkador"*.

O nome de saída, no caso do Linux, pode ser acrescido de uma extensão ou não. Para o Windows é interessante acrescentar a extensão "EXE" (mas não necessário), para evitar algum problema referente a estrutura de arquivo.

Para outras opções do compilador, consulte o manual do Usuário. Somente para o Linux, digite **`man g95`** para ler o manual.

No 3° item acima, o compilador anexa ao código, entre outras coisas, cálculos matemáticos, entrada de dados, e saída de resultados. Por exemplo, as entradas podem ser via teclado, arquivo (pág. 50) ou outro dispositivo. As saídas podem ser via monitor, impressora, arquivo ou por outro dispositivo. É importante relembrar que os arquivos executáveis são específicos para cada processador e/ou sistema operacional. Ou seja, um código compilado num computador, com G95 por exemplo, num processador Intel Pentium não será executado numa Estação Sun SPARC, e vice-versa. Da mesma forma, um código compilado num computador com sistema operacional Windows não será executado num computador com sistema operacional Linux e vice-versa. Assim, quando trocarmos de plataforma (processador e/ou sistema operacional), devemos compilar novamente o código.

3. ***Execute o código (programa) usando o comando:***

No Linux:	**`./nome_programa`** ou **`./a.out`**
No Windows:	**`nome_programa.exe`** ou **`a.exe`**
ou simplesmente:	**`nome_programa`** ou **`a`**

Agora que sabemos a sequência da criação de um programa Fortran, segue abaixo um programa que imprime uma mensagem na tela do monitor.

Programa 1.1 – O primeiro programa.

```
WRITE(*,*) 'Ola mundo externo ....'
END
```

Digite-o num editor de texto preferido, salve-o com um nome desejado, compile-o, conforme descrito na página 17 e após execute-o. Este é o mais simples dos programas em Fortran. Contém apenas duas linhas e utiliza a instrução **`WRITE(*,*)`** para transferir para a tela do monitor a frase **`Ola mundo externo ....`**, que está contida entre as aspas simples. As aspas simples podem ser substituídas por duplas aspas, por exemplo, para contemplar a seguinte saída, que contém uma aspa simples no seu texto:

```
WRITE(*,*) "A queda d'agua eh bonita."
END
```

Observe, novamente, que todo programa Fortran 95 termina com a instrução **`END`**.

O SEGUNDO PROGRAMA

Neste exemplo, o programa 1.2, já são apresentadas boas práticas de programação que serão descritas na sequência. O código transforma o valor do ângulo em graus para radianos, e

imprime o valor do cosseno do ângulo. Todo o procedimento foi executado em precisão simples, cujo tema será tratado na página 23. A numeração a esquerda não faz parte do código e a sua utilização é para melhor explicar cada linha do código (ou programa). Esta será uma prática daqui para frente. Lembre-se que ainda terás contato com todos os elementos Fortran, portanto, se não compreender alguma descrição ela será tratada ao longo do livro.

Programa 1.2 – Exemplo de programa que converte ângulos em graus para radianos.

```
1  PROGRAM graus_para_rad
2  IMPLICIT NONE
3  !
4  ! Este programa converte angulos em graus para radianos
5  !
6  REAL :: theta, ang_rad
7  REAL, PARAMETER :: pi=3.14159265
8     WRITE(*,*)"Indique um angulo em graus:  "
9     READ(*,*)theta
10    ang_rad = theta*pi/180.0  ! Aqui ocorre a conversao
11    WRITE(*,*) 'O angulo ',theta,', em graus, vale',&
12       ang_rad,' radianos'
13    WRITE(*,*) 'cos(theta) = ',COS(ang_rad)
14 END PROGRAM graus_para_rad
```

Procurou-se neste exemplo deixar clara a estrutura de um programa Fortran 95 e observe que as instruções da linguagem estão em negrito. Embora seja opcional, um programa Fortran 95 inicia com a instrução **PROGRAM** seguida do nome do programa. Na sequência constam as declarações de variáveis, entre as linhas 2 e 7. O corpo do programa, que contém as instruções excutáveis (e também não executáveis), está entre as linhas 8 e 13. E, finaliza com a instrução **END** seguida do nome do programa.

A seguir veremos uma descrição de cada linha, indicando o que executa ou instrui o computador a realizar. Aproveitaremos para indicar as boas práticas de programação e algumas exigências da linguagem Fortran.

Linha 1: Nesta linha ocorre o início do código ou programa em Fortran. A palavra-chave é **PROGRAM** seguido do nome do programa. Como visto anteriormente, na página 15, devemos ter cuidado ao nomear um programa. O nome de um programa Fortran 95 terá até 31 caracteres e iniciará sempre com uma letra do alfabeto, poderá conter letras, algarismos e o caracter "traço baixo" ou *underscore* (_).

Reiterando, embora seja opcional, é uma **boa prática de programação** colocar a instrução **PROGRAM** (sempre na primeira linha) seguida de um nome.

Linha 2: O **IMPLICIT NONE**, que já estava disponível nas últimas versões do FORTRAN 77, obriga-nos a declarar todas a variáveis do problema, auxiliando a depurar eventuais erros de escrita ou de dupla declaração. Embora seja opcional, também é uma **boa prática de programação** colocar a instrução **IMPLICIT NONE**. fortran 95

Linha 3: O caracter ! (sinal de exclamação) instrui o compilador que a sua direita o con- fortran 95

teúdo seja ignorado, ou seja, a linha é apenas um *comentário* no programa. É uma **boa prática de programação** escrever comentários a respeito de certas atitudes e linhas do programa. Assim, comente o máximo possível o seu programa. Neste caso foi utilizado simplesmente para deixar uma linha em branco, embora no Fortran é possível deixar em branco quantas linhas desejar, sem a necessidade do sinal de exclamação.

Linha 4: Nesta linha o comentário é para descrever a utilização do programa. O uso de comentários evita o esquecimento, por exemplo, do significado de cada variável ou do que trata o programa, fato que é comum com o passar do tempo. Ou ainda, possibilita que outros programadores entendam melhor o que foi programado. Escreva o objetivo do programa e das sub-rotinas. Insira também formas de contato com o programador, como por exemplo, o e-mail. Não esqueça, após o sinal `!` o conteúdo será ignorado.

Linha 5: Mais uma linha em branco "comentada".

Linha 6: Aparece a primeira declaração global de variável, que é do tipo real. Observe atentamente a sintaxe. Primeiro se escreve o tipo de dado (variável) seguido de dois dois-pontos e, na sequência, separados por vírgulas as variáveis. Os nomes da variáveis seguem os mesmos critérios que vimos para o nome do programa. É aconselhável nomear as variáveis com nomes mais próximo possível da sua função no programa. Por exemplo, **`theta`**, no jargão matemático está relacionado com ângulo, assim como o nome **`ang_rad`**, ângulo em radianos. As declarações de variáveis serão estudadas com mais detalhes na página 25.

Linha 7: Aparece a segunda declaração de variável, que também é do tipo real. Note que foi acrescentado um parâmetro na declaração: o **`PARAMETER`**, que instrui o compilador a **fixar** o valor da variável **`pi`** ao longo de todo o programa, não permitindo que seja alterado. Uma tentativa de alterar o valor de **`pi`** acarretará num erro de execução. A diferença para a declaração anterior é que as duas variáveis, **`theta`** e **`ang_rad`**, podem ser alteradas a qualquer momento no programa, enquanto **`pi`** permancerá **constante** durante toda a execução.

Linha 8: Nesta linha aparece a primeira instrução executável: o **`WRITE`**, que é uma instrução de saída de resultado (ou de escrita ou impressão). Neste caso, devido ao primeiro asterisco (*), a saída será na tela do monitor. Note que a mensagem está entre duplas aspas.

Linha 9: Agora uma instrução executável de leitura é apresentada: o **`READ`**, que receberá um valor para a variável **`theta`** digitada via teclado, também, devido ao primeiro asterisco. Mais informações a respeito de instruções de leitura e escrita estão no Capítulo 2, na pág. 49.

Linha 10: Nesta linha ocorre a conversão de graus para radianos, por intermédio de uma expressão matemática. Observe que o sinal de igual não indica uma igualdade propriamente dita, mas sim uma atribuição (que será estudado na página 27). Isto é, primeiro são realizadas todas as operações no lado direito da igualdade para posterior atribuição do resultado final à variável, neste caso a variável **`ang_rad`**. Note também que foi adicionado um comentário após uma instrução. *Esta é mais uma característica do Fortran 95*, isto é, ***permite que se faça comentário em qualquer parte do programa, sempre iniciando com um !***.

fortran 95

Linha 11: Nesta linha é impresso, pelo **WRITE**, o resultado das operações executáveis anteriores, e novamente na tela do monitor. Agora é uma múltipla saída, todas separadas por vírgula, que envolve valores armazenados nas variáveis (que será estudado na página 27), e de frases literais (entre aspas). Observe também o caracter **&**, que posicionado no final da linha indica que a linha continuará na seguinte. Se um nome, palavra-chave ou constante é quebrado por um **&**, o primeiro caracter da próxima linha deve ser outro **&**, seguido do restante do nome, palavra-chave ou constante. Veja o exemplo:


```
11    WRITE(*,*) 'O angulo ',theta,', em graus, &
12       &vale', ang_rad,' radianos'
```

O número máximo permitido de continuações de linhas é igual a 39. *Atenção:* o **&** não funciona no interior dos comentários (!) e não pode ser seguido de um comentário.

Linha 12: Aqui continua a linha anterior, indicado pelo **&** no final da linha 11. É opcional o uso de outro **&** no início da linha, a não ser nos casos citados acima.

Linha 13: É impresso o outro resultado, também na tela do monitor.

Linha 14: É a linha que encerra o programa. Caso não tivéssemos escrito a linha 1, poderíamos encerrar o programa somente com a instrução **END**. Como escrevemos a instrução **PROGRAM graus_para_rad**, devemos encerrar obrigatoriamente com a instrução **END** seguido de **PROGRAM graus_para_rad**.

Encerramos a breve análise do nosso segundo programa Fortran. Foi somente um exemplo, e assim deve ser encarado, porque existem outras tantas instruções, funções, declarações Fortran, que somente lendo este livro até o final terás uma boa noção antes de começar efetivamente a programar em Fortran. A medida que formos avançando outros programas serão propostos.

Em resumo, a estrutura geral de um programa Fortran é:

1. **BLOCO DA DECLARAÇÃO DAS VARIÁVEIS**, que é um conjunto de instruções não-executáveis que definem as variáveis que serão utilizadas ao longo do programa. Os sub-programas tem acesso as variáveis declaradas nesta etapa via troca por parâmetros.

2. **BLOCO DE EXECUTÁVEIS** - é o corpo do programa (contém entradas e saídas de dados inclusive), ou seja, é o código em si. Embora o bloco seja chamado de executável, poderá conter instruções não-executáveis.

3. **BLOCO DE ENCERRAMENTO DO PROGRAMA**, que são instruções que finalizam o programa e podem ser: **END [PROGRAM]** e **STOP**, este poderá aparecer em qualquer parte do programa e quantas vezes for necessária. Já o **END [PROGRAM]** é único.

A seguir, para avançarmos no aprendizado da linguagem Fortran, veremos como funciona a "despensa de dados" para o Fortran, especialmente como armazenar e manipular números inteiros e reais, na sequência os complexos.

1.5 A "DESPENSA DE DADOS" NO FORTRAN 95

O Fortran disponibiliza uma variedade de maneiras para armazenar e lidar com os dados. Os tipos de dados suportados pelo Fortran são:

- **INTEIROS:** números nos quais não há frações, ou que não possuem parte fracionária. São exatos.
- **REAIS:** números que são representados com uma parte fracionária ou que possuem ponto decimal. São conhecidos também como números representados em *ponto flutuante*. Aqui os números reais são tratados com ponto decimal e não com a vírgula, como estamos habituados no dia-a-dia.
- **COMPLEXOS:** números do plano complexo, que possuem uma parte imaginária (que é representada por um número real) e uma parte real (Na página 24).
- **LÓGICOS:** são dados que assumirão somente um valor: *verdadeiro* ou *falso* (Na pág. 24).
- **CARACTERES:** são dados constituídos por letras e símbolos (Na página 24).

Veremos com mais detalhes os inteiros e reais. Deixaremos para o estudante se aprofundar nos demais tipos de dados representativos no Fortran.

1.5.1 Dados Inteiros - `INTEGER` (Precisão Simples)

Os dados inteiros são armazenados "exatamente" na memória do computador e consistem de números inteiros positivos, inteiros negativos e zero. A quantidade de memória disponível para armazená-los dependerá de computador para computador, ou do *wordsize* do computador, e poderá ser de 1, 2, 4 ou 8 Bytes. O mais comum de ocorrer nos computadores atuais é 4 Bytes (32 bits).

Como um número finito de bits é usado para armazenar cada valor, somente inteiros que caiam dentro de um certo intervalo podem ser representados num computador. Normalmente, o menor número inteiro que pode ser armazenado em n-bits inteiros, chamada de ***precisão simples***, é:

$$\text{Menor Valor Inteiro} = -2^{n-1}$$

e o maior valor que pode ser armazenado em n-bits inteiros é:

$$\text{Maior Valor Inteiro} = 2^{n-1} - 1$$

Para o caso típico de 4 Bytes inteiros, temos para o menor valor e o maior valor possíveis, respectivamente, $-2.147.483.648$ e $+2.147.483.647$. Quando tentarmos usar valores abaixo ou acima destes ocorre um erro chamado de *overflow condition* e é indicado na execução do programa.

1.5.2 Dados Reais ou de Pontos Flutuantes - `REAL`

Os números reais são armazenados na forma de notação científica. Já sabemos que números muito grandes ou muito pequenos podem ser convenientemente escritos em notação científica. Por exemplo, a velocidade da luz no vácuo é aproximadamente 299.800.000 m/s. Este

número será mais "manuseável" se escrito em notação científica: 2,998 $\times 10^8$ m/s. As duas partes de um número expresso em notação científica são chamadas de **mantissa** e **expoente** da potência de dez. A mantissa é $2,998$ e o expoente é 8 (no sistema de base 10).

Na linguagem do computador os números reais são escritos de forma similar, a diferença se encontra no sistema usado porque o computador trabalha na base 2. Assim, se N-bits são dedicados para representar (e armazenar) um número real, parte é reservada para a mantissa e parte para o expoente. A mantissa caracteriza a **precisão** e o expoente caracteriza o **tamanho** que pode ser assumido pelo número. É nesta repartição, e também na quantidade, de bits que começa a diferenciação entre os computadores e compiladores Fortran.

■ PRECISÃO SIMPLES (Single Precision)

A grande parte dos computadores usam como precisão simples 4 Bytes (32 bits – o (***word-size***) do computador) para repartir entre a mantissa e o expoente. Normalmente esta divisão contempla 24 bits para a mantissa e 8 bits para o expoente. Assim, temos:

i) Mantissa (precisão) $\Rightarrow n =$ 24 bits (3 Bytes)

$\pm 2^{n-1} = \pm 2^{23} = 8.388.608 \Rightarrow$ que equivale a **7 algarismos significativos**,

ii) Expoente $\Rightarrow n' = 8$ bits (1 Byte)

$2^{n'} = 2^{8\,\text{bits}} = 2^{255_{10}}$, sendo metade para a parte positiva e metade para a negativa

Assim, o intervalo é dado por $2^{-128} \longleftrightarrow 2^{127}$, que resulta em $\mathbf{10^{-38}} \Longleftrightarrow \mathbf{10^{38}}$.

Isto quer dizer que um número escrito em precisão simples terá até 7 ou 8 algarismos significativos (dígitos decimais) e o seu expoente (da potência de 10) deve estar contido no intervalo entre -38 e 38. Excedendo a este intervalo acarretará no erro de *overflow*.

■ PRECISÃO DUPLA ou DUPLA PRECISÃO (Double Precision)

O Fortran 95 inclui uma possibilidade de representar números reais de forma mais ampla, do que a precisão simples - *default* nos computadores. Esta possibilidade é conhecida como **Dupla Precisão** (ou Double Precision). Usualmente a dupla precisão é de 8 Bytes (ou 64 bits), sendo 53 bits para a mantissa e 11 bits para o expoente. Assim, temos:

i) Mantissa (precisão) $\Rightarrow n = 53$ bits

$\pm 2^{n-1} = \pm 2^{52} \Rightarrow$ que equivale entre **15 e 16 algarismos significativos**,

ii) Expoente $\Rightarrow n' = 11$ bits

$2^{n'} = 2^{11\,\text{bits}} = 2^{2048_{10}}$, sendo metade para a parte positiva e outra para a negativa

Assim, o intervalo é dado por $2^{-1024} \longleftrightarrow 2^{1024}$, que resulta em $\mathbf{10^{-308}} \Longleftrightarrow \mathbf{10^{308}}$.

Desta forma, um número escrito em precisão dupla terá até 15 ou 16 algarismos significativos (dígitos decimais) e o seu expoente (da potência de 10) deve estar contido no intervalo entre -308 e 308. Excedendo a este intervalo acarretará no erro de *overflow*.

1.5.3 Os Números Complexos - COMPLEX

O estudo feito para os números reais é extensivo para os números complexos. A forma geral de um número complexo é $c = a + bi$, em que c é o número complexo, a (parte real) e b (parte imaginária) são ambos reais, e i é $\sqrt{-1}$. Em Fortran, os números complexos são representados por dois números reais constantes separados por vírgula e entre parênteses. O primeiro valor corresponde a parte real e o segundo a parte imaginária. Vejamos os seguintes casos em Fortran, cujo número complexo está ao lado:

`(1.,0.)`	$\Longrightarrow$	$1 + 0i$ (real puro)
`(0.7071,0.7071)`	$\Longrightarrow$	$0.7071 + 0.7071i$
`(1.01E6,0.5E2)`	$\Longrightarrow$	$1010000 + 50i$
`(0,-1)`	$\Longrightarrow$	$-i$ (imaginário puro)

Em que o **E** representa a base 10 e o número após é o expoente. Desta forma, o que vimos para os reais, é válido para os complexos. A diferença está no procedimento Fortran, que é feito pela declaração **COMPLEX**, que veremos adiante. A função **CMPLX**, que será estudada na página 32–tabela 1.2, converte um número inteiro ou real em complexo.

Programando em Fortran 95 deveremos ter cuidado ao declarar as precisões de nossas variáveis, já que tanto a definição de precisão simples como a de precisão dupla podem mudar de computador para computador[N13]. O Fortran 95 possibilita modificarmos a mantissa e o expoente, para escrevermos programas que possam ser facilmente portáveis entre processadores diferentes, com tamanho de palavra (*wordsize*) diferentes. Isto é obtido por uma função intrínseca que seleciona automaticamente a mínima precisão especificada, mesmo quando se troca de computador. Para os reais, esta função é a SELECTED_REAL_KIND, que veremos no Capítulo 7, na página 107. Para os inteiros temos a função SELECTED_INT_KIND.

fortran 95

1.5.4 Os outros dados: LOGICAL e CHARACTER

Além dos dados inteiros, reais e complexos, existem outros dois dados (ou tipos de variáveis para armazenar dados): os *lógicos* e os *caracteres*.

Os Lógicos (LOGICAL)

São dados que assumirão somente valor *verdadeiro* ou *falso* e, são especialmente úteis em tomadas de decisão, que serão estudadas no Capítulo 3, na página 63. Os valores são: **.TRUE.** (Verdadeiro) e **.FALSE.** (Falso).

Os Caracteres literais (CHARACTER)

São dados constituídos por letras e símbolos e são formados pelo conjunto de códigos da *American Standard Code for Information Interchange* (ASCII), que determina um padrão de caracteres, e está reproduzida parcialmente no Apêndice A.

[N13] Exemplos da dependência da combinação Processador/Compilador: num Supercomputador Cray T90/CF90[14]–[15], a precisão simples é 64 bits e a dupla 128 bits; já num PC/Lahey Fortran 95 e num PC/G95, a precisão simples é 32 bits e a dupla 64 bits.

1.6
A DECLARAÇÃO DAS VARIÁVEIS

Em Fortran, todas as variáveis que armazenarão os dados devem ser declaradas. Uma variável sempre terá um ***nome***, um ***tipo***, um ***tamanho*** e um ***valor***. São cinco os tipos de dados intrínsecos, três numéricos: **INTEGER**, **REAL**, **COMPLEX**, e dois não-numéricos: **LOGICAL** e **CHARACTER**. O tipo de declaração **DOUBLE PRECISION**, disponível no FORTRAN 77, é ainda suportado pelo Fortran 95, mas é considerado um subconjunto (ou um tipo – *kind*) do **REAL**.

Os tipos de dados foram tratados anteriormente e assumiremos que uma variável armazenará algum tipo de dado. Então estas duas palavras, dados e variáveis, serão sinônimos, no sentido que ambas estão relacionadas a valores que serão armazenados e manipulados por intermédio de variáveis.

A sintaxe geral de uma declaração de variável é:

```
<tipo> [([KIND=]<par_repres.>)][<atributos>][::] <variaveis>
```

em que:

<tipo>: é um dos tipos de dados estudados anteriormente e podem ser: **INTEGER**, **REAL**, **COMPLEX**, **LOGICAL** e **CHARACTER**, respectivamente representando, inteiros, reais, complexos, lógicos e caracteres.

([KIND=]<par_repres.>): em Fortran 95, cada um dos cinco tipos intrínsecos anteriores possui um valor inteiro não negativo denominado *parâmetro de representação* do tipo de dado. Este parâmetro é o valor correspondente em bytes disponibilizados para sua representação, como vimos em detalhes para os inteiros e reais, a partir da página 23.
Na normatização da linguagem Fortran 90/95 ficou estabelecido um padrão: qualquer processador deve suportar pelo menos dois parâmetros de representação (o **KIND**) para os tipos de dados **REAL** e **COMPLEX** e pelo menos um parâmetro para os tipos de dados **INTEGER**, **CHARACTER** e **LOGICAL**.

A tabela B.1, no Apêndice B, contém todos os tipos de dados, seus parâmetros de representação e intervalos de armazenamento, baseado no compilador G95.

<atributos>: são informações adicionais a respeito do tipo de dado e podem ser, entre outros:

DIMENSION(<forma>): indica a forma de uma *array*, em que **(<forma>)** indica as dimensões separadas por vírgulas. São exemplos:
- *unidimensional*: **DIMENSION(7)**, é um vetor de 7 elementos;
- *bidimensional*: **DIMENSION(3,4)**, é uma matriz 3×4, com 12 elementos.

Este argumento será estudado no Capítulo 4.

PARAMETER: indica que o dado será constante ao longo de todo o programa, sem possibilidades de alterá-lo,

ALLOCATABLE: é usado juntamente com o **DIMENSION** e indica que o tamanho da *array* será informado ao longo do programa e não no momento da declaração da variável. É a chamada *alocação dinâmica de memória* e será estudada no Capítulo 5.

Obs.:
– Podem ser declarados mais de um atributo para a mesma variável, os quais serão separados por vírgula.
– Ao longo do livro serão apresentados outros atributos para a declaração de variáveis.

`::` os uso dos dois "dois pontos" é opcional, mas será obrigatório caso deseja-se inicializar a variável no momento da sua declaração.

`<variáveis>`: são os nomes das variáveis que armazenarão os dados, separados por vírgulas. O nome de uma variável poderá ter até 31 caracteres e iniciará sempre com uma letra do alfabeto, jamais com um algarismo, que poderá ser utilizado a partir da segunda posição, assim como o *underscore* (_). Não conterá caracter especial (" , ( ){ }[ ]!˜ . : @ # $ % ^ & *) em qualquer posição do nome, bem como letras acentuadas ou cedilhadas.

A ATRIBUIÇÃO DE VALOR À VARIÁVEL

Uma variável é um registrador de posição de memória que pode conter qualquer dado, numérico e não-numérico, sendo possível alterar ou não durante a execução do programa.

No Fortran, uma variável recebe um valor por intermédio do sinal de igualdade. Esta operação recebe o nome de **atribuição** de um valor a uma variável, conforme o exemplo:

```
a = 15
```

Aqui a variável **a** recebeu o valor igual a 15, isto é, foi **atribuído** a ela (na memória) o valor inteiro 15. Esta interpretação é importante porque poderemos ter o seguinte caso:

```
b = 6
b = 5 + b   ! eh um acumulador
```

em que, inicialmente a variável **b** recebe o valor 6, e numa instrução posterior receberá um valor que dependerá do seu valor anterior. Ou seja, o novo valor será o anterior somado de 5. Esta é uma prática muito utilizada em programação e a variável **b** é conhecida por ***acumulador***. Pelo ponto de vista matemático teríamos um erro, pois o resultado final seria $0 = 5$, que é um absurdo. Mas do ponto de vista computacional temos que o primeiro passo é resolver a expressão do lado direito da igualdade, que é igual a 11, e após **atribuir** o resultado para a variável do lado esquerdo da igualdade, que o reterá na memória do computador.

A INICIALIZAÇÃO DE VARIÁVEIS

É importante, conforme descrito na página 10, atribuir um valor inicial a todas as variáveis do programa. Desta forma se evita erros indesejáveis de valores pre-atribuídos devido a "sujeiras" contidas na memória. A inicialização pode ser realizada já na declaração ou em qualquer parte do programa, como no exemplo a seguir.

```
INTEGER :: a = 15, c
REAL :: h = 2.350
    ⋮
  c = 260
```

VARIÁVEIS GLOBAIS e LOCAIS

Uma variável, quanto ao seu uso no código, poderá ser global ou local. Uma variável global é uma variável declarada em um procedimento **MODULE** e ativada pela instrução **USE**, enquanto que uma variável local é aquela declarada no programa principal (**PROGRAM**) e no interior de subprogramas, tais como funções (**FUNCTION**) e sub-rotinas (**SUBROUTINE**).

As variáveis globais podem ser utilizadas (ou acessadas) em qualquer parte do código, inclusive no interior dos subprogramas. Já as variáveis declaradas locais, no programa principal e no interior dos subprogramas, não podem ser acessadas fora dos mesmos e no caso dos subprogramas só "existirão" enquanto o referido subprograma estiver ativo no fluxo de execução do programa. O exemplo abaixo evidencia as diferenças entre elas.

Os procedimentos **MODULE** e as instruções **USE** serão estudadas no Capítulo 8, na página 111 e, os subprogramas (**FUNCTION** e **SUBROUTINE**) no Capítulo 6, na página 97. Mesmo assim é possível compreender como são usadas aqui.

Programa 1.3 – Exemplos de declarações globais e locais.

```
1  MODULE var
2  IMPLICIT NONE
3  INTEGER :: a=1, b=2, c    !-----  VARIAVEIS GLOBAIS
4  END MODULE var
5
6  PROGRAM teste_dec_global
7  USE var                   !-----  ativa as variaveis globais
8  IMPLICIT NONE
9  INTEGER :: res            !-----  VARIAVEl LOCAL
10   CALL soma(res)
11   WRITE(*,*)"O resultado eh: ", res
12   CALL mult(res)
13   WRITE(*,*)"O resultado eh: ", res
14 CONTAINS
15     SUBROUTINE soma(z)
16     IMPLICIT NONE
17     INTEGER :: z, d=4     !-----  VARIAVEIS LOCAIS
18       WRITE(*,*)"Informe o valor da variavel 'c' (inteira): "
19       READ(*,*)c
20       z = a + b + c + d
21     END SUBROUTINE soma
22 END PROGRAM teste_dec_global
23
24 SUBROUTINE mult(z)
25 USE var                   !-----  ativa as variaveis globais
26 IMPLICIT NONE
27 INTEGER :: z, d=4         !-----  VARIAVEIS LOCAIS
28   z = a * b * c * d
29 END SUBROUTINE mult
```

Ao executar o código, informando que a variável **c** receberá o valor igual a 10, teremos como saída:

```
O resultado eh: 17
O resultado eh: 80
```

É importante compreender o que exatamente o código executa, então vejamos atentamente o que está ocorrendo.

a) O procedimento **MODULE** `var` contém a declaração das variáveis, inteiras (**INTEGER**), **a**, **b** e **c**, em que as duas primeiras foram inicializadas.

b) no programa principal (**PROGRAM** `teste_dec_global`) as variáveis **a**, **b** e **c** são ativadas pela instrução **USE** `var`, e estarão disponíveis para todos os procedimentos do programa. A instrução **USE** é sempre localizada logo abaixo o início do programa ou do subprograma, e o objetivo é instruir o compilador a carregar os procedimentos do módulo.

c) na linha 10, o programa principal invoca (chama) um subprograma chamado de **SUBROUTINE** `soma`, cujo único parâmetro de passagem nesta conexão é a variável **res**, que receberá o resultado de uma soma realizada no referido subprograma. Note que na **SUBROUTINE** `soma` as variáveis **a**, **b** e **c** não foram declaradas e nem foi necessário a instrução **USE** `var`. A ausência deste último é porque o subprograma está contido no interior do programa principal, ação possibilitada pela instrução **CONTAINS**. É importante ressaltar que a instrução **CONTAINS** permite que subprogramas pertençam ao programa principal e, a sua localização é tal que os procedimentos abaixo sejam somente **SUBROUTINE**s e **FUNCTION**s.

d) na linha 24, outro subprograma (neste caso a **SUBROUTINE** `mult`) é utilizado para realizar uma multiplicação de variáveis. Observe que pelo fato de estar localizado externamente é necessária a instrução **USE** `var`, para tornar as variáveis **a**, **b** e **c** globais.

e) para encerrar, atente para a variável **c**. Ela foi preenchida por uma entrada de dados por meio de um **READ**, na linha 20, e automaticamente o valor atribuído é disponibilizado globalmente, como pode ser verificado pela operação matemática na linha 28.

Para melhor entendimento dos subprogramas recorra ao Capítulo 6, na página 97, mas abordaremos rapidamente os que utilizamos neste exemplo. Os subprogramas, são unidades de programas e são os menores elementos de um programa Fortran que podem ser compilados separadamente. Os do tipo **SUBROUTINE** são ativados pelas instruções **CALL**, que trocam parâmetros entre o programa principal e os subprogramas. Os parâmetros são "dados" trocados entre o programa principal e a **SUBROUTINE**, e estão localizados entre os parênteses, logo após as instruções **CALL** e **SUBROUTINE**. A ordem dos parâmetros é definida pelo usuário, e deve ser mantida rigorosamente nas chamadas. As variáveis não necessariamente tem o mesmo nome no programa principal e nos subprogramas. Observe que no programa principal a variável que receberá a saída do subprograma é **res** e no interior dos subprogramas é **z**.

Para finalizar, toda vez que um subprograma é ativado, o programa principal altera o seu fluxo de execução para o subprograma. Após a realização da tarefa do subprograma, o

fluxo retorna para a próxima linha após a chamada (**CALL**). Observe também que a variável **res** retém o último valor a ela atribuído e, que a variável **d** é do tipo local.

A seguir são apresentados exemplos de declarações de variáveis num programa Fortran:

Programa 1.4 – Exemplo com declarações de variáveis num programa Fortran.

```
1  INTEGER  a, b, c
2  INTEGER :: ai_doi, junho_2001
3  INTEGER :: dia = 1
4  INTEGER, PARAMETER :: mes = 5
5  REAL(KIND=8) :: oi
6  REAL, PARAMETER :: ola = 4.0
7  REAL, PARAMETER :: pi = 3.141593
8  REAL, DIMENSION(4) :: a1
9  REAL, DIMENSION(3,3) :: b1
10 DOUBLE PRECISION :: dupla
11 CHARACTER(LEN=10) :: primeiro, ultimo
12 CHARACTER(10) :: primeiro = 'Meu nome'
13 CHARACTER :: meio_escuro
14 LOGICAL :: claro
15 LOGICAL :: escuro = .FALSE.
16 COMPLEX :: nao
17 COMPLEX, DIMENSION(256) :: aqui
18  . . .
19 END
```

Inicialmente, como mencionado, os dois pontos (::) são facultativos quando não inicializamos a variáveis, caso da linha 1. Assim, seriam necessários somente nas linhas 3, 4, 6, 7, 12 e 15, mas é uma **boa prática de programação** colocá-los. Como vimos anteriormente, é sempre bom inicializarmos as variáveis, para evitar que venham carregadas de algum *lixo* da memória. fortran 95

As variáveis **a1** e **b1** são variáveis compostas ou simplesmente *arrays*, i.e., vetores ou matrizes, como demonstra o atributo **DIMENSION**. Estes tipos de variáveis serão estudadas no Capítulo 4, na página 79. No primeiro caso, é um vetor de tamanho 4 e, no segundo, uma matriz 3×3. Nestes dois exemplos, é informado ao processador que ele deve reservar na sua memória um espaço para armazenar as *arrays* **a1** e **b1**. Esta é uma alocação estática de memória, ou seja, do início até o fim da execução do programa este espaço de memória está reservado para este procedimento, mesmo que somente sejam usadas no início do programa. Mais adiante, veremos como alocar memória dinamicamente pelo comando ALLOCATABLE, que será estudada no Capítulo 5, na página 91. fortran 95

Outra consideração é com relação ao atributo **KIND**, na linha 5, que especifica o tipo de precisão desejada. Na ausência é assumido como precisão simples, que seria **(KIND=4)**, em que o algarismo 4 indica a precisão simples. Assim, para obtermos precisão dupla é necessário acrescentar o atributo **(KIND=8)**, em que o algarismo 8 indica a dupla precisão. Na declaração é permitido omitir a palavra-chave **KIND=**, podendo assim a declaração ser

escrita como segue

```
REAL(8) :: dupla
```

A declaração da linha 10, **DOUBLE PRECISION**, será considerada obsoleta em breve. É importante, então, substituí-la por

```
REAL(KIND=8) :: dupla
```

que tem o mesmo significado.

Na declaração da linha 11 o atributo **LEN=10** indica o tamanho máximo que o caracter pode assumir. Aqui também pode ser omitido a palavra-chave **LEN=** sem perda de significado, exemplificado na linha seguinte. Por curiosidade, a palavra-chave **LEN** vem da palavra em inglês *length* que significa comprimento.

1.7 OS PROCEDIMENTOS INTRÍNSECOS

Há tarefas que são executadas com frequência quando trabalhamos no computador, por exemplo, quando efetuamos cálculos podemos utilizar seguidamente o cosseno de um ângulo ou até mesmo o logaritmo ou a raiz quadrada de um número real. O Fortran oferece com muita eficiência um conjunto de procedimentos intrínsecos, que fazem parte do núcleo do compilador. O Fortran 95 tem mais de 130 procedimentos intrínsecos, divididos em diferentes classes:

- ***Elementares:*** são os procedimentos matemáticos, numéricos, de manipulação de caracteres e de bits;
- ***Consulta:*** que relatam ou retornam o estado de determinado procedimento, como por exemplo, se a alocação de memória ou a leitura de um dado foi um sucesso ou não;
- ***Transformação:*** procedimento que transforma de um estado para outro, como por exemplo, a conversão de um número real em inteiro ou vice-versa.
- ***Mistos:*** que incluem rotinas relativas ao processador e ao tempo, como por exemplo, **DATE_AND_TIME**.

Não nos preocuparemos em mostrar todos e nem em apresentar na classificação feita acima. Alguns desses procedimentos serão vistos a seguir, especialmente os relativos a funções matemáticas e de transformação numérica. Outros, referentes a caracteres, serão estudados na página 36.

1.7.1 As Funções Matemáticas Intrínsecas

Como ocorre numa simples calculadora de mão, o Fortran 95 oferece quase uma centena de funções pré-definidas ou conhecidas por *funções intrínsecas*, ou *procedimentos intrínsecos*, tais como cosseno e seno de um ângulo. Algumas funções estão apresentadas na tabela 1.1, nas quais são indicados os possíveis argumentos de cada uma das funções.

Tabela 1.1 – Alguns procedimentos intrínsecos (matemáticos) do Fortran 95.

Instrução	Argumento	Função
ACOS(x)	$\mathbb{R}$	arccos(x)
ASIN(x)	$\mathbb{R}$	arcseno(x)
ATAN(x)	$\mathbb{R}$	arctag(x) ou arctg(x)
ATAN2(y,x)	$\mathbb{R}$	arctag(y/x) ou arctg(y/x)
COS(x) †	$\mathbb{R}\,\mathbb{C}$	cos(x)
SIN(x) †	$\mathbb{R}\,\mathbb{C}$	seno(x)
TAN(x) †	$\mathbb{R}$	tag(x) ou tg(x)
EXP(x)	$\mathbb{R}$	e^x
LOG(x) ††	$\mathbb{R}\,\mathbb{C}$	ln(x)
LOG10(x) ††	$\mathbb{R}$	log(x)
ABS(x)	$\mathbb{I}\,\mathbb{R}\,\mathbb{C}$	|x| (módulo de x)
MOD(x,y)	$\mathbb{I}\,\mathbb{R}$	x/y (resto da divisão). É o resultado de **x** − **INT**(x/y)*y. Ex.: **MOD**(3,2) é 1 e **MOD**(2,−3) é 2. Para reais: **MOD**(4.5,1.5) é 0.0 e, **MOD**(5.0,1.5) é 0.5, ou seja, deu exato 3.0 e para completar o 5.0 é necessário 0.5.
SQRT(x) ‡	$\mathbb{R}\,\mathbb{C}$	$\sqrt{x}$
DIM(x,y)	$\mathbb{I}\,\mathbb{R}$	fornece a diferença positiva. Se X > Y, então **DIM**(x,y) = X−Y. Se Y > X e o resultado é negativo, então **DIM**(x,y) = 0. Ex.: **DIM**(5,3) é 2 e **DIM**(3,5) é 0. E **DIM**(4.5,2.5) é 2.0.
MAX(x1,x2,...)	$\mathbb{I}\,\mathbb{R}$	fornece o maior valor entre os argumentos. Deve ter no mínimo 2 argumentos. Ex.: **MAX**(2,5,0) é 5.
MIN(x1,x2,...)	$\mathbb{I}\,\mathbb{R}$	fornece o menor valor entre os argumentos. Deve ter no mínimo 2 argumentos. Ex.: **MIN**(2,5,0) é 0.

† argumento em radianos. †† argumento > zero. ‡ argumento ≥ 0.
Convenção para o Argumento: $\mathbb{I}$: Inteiro, $\mathbb{R}$: Real de simples ou dupla precisão, $\mathbb{R}_S$: Real de precisão simples, $\mathbb{R}_D$: Real de dupla precisão, $\mathbb{C}$: Complexo.

Na sequência, um código exemplifica o uso de funções intrínsecas do Fortran. Foram utilizadas duas funções, o **COS**, que fornece o cosseno do ângulo (dado em radianos) e, a função **ABS**, que retém o sinal de um número inteiro ou real, isto é, fornece o valor numérico sem o sinal, conhecido como *absoluto do número*.

Programa 1.5 – Exemplo de uso de funções intrínsecas do Fortran.

```
1  PROGRAM uso_funcoes
2  IMPLICIT NONE
3  ! Exemplo do uso de funcoes intrinsecas
4  ! Autor: Gilberto Orengo (e-mail: orengo@orengonline.com)
5  REAL :: pi,alfa,theta,f,fx,modulo
6     pi = 3.14159265
7     WRITE(*,*) "Digite o valor do angulo (em graus):"
8     READ(*,*)alfa
9     theta = (alfa*pi)/180.0 ! Converte p/radianos
10    WRITE(*,*)"Digite o valor da Forca (em Newtons):"
11    READ(*,*)f
```

```
12     fx = f*COS(theta)
13     modulo = ABS(fx)
14     WRITE(*,*)"A Forca no eixo x, em modulo, eh: ", modulo
15  END PROGRAM uso_funcoes
```

Há casos que será necessário utilizar as funções intrínsecas para obter outras. Por exemplo, no caso de logaritmos em outras bases usa-se a mudança de base pela relação:

$$\log_a(x) = \frac{\log(x)}{\log(a)} \qquad \overset{\text{no Fortran}}{\Longrightarrow} \qquad \texttt{LOG10(x)/LOG10(a)}$$

Há funções intrínsecas que determinam relações entre Inteiros, Reais e Complexos, descritas na tabela 1.2 , que também serão importantes no decorrer do livro.

Tabela 1.2 – Algumas funções intrínsecas para manipulação de inteiros e reais.

Instrução	Argumento	Função
REAL(x)	$\mathbb{I}\ \mathbb{R}_D\ \mathbb{C}$	Converte um número inteiro, real de dupla precisão e complexos em um número real de precisão simples. Ex.: **REAL**(7) resulta em 7.00000000.
DBLE(x)	$\mathbb{I}\ \mathbb{R}_S\ \mathbb{C}$	Converte um número inteiro, real de precisão simples e complexo em um número real de precisão dupla. Ex.: **DBLE**(7) resulta em 7.000000000000000.
INT(x)	$\mathbb{R}\ \mathbb{C}$	Converte um número real de precisão simples ou dupla e complexo em um número inteiro por meio de ***truncamento***. Ex.: **INT**(5.9) resulta em 5, e **INT**(−3.5) resulta em −3. **INT**(**CMPLX**(−2.4,1.7)) resulta em −2.
NINT(x)	$\mathbb{R}\ \mathbb{C}$	Converte um número real em um número inteiro por meio de ***arredondamento***; em que x é um real. Ex.: **NINT**(5.9) resulta em 6, e **INT**(−2.4) resulta em −2.
CMPLX(x,y)	$\mathbb{I}\ \mathbb{R}$	Converte um número real ou inteiro(s) em um número complexo; em que x é um real. Ex.: **CMPLX**(5) resulta em (5.0,0.0), e **CMPLX**(−2.4,3.7) resulta em (−2.4,3.7)
CEILING(x)	$\mathbb{R}$	fornece o menor inteiro maior ou igual ao real x. Ex.: **CEILING**(3.15) é 4 e **CEILING**(-3.15) é -3.
FLOOR(x)	$\mathbb{R}$	fornece o maior inteiro menor ou igual ao real x. Ex.: **FLOOR**(3.15) é 3 e **FLOOR**(-3.15) é -4.

Convenção para o Argumento: $\mathbb{I}$: Inteiro, $\mathbb{R}$: Real de simples ou dupla precisão, $\mathbb{R}_S$: Real de precisão simples, $\mathbb{R}_D$: Real de dupla precisão, $\mathbb{C}$: Complexo.

Uma listagem completa das funções pré-definidas está disponível no site *http://www.orengonline.com/fortran95/* ou no Manual de Referência da Linguagem. Para o compilador G95, consulte o endereço eletrônico *http://www.g95.org*.

1.7.2 A Aritmética com inteiros e reais

AS EXPRESSÕES ARITMÉTICAS

As expressões aritméticas são aquelas que apresentam como resultado um valor numérico

que pode ser um número inteiro ou real, dependendo dos operandos e operadores. Os operadores aritméticos estão descritos na tabela 1.3.

Tabela 1.3 – Os operadores aritméticos disponíveis no Fortran 95.

Operadores Aritméticos		
Operador binário	Função	Exemplos
*	multiplicação	5*2; A*B
/	divisão	$2/7$; $8.5/3.1$; M/N
+	soma ou adição	$3+5$; A + J
−	subtração	$5-1$; x − y
**	potenciação	3**5 $\Longrightarrow 3^5$
Operador unário	Função	Exemplos
+	indica número positivo	$+3$
−	indica número negativo	-1

É importante ressaltar alguns pontos na aritmética de inteiros e reais:

a) Dois operadores não podem estar lado-a-lado. Por exemplo: `2*-7` é ilegal. Na programação em Fortran é escrito como segue: `2*(-7)`. Da mesma forma devemos ter cuidado com a exponenciação. O correto é `2**(-3)` e não `2**-3`.

b) A multiplicação implícita não é aceita no Fortran, isto é, a expressão `z(x-y)` deve ser escrita como `z*(x-y)`.

c) A radiciação pode ser transformada numa potenciação, por exemplo, $\sqrt{6} \rightarrow (6)^{1/2}$, que na linguagem Fortran fica `6**(1./2.)`. Deve-se tomar muito cuidado nesta operação e este assunto é tratado com mais detalhes na página 35.

A ordem de prioridades nas operações aritméticas, na linguagem Fortran é:

1° Operações que estão no interior de parênteses, iniciando sempre dos parênteses mais internos para os mais externos. Os parênteses determinaram a ordem da operação. Então, use sem restrições os parênteses para evindeciar a ordem das operações.

2° Todas as potenciações (`**`), da direita para a esquerda, e depois as radiciações (`SQRT`).

3° Todas as multiplicações (`*`) e divisões (`/`), partindo da esquerda para a direita.

4° Todas as adições (`+`) e subtrações (−), realizando as operações da esquerda para a direita.

A ARITMÉTICA DOS INTEIROS

A operação entre números inteiros resulta em números inteiros. Desta forma deveremos ter cuidado com algumas operações, em especial com a divisão. Vejamos os casos abaixo.

$$\frac{1}{2}=0 \qquad \frac{2}{2}=1 \qquad \frac{3}{2}=1 \qquad \frac{4}{2}=2 \qquad \frac{5}{2}=2 \qquad \frac{6}{2}=3$$

$$\frac{7}{2}=3 \qquad \frac{8}{2}=4 \qquad \frac{9}{2}=4 \qquad \frac{5}{4}=1 \qquad \frac{7}{4}=1 \qquad \frac{1}{3}=0$$

Baseado nos resultados devemos ter cautela quando formos utilizar estas operações de divisão entre inteiros, porque, por exemplo na primeira divisão,

$$\frac{1}{2}=0 \neq\ 0.5\,,$$

sendo este último o resultado da divisão entre reais. Lembre-se que os números reais são tratados com ponto decimal e não com a vírgula, como estamos habituados.

Sugere-se que utilize os números inteiros nos procedimentos estritamente necessários, por exemplo, em contadores de repetições ou ainda em índices de variáveis compostas, como matrizes e vetores.

A ARITMÉTICA DOS REAIS

As mesmas operações realizadas anteriormente com inteiros, entre reais resultará em:

$$\frac{1.0}{2.0}=0.5 \qquad \frac{2.}{2.}=1. \qquad \frac{3.}{2.}=1.5 \qquad \frac{4.}{2.}=2. \qquad \frac{5.}{2.}=2.5 \qquad \frac{6.}{2.}=3.$$

$$\frac{7.}{2.}=3.5 \qquad \frac{8.}{2.}=4. \qquad \frac{9.}{2.}=4.5 \qquad \frac{5.}{4.}=1.25 \qquad \frac{7}{4}=1.75$$

$\frac{1.}{3.}=0.3333333$ (Precisão simples) $\qquad$ $\frac{1.}{3.}=0.333333333333333$ (Precisão dupla)

Observe que as duas últimas divisões se diferenciam pela a precisão. Um cuidado necessário é com as operações matemáticas que estamos acostumados a realizar no dia-a-dia, tais como

$$\texttt{3.0*(1.0/3.0)} \neq \texttt{1.0}$$

e,

$$\texttt{2.0*(1.0/2.0)} = \texttt{1.0}$$

Esta diferença está na forma como o computador manipula os dados, que dependem da precisão estabelecida. Assim, quando formos realizar testes de igualdades devemos ter muito cuidado.

A ARITMÉTICA MISTA: ENTRE INTEIROS E REAIS

Quando misturamos operações de reais com inteiros devemos ter cuidado extremo. Vejamos o porquê? Observe as seguintes operações.

1) `2 + 1/2` resulta em: `2`
2) `2. + 1/2` resulta em: `2.`
3) `2 + 1./2` resulta em: `2.5`
4) `2 + 1/2.` resulta em: `2.5`
5) `2. + 1./2.` resulta em: `2.5`

A primeira expressão é somente entre inteiros e não há novidade, porque o resultado será um inteiro. A última também não tem novidade, porque o resultado será um real, porque todos os termos são reais. A partir da segunda expressão, até a quarta, temos operações mistas. O resultado de uma operação mista, entre inteiros e reais, sempre será real. Vejamos como isso ocorre, na seguinte sequência, para a terceira expressão:

`2 + 1./2` $\Longrightarrow$ `2 + 1./2.` $\Longrightarrow$ `2 + .5` $\Longrightarrow$ `2. + .5` $\Longrightarrow$ `2.5`

Cada termo é resolvido separadamente. Primeiro, pela hierarquia, resolve-se a divisão. Como tem uma operação envolvendo um inteiro e um real, inicialmente o Fortran converte o inteiro para real. A seguir a operação de divisão é realizada. No próximo passo novamente há o envolvimento entre inteiro e real. Novamente, primeiro o Fortran converte o inteiro para real, para finalmente fornecer o resultado em real.

Outro cuidado é quanto ao envolvimento de operações mistas com exponenciação. Por exemplo, seja $6^{1/2}$, que na linguagem Fortran fica `6**(1./2.)`. Deve-se tomar muito cuidado nesta operação, porque numa potenciação o expoente é sempre inteiro. Assim, neste caso o expoente não pode ser nulo, já que $1/2 = 0$, na aritmética de inteiros. Então é preciso fazer uma divisão por reais. Quando o expoente é fracionário a operação é substituida por:

$$a^b = e^{b \ln a}\,, \text{em que } b \text{ é real} \quad \Rightarrow \quad 6^{1./2.} = e^{1./2.\,\ln(6.)}\,,$$

sendo que a base inteira foi convertida para real e passou como argumento do logaritmo natural. Então, o cuidado está na base que não poderá ser negativa, porque o logaritmo de número negativo não é definido.

Finalmente, um último cuidado embora não tem nada a ver com operações mistas é importante salientar. A potenciação sempre será realizada da direita para a esquerda, assim

$$(3^4)^2 = 3^8 = 6561\,,$$

é diferente de

$$3^{4^2} = 3^{16} = 43046721\,.$$

Os resultados anteriores não contém pontos decimais porque são valores inteiros.

1.7.3 A Manipulação de Caracteres

Um conjunto de caracteres agrupados (*string*) representa uma palavra. A posição que cada caracter (caráter) ocupa neste agrupamento determinará o significado da palavra. Não são aceitos caracteres acentuados (á, à, ã, ä, ...) e caracteres especiais (π, β, α, ...).

Os computadores somente "entendem" números, assim um código ASCII é a representação numérica dos caracteres tais como 'a' ou '@'. A tabela ASCII possui 256 posições e contém também caracteres acentuados, para contemplar a língua latina (Português, Espanhol, Italiano, etc).

Algumas funções para manipular caracteres são descritas a seguir.

LEN(x) $\Longrightarrow$ retorna o número (inteiro) de caracteres que compõem um conjunto de caracteres (palavra ou *string*), em que x é uma palavra escrita entre aspas (simples ou dupla).
Exemplo: **LEN**('mae') $\Rightarrow$ resultado = 3.
Exemplo de uso em um programa:

Programa 1.6 – O uso da função **LEN**.

```
1  PROGRAM uso_len
2  IMPLICIT NONE
3  CHARACTER(LEN=6) :: nome
4  INTEGER :: a
5     nome = 'fisica'
6     a = LEN(nome)
7     WRITE(*,*)"Tamanho da palavra = ", a
8  END PROGRAM uso_len
```

A saída na tela do monitor, será: **Tamanho da palavra = 6**

LEN_TRIM(x) $\Longrightarrow$ retorna o número (inteiro) de caracteres ***sem*** *os espaços em branco* que compõem um conjunto de caracteres (palavra ou *string*), em que x é uma palavra escrita entre aspas (simples ou dupla).
Exemplo: **LEN_TRIM**('mae e pai') $\Rightarrow$ resultado = 7.
Se fosse usada a função **LEN**('mae e pai') o resultado seria igual a 9.

ACHAR(i) $\Longrightarrow$ retorna o caracter da tabela ASCII correspondente ao inteiro i.
Exemplo: **ACHAR**(65) $\Rightarrow$ resulta na letra "A", e **ACHAR**(97) $\Rightarrow$ resulta na letra "a"
Exemplo do uso em um programa:

Programa 1.7 – O uso da função **ACHAR**.

```
1  PROGRAM uso_achar
2  CHARACTER :: letras
3  INTEGER :: posicao
4     WRITE(*,*)"Digite um inteiro para obter o&
5                &  caracter ASCII:"
6     WRITE(*,*)"(Nao esqueca que eh entre 0 e 256)"
7     READ(*,*) posicao
8     letras = ACHAR(posicao)
9     WRITE(*,*)"O caracter eh: ", letras
10 END PROGRAM uso_achar
```

Se for digitado 110, a saída na tela do monitor será: **O caracter eh: n**, ou se for digitado 56, a saída na tela do monitor será: **O caracter eh: 8**. Observe que neste último é retornado pela função **ACHAR(56)** o caracter 8 e não o algarismo 8. Desta forma não podemos operá-lo aritmeticamente.

IACHAR(x) $\Longrightarrow$ retorna um número inteiro, que indica a posição do caracter x na tabela ASCII, em que **x** deve estar entre aspas (simples ou duplas). Se esta entrada for requisitada por intermédio de uma leitura (pelo comando **READ**), não são usadas aspas, pois será armazenada em uma variável.
Exemplo: **IACHAR**('b') $\Rightarrow$ resulta no inteiro 98.
Exemplo em um programa:

Programa 1.8 – O uso da função **IACHAR**.

```
1  PROGRAM uso_iachar
2  CHARACTER :: letra
3  INTEGER :: posicao
4     WRITE(*,*)"Digite um caracter da tabela ASCII:"
5     READ(*,'(A)')letra
6     posicao = IACHAR(letra)
7     WRITE(*,*) "A posicao na tabela ASCII eh: ",&
8                  posicao
9  END PROGRAM uso_iachar
```

Se for digitado o caracter **s**, a saída na tela do monitor será: **A posicao na tabela ASCII eh: 115**. Este resultado poderá ser manipulado aritmeticamente.

CONSIDERAÇÕES IMPORTANTES:

- É possível retirar ou retornar parte de uma palavra (*string*). Por exemplo, seja a *string* **'pessoa'**. Podemos retirar somente o conjunto **'sso'**. Isto é possível somente com a utilização de variáveis que armazenem caracteres. Vejamos como isto funciona por intermédio do seguinte programa.

Programa 1.9 – Retorna parte de uma *string*.

```
1  PROGRAM parte_de_char
2  CHARACTER(LEN=20) :: total
3  CHARACTER(LEN=3) :: parte
4     WRITE(*,*)"Digite uma palavra, com ate &
5                & 20 posicoes:"
6     READ(*,*)total
7     parte = total(3:5)
8     WRITE(*,*) "A parte da palavra digitada eh: ", parte
9  END PROGRAM parte_de_char
```

Assim, se for digitado a palavra **'pessoa'**, a resposta será:

A parte da palavra digitada eh: sso

- Podemos comparar caracteres da tabela ASCII. A comparação será utilizando as operações lógicas de maior (>) e menor (<), por exemplo:

  'A' > 'a' $\Longrightarrow$ Falso (ou **.FALSE.**)

'a' > 'A' $\Longrightarrow$ Verdadeiro (ou **.TRUE.**)
'AAa' > 'AAb' $\Longrightarrow$ Falso (ou **.FALSE.**)
'i' > '9' $\Longrightarrow$ Verdadeiro (ou **.TRUE.**)
'*' > ')' $\Longrightarrow$ Verdadeiro (ou **.TRUE.**)

Isto é possível porque na tabela ASCII os caracteres são conhecidos pelas suas posições numéricas (inteiros). Assim, por exemplo, todas as letras maiúsculas vem antes das minúsculas, portanto as maiúsculas possuem um número de referência menor do que as minúsculas.

- O operador concatenação (**//**):
  É possível combinar dois ou mais conjunto de caracteres (*strings*) para formar um conjunto maior. Esta operação é feita pelo operador concatenação, a dupla barra para à esquerda, **//**. Vejamos o exemplo.

Programa 1.10 – Exemplo de concatenação de *strings*.

```
1  PROGRAM ex_conc
2  CHARACTER(LEN=15) :: total1
3  CHARACTER(LEN=7) :: total2
4  CHARACTER(LEN=9) :: parte1
5  CHARACTER(LEN=6) :: parte2
6     parte1 = 'Gilberto '
7     parte2 = 'Orengo'
8     total1 = parte1//parte2
9     total2 = parte1(1)//parte2
10    WRITE(*,*) "A primeira concatenacao eh: ", total1
11    WRITE(*,*) "A segunda concatenacao eh: ", total2
12 END PROGRAM ex_conc
```

Os resultados serão, respectivamente, **Gilberto Orengo** e **GOrengo**.

Desta forma, encerramos a apresentação dos elementos essenciais do Fortran 95. A partir deste ponto é possível criar programas simples, sem que necessitem de tomadas de decisão ou de processos de repetição, que veremos no Capítulo 3, na página 63. Encerrando este capítulo veremos como encontrar e depurar erros de programação.

1.8 CORRIGINDO ERROS – *DEBUGGING*

Ao criarmos um código estamos sujeitos a erros, os quais podem ser de três tipos: de sintaxe, de tempo-de-execução e de lógica. Infelizmente errar é também um atributo inerente a programação. O parágrafo abaixo, retirado do livro do Chapman [6], na página 70, ilustra com primazia esta situação.

> "... existem duas coisas certas na vida, a morte e os impostos. Mas se trabalhamos com programação acrescente mais uma certeza nesta lista: se escrever um programa com número de linhas significativos, ele apresenterá erro na primeira execução ..."

Historicamente os erros de programação são conhecidos como ***bugs***[N14], e o processo de localização dos erros como ***debugging***.

A maioria dos compiladores tem uma ferramenta anexa para depurar erros, que auxilia na detecção de problemas de execução do código. Nos compiladores que dispõem de interface gráfica esta tarefa é facilitada. Nos compiladores de linhas de comando, o depurador de erros é ativado por meio de opções de compilação. No Apêndice **??** é descrito o uso do depurador de erros no compilador G95.

A seguir veremos os três tipos de erros citados incialmente.

ERRO DE SINTAXE

O erro de sintaxe é o mais simples e o próprio compilador identifica-o, indicando como um erro de compilação. Este erro está relacionado com a escrita das instruções Fortran. São exemplos de erro de sintaxe:

Programa 1.11 – Exemplos de erros de sintaxes.

```
1 INTEGR :: b          ⟸
2    b = 2
3    WRITE(*;*) b      ⟸
4 END
```

Na primeira linha o erro está na escrita (sintaxe) da declaração de variável do tipo inteiro, que é escrita **INTEGER** e não **INTEGR**. Na terceira linha, o erro está na separação dos argumentos da instrução **WRITE** que são separados por vírgula e não por ponto e vírgula. O correto é:

Programa 1.12 – Correções dos erros de sintaxes.

```
1 INTEGER :: b
2    b = 2
3    WRITE(*,*) b
4 END
```

ERRO DE TEMPO-DE-EXECUÇÃO

É o tipo de erro que ocorre na execução do programa quando uma operação ilegal é tentada. Na maioria das vezes é uma operação matemática, por exemplo, uma divisão por zero. Ou ainda, a atribuição de um caracter no lugar de um inteiro ou real, sendo este também um erro de lógica. Quando este tipo de erro é detectado a execução é abortada.

[N14] *Bug* significa em inglês qualquer tipo de inseto. Os primeiros computadores de tamanhos de grandes salas paravam os processamentos devido a insetos que se localizavam em seus dispositivos. Para retornar ao trabalho era preciso retirar os insetos, ou seja, fazer um *debugging*. Assim, este termo foi mantido no meio acadêmico e até hoje é utilizado para *erro* e *procura de erros*.

ERRO DE LÓGICA

O erro de lógica é mais difícil de identificar, porque o programa é compilado e executado sem problemas, mas o resultado está errado. Neste caso o compilador não tem condições de reconhecer e está ligado diretamente à forma de programar. O compilador pode somente identificar um erro de lógica que está ligada a declaração de variáveis. Neste caso é também um erro de tempo-de-execução. Por exemplo:

Programa 1.13 – Exemplo de erro de lógica.

```
1 INTEGER :: b
2    READ(*,*) b
3    WRITE(*,*) b*2
4 END
```

Neste caso, a variável **b** é declarada como inteira e se for inserida, na linha 2, uma informação diferente de um valor inteiro, por exemplo real, acusará um erro e abortará a execução do programa.

Outros erros de lógica são difíceis de detectar, como por exemplo, a divisão por um número próximo de zero que pode levar o programa a gerar resultados errados. Ou ainda, se muitos arredondamentos são executados muitas vezes podem, também, levar a um erro de resultado. Para estes e outros tipos de erros de lógica o que podemos fazer é tomar algumas precauções, as quais são:

- Utilizar sempre a instrução **IMPLICIT NONE**, que obriga a declaração de todas as variáveis do código computacional;
- Nas expressões aritméticas e/ou lógicas usar parentêses para tornar claro a ordem de execução na expressão.
- Inicializar todas as variáveis. Assim se evita que alguma "sujeira de memória" contamine os cálculos ou as manipulações de dados.
- Escrever todas as entradas de dados. Desta forma é possível visualizar que dados foram informados ao programa e, assim possibilita sua confirmação e se for o caso, sua correção.

Uma prática utilizada para encontrar possíveis erros é permear o código com instruções tipo **WRITE(*,*)'estou aqui 1'**, **WRITE(*,*)'estou aqui 2'**, ..., ou com **WRITE(*,*)<resultado1>**, **WRITE(*,*)<resultado2>** e assim sucessivamente, para termos certeza do fluxo de execução do código. Vejamos um exemplo. Imagine que parte de um código de 320 linhas precisa calcular a tangente de um ângulo, para cada n, que é a soma de n vezes a expressão

$$\frac{1}{\mathrm{e}^n} + \mathrm{e}^n \,,$$

ou seja,

$$n \text{ vezes a } \operatorname{tg}\left(\sum_{n=1}^{j}\left(\frac{1}{\mathrm{e}^n} + \mathrm{e}^n\right)\right) \,,$$

em que $j = 200$ no nosso exemplo.

O programa abaixo reproduz parte do referido código. Por este motivo não deveria apresentar uma saída de valor, um **WRITE**, já que o mesmo é utilizado no decorrer do código original e assim, não precisaria ser impresso neste ponto do código. Mas foi acrescentado uma saída antes da finalização do programa, somente com fins didáticos, para visualizarmos que existem limitações computacionais que devem ser observadas.

Programa 1.14 – Exemplo de erros em códigos computacionais.

```
1   PROGRAM teste_erro
2   IMPLICIT NONE
3   INTEGER :: i, j, m
4   REAL :: x, y, z
5   REAL, PARAMETER : pi=3.14159265
```

```
132   DO i = 1,200        ! Inicia um loop com 200 repeticoes
133      x = 1./EXP(i) + EXP(i)
134      y = y + x
135      z = TAN(y)
136      theta = (y*180.0)/pi ! Converte angulo p/graus
137   END DO              ! Finaliza o loop com 200 repeticoes
138   WRITE(*,*) theta,z  ! Soh para fins didaticos
```

```
320 END PROGRAM teste_erro
```

Ao compilarmos o programa alguns erros serão detectados pelo compilador. O primeiro erro será o de sintaxe, porque foi esquecido os dois pontos (:) na declaração da variável **pi**, indicado pelo compilador, conforme mostra a figura 1.1. Relembrando, sempre que inicializarmos uma variável na sua declaração é obrigatório o uso de dois dois pontos (::).

Com erro deste porte é abortada a compilação. Portanto, os erros de lógica não aparecem. Mas cuidado, nem todos os erros de sintaxe são detectados nesta etapa. Por exemplo, se por engano escrever **TA** no lugar de **TAN**, para a função tangente, não aparecerá nesta primeira compilação. Quando for detectado, será como erro de definição de função.

Realizada esta correção, a colocação dos dois pontos (:), compila-se novamente. E aí a surpresa!!! Acompanhe a detecção dos erros de lógica na figura 1.2.

Perceba que o compilador procura auxiliar indicando o provável tipo e posição do(s) erro(s). Isto fica claro com a notação:

```
In file ex_erro.f90:7
      x = 1./EXP(i) + EXP(i)
                           1
Error: Type of argument 'x' in call 'exp' at (1)
should be REAL(4), not INTEGER(4)
```

que indica o arquivo (**ex_erro.f90**) e a linha **7**. Perceba ainda que o **1**, abaixo do **EXP(i)**, indica a provável posição do erro na linha. O provável tipo de erro (**Error:**) está no tipo de argumento da exponencial, que deve ser real de precisão simples, e não inteiro.

Neste caso foram dois tipos de erro, embora apareçam três na indicação do compilador. O primeiro é a respeito do argumento da exponencial, que espera um valor real e foi passado

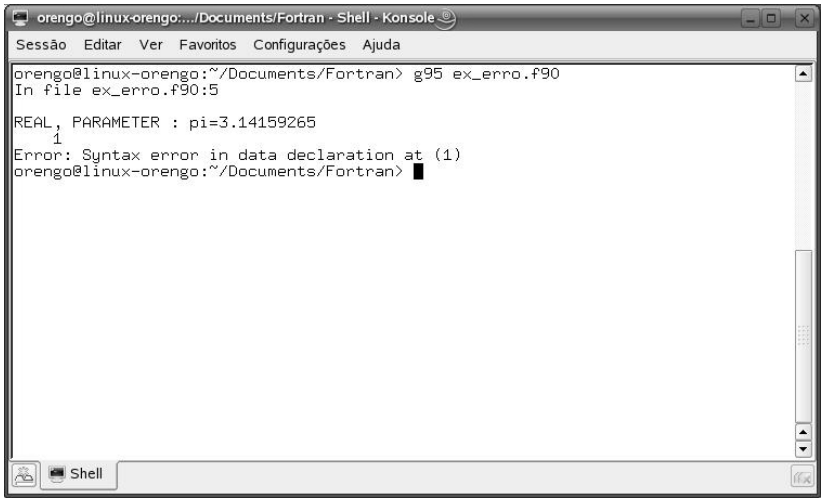

Figura 1.1 – No terminal do Linux: indicação do erro de sintaxe, apresentados pelo compilador G95.

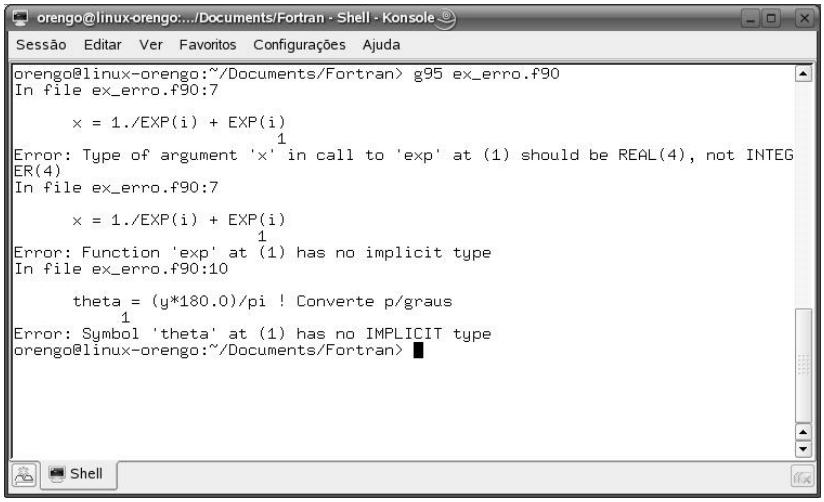

Figura 1.2 – Indicação dos erros apresentados pelo compilador G95.

um inteiro. A correção é feita utilizando a função **`REAL(i)`**, que converte o inteiro para real. Observe que não é possível trocar a declaração da variável **`i`** porque ela é um contador do loop **`DO`**, que será estudado no Capítulo 3, na página 63. E contador é sempre inteiro. O segundo erro, o indicado pelo compilador, é decorrente do primeiro, porque é esperado então que **`EXP(i)`** seja uma função definida pelo usuário já que estaria recebendo um valor inteiro. Mas, corrigido o primeiro erro, este deixa de existir. Comprove se é verdadeira a afirmaçao !!! Para isto, corrija somente o primeiro erro e re-compile.

O segundo erro de lógica (ou terceiro indicado pelo compilador) é a ausência de declaração da variável **theta**. Esta detecção foi possível pela presença do **IMPLICIT NONE** na segunda linha.

Corrigido, compilado e executado o resultado é surpreendente:

```
+Inf NaN
```

Não ocorreu o esperado, que eram resultados numéricos. O **+Inf** significa *overflow* (+ infinito), ou seja, número muito grande, que ultrapassou o limite do expoente da potência de dez. O **NaN** significa *Not-a-Number*, isto é, "não é um número".

Para identificar onde está o problema, foi acrescentado uma saída de informação, localizada no interior da estrutura de repetição **DO**. A saída utilizada foi:

```
WRITE(*,*)"O resultado da tangente de ",theta, "eh: ",z, &
          "(No loop: ", i, ")"
```

O objetivo é imprimir em cada passo (*loop*), o resultado parcial. Observe parte das 200 saídas:

```
O resultado da tangente de  1.0087063E+38 eh:  -10.567177 (No loop:  83 )
O resultado da tangente de  2.741948E+38 eh:  2.4247808 (No loop:  84 )
O resultado da tangente de  +Inf eh:  2.221997 (No loop:  85 )
O resultado da tangente de  +Inf eh:  1.0551705 (No loop:  86 )
O resultado da tangente de  +Inf eh:  0.4865762 (No loop:  87 )
O resultado da tangente de  +Inf eh:  10.228403 (No loop:  88 )
O resultado da tangente de  +Inf eh:  NaN (No loop:  89 )
```

No *loop* 85 já aparece o problema, que é a extrapolação do limite do expoente da potência de dez. A correção neste caso depende do problema que está sendo resolvido. Uma possível solução é a troca para precisão dupla das variáveis reais. Neste caso será necessário executar novamente com a mesma saída provisória, para verificar se o problema foi contornado. Outra possibilidade: se não é necessário realizar um *loop* com 200 repetições, pode-se alterar o *loop* para **DO i = 1,84**.

O programa 1.14, corrigido, é apresentado a seguir. As linhas que foram corrigidas estão destacadas e as linhas 137 e 138 podem ser descartadas depois de resolvido o erro.

Programa 1.15 – Programa 1.14 corrigido.

```
1   PROGRAM teste_erro
2   IMPLICIT NONE
3   INTEGER :: i, j, m
4   REAL :: x, y, z, theta                    ⟸
5   REAL, PARAMETER :: pi=3.14159265          ⟸
```

```
132   DO i = 1,200        ! Inicia um loop com 200 repeticoes
133      x = 1./EXP(REAL(i)) + EXP(REAL(i))    ⟸
134      y = y + x
135      z = TAN(y)
136      theta = (y*180.0)/pi ! Converte angulo p/graus
137      WRITE(*,*)"O resultado da tangente de ",theta,&
138                "eh: ", z, "(No loop: ", i, ")"
```

```
139    END DO                ! Finaliza o loop com 200 repeticoes
140    WRITE(*,*) theta,z    ! Soh para fins didaticos
```

```
320  END PROGRAM teste_erro
```

AS "SUJEIRAS" DA MEMÓRIA

Em vários momentos neste capítulo foi mencionado sobre as "sujeiras" da memória. E ainda, que as variáveis sejam inicializadas para evitar erros, justamente por conterem valores indevidos anteriormente armazenados no endereçamento de memória utilizado. Para verificar na prática, digite o programa abaixo, compile-o e execute-o.

Programa 1.16 – Exemplo de erro devido a "sujeiras" da memória.

```
1  PROGRAM testa_sujeira_memoria
2  IMPLICIT NONE
3  INTEGER :: a, b, c
4    WRITE(*,*) a + b + c
5  END PROGRAM testa_sujeira_memoria
```

Provavelmente terás uma surpresa, porque nenhum valor foi atribuído as variáveis **a**, **b** e **c** e, assim, espera-se um valor nulo ou até mesmo um erro de execução. Mas não é o que ocorre. No momento do fechamento desta edição, o valor obtido pela execução do programa no Windows foi **16384** e no Linux **-1073664380**. Se executarmos em outra ocasião, é muito provável que o valor seja diferente, porque dependerá dos valores contidos na memória. Então, cuidado!!! Sugestão? sempre inicialize as variáveis.

Mais informações a respeito de procura e correção de erros de programação em Fortran podem ser obtidas nos manuais dos compiladores Fortran, especialmente do compilador que será usado para testar os programas aqui apresentados e para os que serão solicitados nos exercícios a seguir.

Em cada capítulo, quando necessário, um capítulo sobre *debugging* será escrito. Isto demonstra a importância de reconhecer erros e corrígi-los para que o código de fato funcione como esperamos.

■ ■ ■ ■ ■

É importante para a verificação de aprendizado a prática com exercícios e a solução de problemas. Perceba que há uma diferença entre *solução de problemas* e *prática com exercícios*. O primeiro subentende que o segundo foi realizado exaustivamente. Estaremos mais preparados para resolver ou solucionar problemas se estivermos devidamente treinados. Portanto, é altamente recomendável que faça os exercícios a seguir, bem como os disponíveis no site *http://www.orengonline.com/fortran95/*.

As respostas e comentários a respeito dos exercícios encontram-se no referido endereço eletrônico, em *exercícios*. E, para verificar se está pronto para resolver problemas, um conjunto deles está disponível em *problemas*, também no site acima indicado.

Para reflexão !!

"Só existem duas coisas infinitas: o universo e a estupidez humana. E não estou muito seguro da primeira."

Albert Einstein
1879 – 1955

EXERCÍCIOS

1.1) Sejam as variáveis **a**, **b**, **c**, **d**, **e**, **f**, **g**, que são inicializadas como segue:
a = 3. b = 2. c = 5. d = 4. e = 10. f = 2. g = 3.
Avalie os seguintes procedimentos:

a) **y = a*b+c*d+e/f**g**

b) **y = a*(b+c)*d+(e/f)**g**

c) **y = a*(b+c)*(d+e)/f**g**

Siga o exemplo de solução:

a) - Expressão para cálculo: **y = a*b+c*d+e/f**g**

- Completando com os valores: **y = 3.*2.+5.*4.+10./2.**3.**

- Calculando `2.**3.`: **y = 3.*2.+5.*4.+10./8.**

- Calculando as multiplicações e divisões, da esquerda para a direita:

y = 6. +5.*4.+10./8.

y = 6. +20. +10./8.

y = 6. +20. +1.25

- Calculando as adições e o resultado final: **y = 27.25**

1.2) No Fortran, a operação entre inteiros e reais não é uma boa prática, pois o computador converte o número inteiro em real e a seguir procede a operação. Desta forma, podemos perder informações nestas operações. Informe, do ponto de vista do Fortran, qual será o resultado em cada uma das operações abaixo.

a) **3/2**
b) **3./2.**
c) **3./2**
d) **1 + 1/4**
e) **1. + 1/4**
f) **1 + 1./4**
g) **2 + 6./4**

1.3) O programa abaixo será compilado? Justifique sua resposta.

```
1 PROGRAM exemplo1
2 IMPLICIT NONE
3 INTEGER :: i,j
4 INTEGER, PARAMETER :: k=4
5    i = k**2
6    j = i/k
7    k = i+j
8    WRITE(*,*)i,j,k
9 END PROGRAM exemplo1
```

1.4) Que valores sairão do programa abaixo? Tente antes de executar o programa no computador, calcular as saídas. (O assunto é com respeito a **INT** e **NINT**. O **INT** retorna a parte inteira de um real por truncamento e o **NINT** por arredondamento)

```
1 PROGRAM exemplo2
2 IMPLICIT NONE
3 INTEGER :: i1, i2, i3
4 REAL :: r1 = 2.4, r2
5    i1 = r1
6    i2 = INT(r1*i1)
7    i3 = NINT(r1*i1)
8    r2 = r1**i1
9    WRITE(*,*)i1, i2, i3, r1, r2
10 END PROGRAM exemplo2
```

1.5) O programa abaixo tem o objetivo de calcular os comprimentos dos lados de um triângulo retângulo (A=adjacente e B=oposto), dados os valores da hipotenusa(C) e do ângulo(θ). O programa será executado? Se não, explique. Se sim, ele produzirá resultados corretos? Explique. Caso necessite de alteração, o que você modificaria ou acrescentaria?

```
1 PROGRAM exemplo3
2 REAL :: a, b, c, theta
3    WRITE(*,*) 'Entre com o comprimento da hipotenusa C:'
4    READ(*,*)c
5    WRITE(*,*) 'Entre com o valor do angulo THETA em graus:'
6    READ(*,*)theta
7    a = c*COS(theta)
8    b = c*SIN(theta)
9    WRITE(*,*)'O comprimento do lado adjacente eh:',a
10   WRITE(*,*)'O comprimento do oposto eh:       ',b
11 END PROGRAM exemplo3
```

1.6) Que valores sairão, quando as seguintes instruções forem executadas?

```
1  PROGRAM exemplo4
2  INTEGER :: i
3  LOGICAL :: l
4  REAL :: a
5     a = 0.05
6     i = NINT(2.*3.141593/a)
7     l = i>100
8     a = a*(5/3)
9     WRITE(*,*)i, a, l
10 END PROGRAM exemplo4
```

1.7) Escreva um programa em Fortran 95 que calcule o pagamento semanal de empregados horistas. O programa deverá perguntar ao usuário o valor pago por hora (por funcionário) e o número de horas trabalhada na semana. O cálculo deve se basear na seguinte expressão

$$\text{Total Pago} = \text{Valor da Hora} \times \text{Horas Trabalhadas} \tag{1.1}$$

Finalmente, você deverá mostrar o valor pago total. Teste seu programa calculando o valor pago semanalmente para uma pessoa que ganha R\$ $7,50$ por hora e trabalha 39 horas.

1.8) A distância entre dois pontos (x_1, y_1) e (x_2, y_2) no plano Cartesiano é dado pela equação

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{1.2}$$

Escreva um programa em Fortran 90/95 para calcular a distância entre quaisquer dois pontos (x_1, y_1) e (x_2, y_2) especificados pelo usuário. Use boas práticas de programação no seu programa. Calcule a distância entre os pontos (2,3) e (8,-5).

1.9) Desenvolva um programa que fornecido, pelo usuário, um número entre 1000 e 9999, separe e escreva os valores referentes ao milhar, a centena, a dezena e a unidade. *Exemplo: no caso do número 7452, milhar: 7000; centena: 400; dezena: 50 e a unidade: 2.*

1.10) O aumento dos funcionários de uma fábrica é calculado com base em 80% do IGP-M acumulado no ano (Índice Geral de Preços do Mercado, da Fundação Getúlio Vargas – FGV). Faça um programa que leia o número do funcionário, o valor do IGP-M e o salário atual do funcionário. A seguir calcule o aumento e o novo salário. Escreva o número do funcionário, o aumento e o novo salário. (*Obs.: use os seguintes nomes às variáveis: número do funcionário = nf; IGP-M = igpm; aumento do salário = as; salário atual = satual; novo salário = ns.)*

1.11) O custo final ao consumidor de um carro novo é a soma do custo de fábrica com a percentagem do distribuidor e dos impostos (aplicados ao custo de fábrica). Supondo que a percentagem do distribuidor seja de 28% e os impostos de 45%, escreva um programa que leia o custo de fábrica de um carro e escreva o custo final ao consumidor. (*Obs.: 1) Use os seguintes nomes às variáveis: custo de fábrica = cf; custo final ao consumidor = cfc; percentagem do distribuidor = pd; impostos = pi. 2) Construa o programa com o máximo de informações ao usuário)*

1.12) Escreva um programa que calcule a posição, velocidade e aceleração de um corpo sujeito ao movimento harmônico simples, baseado nas equações abaixo. Para iniciar, use $\epsilon = 0$, $n = 3.14159265$, $b = 2.5$. Teste para um conjunto de valores de t.

$$\begin{aligned} \text{posicao} &= x = b\sin(nt + \epsilon) \\ \text{velocidade} &= v = nb\cos(nt + \epsilon) \\ \text{aceleracao} &= a = -n^2 b\sin(nt + \epsilon) \end{aligned}$$

CAPÍTULO 2

TRABALHANDO COM ARQUIVOS – ENTRADAS/SAÍDAS (I/O) DE DADOS

Neste capítulo você encontrará:

Para reflexão !!

"Se um dia tiver que escolher entre o mundo e o amor... Lembre-se: Se escolher o mundo, ficará sem o amor, mas se escolher o amor, com ele conquistará o mundo!!"

Albert Einstein
1879 – 1955

2.1 INTRODUÇÃO

As entradas e saídas de dados em Fortran são realizadas pelas ***unidades* lógicas**. Uma unidade lógica é:

- um número inteiro não negativo associado a um dispositivo físico tal como uma unidade de disco (HD, disquete, CD,...), teclado, monitor ou uma impressora. A unidade lógica é conectada a um arquivo ou dispositivo pela instrução **OPEN** (ver Seção 2.4, na pág. 53), exceto nos casos dos arquivos pré-conectados.

- um asterisco, "*", indica o arquivo ou dispositivo padrão (*default*), pré-conectado, de entrada e de saída, usualmente o teclado e a tela do monitor, respectivamente. Leia a nota da página 54.

- uma variável tipo **CHARACTER** corresponde ao nome de um arquivo interno.

ARQUIVO

O Fortran trata todos os dispositivos físicos tais como unidades de discos, teclado, impressora, monitor ou arquivos internos como **arquivo** (*file*) ou **arquivo externo**. A *estrutura do arquivo* é determinada pelos dados (formatados ou não), o tipo de acesso ao arquivo e espaço (comprimento) para a informação.

O Fortran fornece instruções e procedimentos para manipular dados por leitura e escrita, conhecidas por entradas e saídas de dados, inclusive com o uso de arquivos internos. Do inglês vem a denominação *Input/Output*, ou simplesmente I/O, para Entrada/Saída de dados. Entre as instruções disponíveis veremos somente algumas, a saber: **WRITE** e **READ**, que são responsáveis pela transferência de dados, **OPEN**, que conecta as unidades lógicas aos arquivos e **FORMAT**, que fornece informações explícitas de formato dos dados. O objetivo deste livro é dar uma formação básica, portanto não serão tratados assuntos relativos a tratamento de posicionamento interno num arquivo, tipos de arquivos, entre outros. Para um avanço no estudo da linguagem Fortran, consulte as referências bibliográficas, na pág. 135.

2.2 A INSTRUÇÃO WRITE

A instrução **WRITE** é utilizada para transferir dados para o arquivo de saída (*Output*), ou seja, escreve os resultados ou informações de saída do programa.

A sintaxe é:

```
WRITE([UNIT=]<unidade>,[FMT=]<formato>,[ADVANCE=<modo>])
```

em que:

[UNIT=]<unidade>: é um argumento obrigatório e indica a unidade lógica (dispositivo[N1]) para a qual será transferido o valor (ou informação) contido na memória. Como unidade padrão (*default*) é utilizada a tela do monitor, e é indicado por um asterisco (*). Uma unidade lógica diferente da padrão é determinada pela instrução **OPEN** (ver Seção 2.4, na pág. 53). A palavra-chave **UNIT** é opcional[N2] e na maioria dos programas não é utilizada.

[FMT=]<formato>: é um argumento obrigatório e especifica o(s) formato(s) para a escrita dos dados. Para instruir o compilador a escrever numa formatação livre (ou "lista-dirigida") é utilizado o asterisco (*), isto é, o resultado ou a informação será escrita da forma como foi gerado, em acordo com a declaração da variável que contém a informação. Uma saída com formatação livre não tem boa apresentação visual. Uma formatação diferente da padrão é obtida pela instrução **FORMAT**, que será descrita mais adiante (Seção 2.6, na pág. 56). A palavra-chave **FMT** é opcional e não é utilizada na maioria dos programas.

[N1] Dispositivo significa algum meio físico como por exemplo, tela do monitor, a impressora, um arquivo numa unidade de disco seja magnética ou CD/DVD, o teclado, ou outro.

[N2] Neste livro, toda instrução ou argumento escrito entre colchetes ([...]) é considerado opcional.

ADVANCE=<modo>: argumento opcional, especifica se a próxima saída (ou entrada) deve iniciar numa nova linha ou não. O padrão (*default*), na ausência do parâmetro, é **<modo>="YES"**, ou seja, a nova saída avançará para uma nova posição ou nova linha. Adotando o argumento **<modo>** como **"NO"** não ocorrerá avanço para a nova linha ou posição de escrita. Uma aplicação pode ser vista na pág. 52, no exemplo da instrução **READ**.

Vejamos o seguinte exemplo.

Programa 2.1 – Exemplo de saída de dados.

```
1  PROGRAM saida
2  IMPLICIT NONE
3  INTEGER :: a=10, b=20, c=135
4     WRITE(*,*) a
5     OPEN(UNIT=10, FILE="saida1.txt")
6     WRITE(10,*) b
7     WRITE(10,200) c
8  200 FORMAT(I3)
9     CLOSE(10)
10 END PROGRAM saida
```

No primeiro **WRITE**, o conteúdo da variável "**a**" será escrito na tela do monitor e em formato livre. No segundo, o resultado armazenado na variável "**b**" será escrito no arquivo de nome *saida1.txt*, identificado pela unidade lógica 10, procedimento permitido pela instrução **OPEN**. Já a outra saída de resultado, pela variável "**c**", é também no arquivo anterior, mas formatada pela instrução **FORMAT**, localizada no programa pelo rótulo número 200, que neste caso instrui o compilador a imprimir um número (do tipo inteiro) com 3 algarismos. Como exemplo, o número 135 poderá ser impresso ajustadamente no espaço designado. Quando o número não couber nos espaços determinados pela formatação uma sequência de asteriscos será impressa no lugar do número ou da informação. A instrução **CLOSE**, que encerra a atividade da unidade 10, é estudada na pág. 54.

2.3 A INSTRUÇÃO READ

A instrução **READ** transfere dados de uma unidade de entrada (*Input*) para uma variável, isto é, lê dados que alimentarão com valores e/ou informações o programa em Fortran.

A sintaxe é:

```
READ([UNIT=]<unidade>,[FMT=]<formato>)
```

em que:

[UNIT=]<unidade>: é a unidade lógica (dispositivo) da qual será obtido o valor (ou informação). Como unidade padrão é utilizado o teclado e é indicado por um asterisco (*). Entenda-se por teclado, os dados digitados por este dispositivo. Uma unidade lógica diferente da padrão é determinada pela instrução **OPEN**. A palavra-chave **UNIT** é opcional, e não é utilizada na maioria dos programas.

[FMT=]<formato>: especifica com que formato(s) a leitura dos dados é realizada. Para instruir o compilador a ler numa formatação livre (ou lista-direta) é utilizado o asterisco (*). Neste caso, a memória receberá uma informação sem nenhum tipo de preocupação com a editoração da informação. Uma formatação diferente da livre é obtida pela instrução **FORMAT**, que será descrita na Seção 2.6 (na pág. 56). A palavra-chave **FMT** é opcional e não é utilizada na maioria dos programas.

O argumento **ADVANCE** apresentado na instrução **WRITE** também pode ser utilizado aqui, com o mesmo significado.

Vejamos um exemplo:

Programa 2.2 – Exemplo de entrada de dados.

```
1  PROGRAM leitura
2  IMPLICIT NONE
3  REAL :: a, b, c
4     WRITE(*,60,ADVANCE="NO")"Digite um numero real: "
5     READ(*,*) a
6     OPEN(UNIT=20, FILE="dados1.txt")
7     READ(20,*) b
8     READ(20,50) c
9  50 FORMAT(F5.2)
10 60 FORMAT(A)
11    CLOSE(20)
12 END PROGRAM leitura
```

No primeiro **READ**, o conteúdo da variável "**a**" será preenchido via teclado e em formato livre. Utilizar o teclado significa digitar uma informação, compatível com o tipo da declaração da variável, e na sequência teclar <*ENTER*>[N3]. Observe que no **WRITE** é aplicado o argumento **ADVANCE**, e a próxima instrução de leitura (ou de escrita) não avançará para a próxima linha. Assim, na sequência, o cursor ficará posicionado ao lado da saída anterior aguardando a leitura da variável "**a**".

FALAR SOBRE O 60 format

No segundo **READ**, a variável "**b**" receberá um valor que está armazenado num arquivo de nome *dados1.txt*, identificado pela unidade lógica 20, anteriormente habilitada pela instrução **OPEN**. Já a outra entrada de informação, pela variável "**c**", é também oriunda do arquivo anterior, mas formatada pela instrução **FORMAT**, localizada no programa pelo rótulo número 50, que neste caso instrui o compilador a ler um número, do tipo real, com 5 espaços, sendo dois para as casas decimais após a vírgula. Neste caso, o número 56.50 caberá ajustadamente, pois tem 5 espaços (caracteres) e com duas casas decimais. Perceba que o ponto decimal faz parte da contagem dos espaços disponíveis. Quando o número não couber nos espaços determinados pela formatação, o compilador arredondará o número, fazendo-o se ajustar na formatação determinada ou imprimirá asteriscos.

[N3]Usaremos a convenção <*ENTER*> para designar teclar *ENTER*.

CONSIDERAÇÕES ADICIONAIS SOBRE `WRITE/READ`

2.1) É possível ter múltiplas saídas e/ou entradas de dados, da seguinte forma:

```
WRITE(*,*) a, c, resultado
READ(*,*) b,g
```

nas quais, as variáveis (separadas por vírgulas) "**a**", "**c**" e "**resultado**" fornecem saídas; e as variáveis "**b**" e "**g**" recebem valores de entrada.

2.2) Para lermos uma informação do tipo **CHARACTER** é necessário alterar a instrução de leitura da seguinte forma:

```
READ(*,'(A)') b
```

em que o argumento **'(A)'** especifica a leitura de caracter, exigindo assim a declaração da variável "**b**" como segue

```
CHARACTER :: b
```

ou, por exemplo

```
CHARACTER(LEN=3) :: b
```

Nesta última declaração, a variável armazenará um caracter de comprimento igual até 3 unidades.

2.4 A INSTRUÇÃO `OPEN`

Esta instrução conecta ou reconecta um arquivo externo a uma unidade lógica de entrada ou de saída. Vimos que os dispositivos padrões de entrada e saída, na maioria dos compiladores Fortran 95, são respectivamente, o teclado e a tela do monitor. A instrução **OPEN** permite alterar o dispositivo de entrada e de saída, que é realizada na seguinte sequência: associa-se o nome de um arquivo externo a uma unidade lógica, que será usada nas instruções de entrada e saída de dados, e atribui-se um estado (*status*) ao arquivo.

A sintaxe é:

```
OPEN([UNIT=]<número>, FILE="<nome_arq.>", [STATUS="<estado>"])
```

e deve estar localizada antes da unidade lógica ser utilizada. Os argumentos da instrução são:

[UNIT=]<número>: é o argumento que se refere a unidade lógica e ao seu respectivo arquivo de entrada/saída; **UNIT** é uma palavra-chave e o **<número>** é um número inteiro não negativo, sendo que o único cuidado é para não coincidir com o número adotado pelo compilador para os dispositivos padrões, de entrada e saída. Neste sentido, aconselha-se adotar numeração acima de 8. A palavra-chave **UNIT=** é opcional.

FILE="<nome_arq.>": fornece o nome associado com a unidade lógica. É importante que o nome contenha uma extensão, por exemplo "**dados1.txt**".

STATUS="<estado>": é um argumento opcional e fornece o estado do arquivo, e **<estado>** pode ser:

a) **OLD**: o arquivo já existe,

b) **NEW**: o arquivo não existe, e será criado,

c) **REPLACE**: um novo arquivo será aberto. Se o arquivo já existe os dados serão apagados, para ser *re-utilizado*,

d) **STRATCH**: o arquivo será criado temporariamente, e após o uso será eliminado. Neste caso o argumento **FILE** ***não pode ser especificado***, isto é, o arquivo não terá nome.

e) **UNKNOWN**: desconhecido.
Este é o argumento padrão por definição (*default*), na ausência do **STATUS**.

Exemplos:

```
OPEN(UNIT=10, FILE="resultados.txt", STATUS="NEW")
OPEN(UNIT=20, FILE="dados1.txt", STATUS="OLD")
OPEN(50, FILE='entrada1.txt')
```

No primeiro exemplo, uma unidade lógica com o número 10 é criada e cuja unidade física (ou arquivo) de nome *resultados.txt* conterá e/ou estará habilitado para armazenar valores. Perceba que é um arquivo novo, ou seja, ele será criado na execução do programa. Caso o arquivo interno já exista, o compilador retornará uma mensagem de erro, e encerrará a execução. No segundo exemplo, a unidade lógica será de número 20 e o arquivo é *dados1.txt* existente, caso contrário o compilador acusará um erro de inexistência do arquivo interno. No último caso, a unidade lógica é a 50 e o arquivo *entrada1.txt*, mas observe que a palavra-chave **UNIT** foi omitida, bem como o *status* e, portanto, é considerado como desconhecido.

AS UNIDADES PRÉ-CONECTADAS

Foi mencionado que a numeração adotada para a unidade lógica não pode ser qualquer. Os compiladores designam números inteiros para as suas unidades lógicas padrão, chamadas de *Unidades de Entrada/Saída Pré-conectadas*. Por exemplo, a unidade lógica 5 está conectada ao dispositivo padrão de entrada, usualmente o teclado, e a unidade 6 para o dispositivo de saída, usualmente a tela do monitor. Assim, nas instruções de entrada/saída de dados o asterisco representa estas unidades

2.5 A INSTRUÇÃO CLOSE

Quando uma unidade lógica está conectada ao seu arquivo externo, ele permanece neste estado até o sua desconexão, ou até que cesse a execução do programa. Em outras palavras,

nesta situação um arquivo permanece aberto. Assim, após o uso do arquivo é importante fechá-lo para evitar erros indesejáveis. A instrução **CLOSE** desconecta uma unidade lógica do seu arquivo externo. Ou seja, fecha o acesso ao arquivo e libera o número da unidade lógica de I/O associada a ela.

A sintaxe é:

```
CLOSE([UNIT=]<número>)
```

em que:

<número>: é o número da unidade lógica definida no argumento **UNIT** da instrução **OPEN**. Podemos fechar vários arquivos simultaneamente, bastando para isso informar as unidades separadas por vírgulas.

Exemplos:

```
CLOSE(10,12,20)
CLOSE(UNIT=11)
CLOSE(50)
```

Observe que no primeiro exemplo a instrução **CLOSE** encerra simultaneamente 3 unidades lógicas. Outro ponto importante é que a palavra-chave **UNIT** é opcional e só foi utilizada no segundo exemplo.

Para a instrução **OPEN** há outros argumentos, que não serão tratados aqui, e é aconselhável verificá-los nos livros indicados nas referências bibliográficas, na pág. 135.

Exemplos resolvidos:

1) Escreva um programa em Fortran 95 que leia um valor inteiro via dispositivo padrão, multiplique-o por dez e escreva o resultado em um arquivo com nome *saida1.txt*.

Solução: *Adotaremos como unidade de saída a 50, e as variáveis "a" e "b", respectivamente, para entrada e saída de dados. Assim, o código abaixo resolve o nosso problema. Digite-o, compile-o e execute-o.*

```
PROGRAM open1
IMPLICIT NONE
INTEGER :: a,b
OPEN(UNIT=50,FILE="saida1.txt")
  WRITE(*,*)"Digite um no. inteiro"
  READ(*,*) a
  b = a*10
  WRITE(50,*)b
  CLOSE(50)
END PROGRAM open1
```

2) Escreva um programa em Fortran 95 que leia um valor inteiro armazenado num arquivo de nome *saida1.txt* (do exemplo anterior), multiplique-o por mil e escreva o resultado em um arquivo com nome *saida2.txt*.

Solução: *Adotaremos como unidade de entrada a 60 e a de saída a 61 e, as variáveis "a" e "b", respectivamente, para entrada e saída de dados. Devemos lembrar que o arquivo de leitura já existe e, portanto, o argumento* **STATUS** *terá como parâmetro* **"OLD"**. *O código abaixo resolve o problema. Digite-o, compile-o e execute-o. Observe o uso da instrução* **&** *para quebra de linha no Fortran.*

```
PROGRAM open2
IMPLICIT NONE
INTEGER :: a,b
OPEN(UNIT=60,FILE="saida1.txt",&
    & STATUS="OLD")
OPEN(61,FILE="saida2.txt")
  READ(60,*) a
  b = a*1000
  WRITE(61,*)b
  CLOSE(60)
  CLOSE(61)
END PROGRAM open2
```

Podemos atribuir à unidade lógica uma variável, conforme o exemplo abaixo, no qual a variável "**ue**", cujo valor inteiro é informado pelo usuário, substitui o inteiro no argumento **UNIT** dos exemplos acima.

Programa 2.3 – Exemplo do uso de unidades lógicas.

```
1  PROGRAM unidade1
2  IMPLICIT NONE
3  INTEGER :: a,b,ue
4    WRITE(*,*)"Digite um numero inteiro:"
5    READ(*,*) a
6    b = a*10
7    WRITE(*,*)"Informe o numero (>8) da unidade logica &
8                &de saida:"
9    READ(*,*) ue
10   OPEN(UNIT=ue,FILE="saida1.txt")
11   WRITE(ue,*)b
12   CLOSE(ue)
13 END PROGRAM unidade1
```

2.6 FORMATANDO AS SAÍDAS E/OU ENTRADAS (FORMAT)

Nos exemplos anteriores utilizamos entradas e saídas de dados sem formatação, isto é, da maneira como foram gerados os dados (ou resultados) elas foram impressas, seja na tela do monitor ou num arquivo. É possível converter a representação interna, como está na memória, conforme a declaração da variável, para uma representação externa, utilizando a instrução não executável **FORMAT**. Este comando simplesmente instrui o compilador que a saída (ou entrada) de dado obedecerá uma formatação estabelecida.

A sintaxe da instrução **FORMAT** é:

```
<rótulo> FORMAT(<código-chave>)
```

em que:

<rótulo>: é um número inteiro associado a instrução **FORMAT**, o qual será o número de chamada por uma instrução **READ** e/ou **WRITE**. A instrução **FORMAT** pode ser posicionada em qualquer linha no programa, após as declarações de variáveis.

<código-chave>: especifica o(s) formato(s) adotado para entrada e/ou saída de dados, baseado na tabela 2.1. Podem ser utilizados mais de um código separados por vírgula.

Até agora todas as saídas e entradas de dados foram realizadas sem preocupação com o formato. A partir deste momento deveremos ter cuidado ao utilizar a instrução **FORMAT**. Veremos alguns pontos importantes.

OS CÓDIGOS-CHAVE

Existe mais de uma dúzia de espécies de formatação (códigos-chave) em Fortran 95, mas usaremos somente alguns, descrito na tabela 2.1. Para auxiliar na compreensão da tabela temos que:

w: é a largura do campo (número de dígitos) destinado ao referido dado.
A largura do campo (**w**) inclui o número total de posições, inclusive o sinal e o ponto decimal da mantissa, o símbolo **E**, o sinal (de + ou de −) e os dois dígitos do expoente (ou os dígitos determinados para o expoente, **e**).;

d: corresponde aos dígitos após o ponto decimal espaço (casas decimais da mantissa);

e: os dígitos do expoente **E**;

n: significa o número de vezes que a formatação será utilizada, ou seja, indica uma repetição. Por exemplo, no caso de **3A8**, significa que serão impressos ou lidos 3 variáveis do tipo **CHARACTER** de tamanho igual a 8 cada.

Tabela 2.1 – Descrição de algumas formatações utilizadas na instrução **FORMAT**.

Código-chave	Significado
nAw	Interpreta os próximos **w** caracteres como um conjunto de caracteres (código-chave **A**) de texto.
nIw	Interpreta os próximos **w** dígitos como um número inteiro (código-chave **I**).
nFw.d	Interpreta os próximos **w** dígitos como um número real, numa notação sem potência de dez (código-chave **E**), com **d** casas após o ponto decimal.
nEw.d[Ee]	Interpreta os próximos **w** dígitos como um número real, de precisão simples, numa notação com potência de dez (código-chave **E**), com **d** casas após o ponto decimal.
nESw.d[Ee]	Interpreta os próximos **w** dígitos como um número real, de precisão simples, numa notação científica de potência de dez (código-chave **ES**), com **d** casas após o ponto decimal.
nDw.d[Ee]	Interpreta os próximos **w** dígitos como um número real, de precisão dupla, numa notação com potência de dez (código-chave **D**), com **d** casas após o ponto decimal.
nX	Fornece **n** espaços horizontais (código-chave **X**).
Tc	É o conhecido *Tab*: move a leitura/escrita para a coluna **c**, em que **c** é o número da coluna.

Mais informações sobre os tipos de formatação (códigos-chave) podem ser obtidas nas referências indicadas no final do livro.

Sempre que o tamanho da informação for menor que o espaço reservado, a informação será escrita justificada à direita e serão acrescentados espaços em branco antes, para com-

pletar o espaço reservado. Quando a saída é sem formatação o que determinará o espaço reservado será a definição de precisão adotada. Por exemplo, um número inteiro de precisão simples terá 11 espaços reservados para a saída ou leitura, já incluído o sinal. O sinal positivo (+) é omitido.

O TAMANHO DO PAPEL DE IMPRESSÃO

Antes do computador enviar a informação para o dispositivo de saída, é construído uma imagem de cada linha. A memória do computador que contém esta imagem é chamada de **área de armazenamento de dados temporário** (*output buffer*). A largura de cada linha é igual a 133 caracteres, sendo o primeiro espaço destinado ao **caracter de controle**, que veremos a seguir. Os demais 132 espaços são para armazenar a saída de dados. Assim, cada impressão ou saída de informações é dividida em páginas, de tamanho aproximadamente igual a 37,8 cm para a largura e 27,9 cm para a altura. Estas dimensões estão relacionadas com os formulários contínuos muito utilizados até o final do século passado. Cada página é dividida em linhas, e cada linha em 132 colunas, com um caracter por coluna. Para estas medidas, e dependendo do número de linhas por polegada que a impressora imprime, correspondem entre 60 e 72 linhas. As margens superior e inferior equivalem aproxidamente a 1,27 cm.

O CARACTER DE CONTROLE

Este item é muito importante na saída de dados pelo **`WRITE`**. O primeiro caracter da imagem, contida na **área de armazenamento de dados temporário**, é conhecido como **caracter de controle** e especifica o espaçamento vertical da saída. A tabela 2.2 apresenta as possibilidades e as ações deste caracter de controle.

Tabela 2.2 – Caracteres de controle da saída formatada.

Caracter de controle	Ação
1	Avança para a próxima página (Nova página)
<espaço>	Espaçamento simples entre linhas
0	Espaçamento duplo entre linhas
+	Não há espaçamento entre linhas. Sobrescreve (escreve em cima da) a linha anterior

Se qualquer outro caracter for utilizado como caracter de controle resultará em espaçamento simples entre linhas, como se fosse usado o espaço em branco (<espaço>).

Para evitar erros na saída, não esqueça do caracter de controle. Por exemplo, a seguinte instrução de saída

```
    WRITE(*,100) a
100 FORMAT(I2)
```

imprimirá um valor numérico inteiro de duas posições. Esta é a intenção. Vejamos dois possíveis valores. Se for passado para a variável **`a`** o valor 7, a saída será exatamente 7, justificada a direita. Mas se for passado o valor 71 o resultado será 1, porque o primeiro

caracter é destinado ao de controle, neste caso o 7 e foi assumido como espaço em branco, portanto o espaçamento entre linhas será simples. No caso anterior não ocorreu erro porque o primeiro caracter foi um espaço em branco, já que o número tinha somente um algarismo. Então muita atenção.

Alguns exemplos de formatação são apresentados a seguir.

Programa 2.4 – Exemplo do uso de formatos.

```
1  PROGRAM format1
2  IMPLICIT NONE
3  REAL :: a = 123.45, b = 6789.10, c = 0.000778
4     WRITE(*,100) a     ! a saida: 0.12345E+03
5     WRITE(*,200) a     ! a saida: 123.45
6     WRITE(*,300) a     ! a saida: 0.123450E+03
7     WRITE(*,400) a     ! a saida:   0.12E+03
8     WRITE(*,500) a     ! a saida: 0.12345E+000003
9     WRITE(*,600) a     ! a saida:   1.23E+02
10    WRITE(*,700) a,b,c ! a saida:   1.23E+02  6.79E+03  7.78E-04
11 100 FORMAT('0',E11.5)
12 200 FORMAT('0',F6.2)
13 300 FORMAT('1',E12.6)
14 400 FORMAT(' ',E10.2)
15 500 FORMAT(' ',E15.5E6)
16 600 FORMAT(' ',ES10.2)
17 700 FORMAT('+',3ES10.2)
18 END PROGRAM format1
```

Observe as saídas escritas como comentário[N4]. Atente, especialmente para a segunda formatação, que é exatamente o número original. As demais alteram a forma de escrever e não mudam o significado matemático da variável "**a**", a não ser nas últimas que reduzem a precisão do valor. Na última, foram realizadas três saídas com a mesma formatação, definida no **FORMAT** de rótulo 700. Nesta é utilizada a formatação em notação científica, e observe o caso do número 6789.10, que é arredondado para se ajustar ao espaço determinado e, no número 0.000778 os zeros são abandonados e ajustados no expoente da notação científica. As duas primeiras saídas sairão na página corrente, com espaçamento duplo entre linhas. Na terceira saída, o caracter de controle instrui que o resultado seja escrito numa página nova. Os demais serão escritos nesta mesma página com espaçamento simples entre as linhas. A última saída sobrescreverá a anterior.

Mais informações a respeito de formatação de saída/entrada de dados são obtidas nos livros sobre Fortran, citados nas referências bibliográficas, na pág. 135.

Quando o número não se ajusta na formatação determinada, um conjunto de asteriscos é impresso no lugar do número ou da expressão. Veja o exemplo abaixo.

[N4]Lembrete: um comentário em Fortran é inserido pela instrução **!** (sinal de exclamação) antes da sentença que desejamos comentar. Pode aparecer em qualquer parte do programa.

Programa 2.5 – Exemplo de formatação inadequada.

```
1 PROGRAM format2
2 IMPLICIT NONE
3 INTEGER :: a1 = 12345, a2 = 678
4    WRITE(*,100) a1     ! a saida: ****
5    WRITE(*,100) a2     ! a saida:  678
6 100 FORMAT(' ',I4)
7 END PROGRAM format2
```

Na primeira saída os asteriscos substituiram o número 12345, porque o espaço reservado para sua impressão é menor que o tamanho (em espaços) do número. Na segunda saída uma coluna em branco foi acrescentada antes do número 678 para completar o número de casas da declaração formatada, já que ele é menor que o espaço reservado. Ou seja, sempre que o número for menor que o espaço reservado, o número será escrito justificado à direita.

As formatações podem ser utilizadas diretamente nas instruções **READ** e **WRITE**. Por exemplo, as seguintes instruções

```
   WRITE(*,100) a1, a2
   READ(*,200) a3, a4
100 FORMAT(' ',I3,2X,E10.2)
200 FORMAT(2I5)
```

podem ser substituídas por

```
   WRITE(*,'(' ',I3,2X,E10.2)') a1, a2
   READ(*,'(2I5)') a3, a4
```

2.7 CORRIGINDO ERROS – *DEBUGGING*

Quando utilizamos entradas e saídas formatadas via arquivo é necessário muita atenção. Veremos dois casos de posicionamento dos dados de entrada.

ENTRADA DE DADOS EM UMA ÚNICA LINHA DE UM ARQUIVO

O programa abaixo recebe dados reais de um arquivo em uma única entrada, pela instrução **READ**, conforme uma formatação específica.

Programa 2.6 – Exemplo de erro devido a formatação inadequada.

```
1 PROGRAM format_real
2 IMPLICIT NONE
3 REAL :: a, b, c
4    OPEN(UNIT=10, FILE="dados.txt", STATUS="OLD")
5    WRITE(*,*) "Digite dois numeros reais para soma-los: "
6    READ(10,100) a, b ! Unica entrada
7    c = a + b
```

```
8     WRITE(*,100) c
9   100 FORMAT(F5.2)
10  END PROGRAM format_real
```

O arquivo *dados.txt* contém os seguintes valores, em que o sómbolo ␣ indica espaço em branco:

```
30.40␣␣␣21.50
```

A saída será:

```
␣50.60
```

■ ■ ■ ■ ■

A verificação de aprendizado é pela prática com exercícios e com a solução de problemas. É altamente recomendável que faça os exercícios a seguir, bem como os disponíveis no site *http://www.orengonline.com/fortran95/*. Após este treinamento estará apto a resolver problemas.

As respostas e comentários a respeito dos exercícios encontram-se no referido endereço eletrônico, em *exercícios*. E, para verificar se está pronto para resolver problemas, um conjunto deles está disponível em *problemas*, também no site acima indicado.

É importante lembrar que este livro é introdutório e, portanto, não contém todas as capacidades mais avançadas das instruções Fortran. Para uma abordagem neste nível aconselha-se os livros listados nas referências biblioigráficas.

EXERCÍCIOS

2.1) O que será impresso nas seguintes instruções Fortran? Use a = -345 e b = 1.0020E6 e c = 1.0001E6

a)
```
INTEGER ::  a
WRITE(*,100) a
100 FORMAT('1','a = ', I4.1)
```

b)
```
WRITE(*,'(' ', ES12.6)') b
```

c)
```
WRITE(*,'(' ',"O resultado da diferenca entre ",ES14.6, "e ",
ES14.6, " eh:  ", ES14.6)') b,a,b-a
```

2.2) Qual é a largura (**w**) mínima necessária para imprimir um valor real num formato de notação com potência de dez (**E**) e numa notação científica de potência de dez (**ES**) com 5 bits significativos de precisão? *Obs.: consulte a respeito de precisão numérica na página 22, no Capítulo 1.*

2.3) Escreva um programa que leia dois valores reais de precisão simples de um arquivo de nome *entrada.txt* e escreva a soma deles em um novo arquivo, já existente, de nome *saida.txt*. Use a instrução **FORMAT** para os valores.

2.4)

CAPÍTULO 3

AS ESTRUTURAS DE CONTROLE

Neste capítulo você encontrará:

Saudade (09/09/2004 - 11:00)

Eternidade: tanto faz
dia após dia se faz
Saudade: esta faz
o tempo ser sagaz

Gilberto Orengo
1961 –

3.1 INTRODUÇÃO

Todos os programas vistos até o momento executam uma série de procedimentos um após o outro numa ordem fixa. Estes são conhecidos como *programas ou códigos seqüenciais* e são as formas mais simples de execução de uma tarefa. Neles são lidos e processados dados de entrada que, ao final imprimem o resultado, sem que em algum momento seja preciso decidir sobre uma ou outra sequência de execução, ou ainda sem repetição de um certo processo. Códigos mais elaborados necessitarão de procedimentos que permitam o controle na ordem ou no fluxo de execução de procedimentos, como por exemplo, "pulando" para outra parte do programa ou até mesmo para outra tarefa (subtarefa - **`SUBROUTINE`** e **`FUNCTION`**, que serão estudadas adiante, na pág. 97).

Este tipo de controle, na estrutura do fluxo de execução do programa, é uma das características de uma *linguagem estruturada*, como o Fortran.

As duas categorias de controle de procedimentos disponíveis no Fortran, são:

- **TOMADAS DE DECISÃO (Desvios)** ou ramificações, permitem selecionar um desvio para o fluxo de execução do programa, em acordo com uma condição lógica. Veremos as seguintes instruções de decisão:
  - **`IF...ELSE IF...ELSE...END DO`**, nas páginas 67 e 69,
  - **`IF`** lógico, na página 71,
  - **`SELECT CASE`**, na página 71;
- **PROCESSOS DE REPETIÇÃO (*loops*)**, instruem que uma parte do código seja executada repetidamente, sob uma determinada condição. Veremos as seguintes instruções de repetição:
  - **`DO...END DO`**, na página 73,
  - **`DO...IF...END DO`** (*loop* infinito), na página 74.

Para usarmos estas estruturas é necessário vermos inicialmente as expressões aritméticas e lógicas, nas quais são utilizados os operadores aritméticos, relacionais e lógicos, que servirão de base para os testes de lógica no controle do fluxo de execução do programa.

3.1.1 Expressões Aritméticas

Embora tenhamos visto as expressões aritméticas no Capítulo 1, repetiremos aqui. As expressões aritméticas são aquelas que apresentam como resultado um valor numérico que pode ser um número inteiro ou real, dependendo dos operandos e operadores. Os operadores aritméticos estão descritos na Tabela 3.1.

Tabela 3.1 – Os operadores aritméticos disponíveis no Fortran 95.

Operadores Aritméticos		
Operador	Função	Exemplos
*	multiplicação	5*2; A*C
/	divisão	$2/7$; $8.5/3.1$; H/B
$+$	soma ou adição	$3+5$; A + J
$-$	subtração	$5-1$; A1 − B
**	potenciação	3**5 $\Longrightarrow 3^5$

3.1.2 Expressões Lógicas

Uma expressão lógica é aquela que possui operadores lógicos e/ou relacionais.

OPERADORES RELACIONAIS

São utilizados para comparar dois valores de um mesmo tipo. Tais valores são representados por variáveis (e constantes) ou expressões aritméticas (que contém operadores aritméticos). Os operadores relacionais são utilizados para a construção de equações. O resultado de uma operação relacional será sempre um valor *lógico*, isto é, um Verdadeiro/Verdade (V ou **.TRUE.**) ou Falso/Falsidade (F ou **.FALSE.**). Os operadores relacionais adotados pelo Fortran 95 estão descritos na Tabela 3.2. Observe a segunda coluna, que traz a forma obsoleta do FORTRAN 77, mas que ainda é válida.

Tabela 3.2 – Os operadores relacionais disponíveis no Fortran 95.

Operadores Relacionais		
Fortran 95	FORTRAN 77	Função
==	**.EQ.**	igual a
>	**.GT.**	maior que
<	**.LT.**	menor que
>=	**.GE.**	maior ou igual a
<=	**.LE.**	menor ou igual a
/=	**.NE.**	diferente de

Como mencionado anteriormemte, o resultado é um valor lógico. Por exemplo, na relação A + B = C, o resultado será verdadeiro (V) ou falso (F) à medida que o resultado da expressão aritmética A + B seja, respectivamente, igual ou diferente do conteúdo da variável C.

OPERADORES LÓGICOS

São cinco (5) os operadores lógicos para a formação de outras proposições lógicas simples. Os operadores estão descritos na Tabela 3.3, e as possibilidades de agrupamento estão descritos na *Tabela-verdade* (Tabela 3.4) abaixo.

Tabela 3.3 – Os operadores lógicos disponíveis no Fortran 95.

Operadores Lógicos	
Operador	Função
.NOT.	negação
.AND.	conjunção
.OR.	disjunção
.EQV.	equivalência
.NEQV.	não equivalência

Tabela 3.4 – Resultado de operações lógicas - *Tabela-verdade*.

1° valor	Operador	2° valor	Resultado
–	**.NOT.**	V	F
–	**.NOT.**	F	V
F	**.AND.**	F	F
F	**.AND.**	V	F
V	**.AND.**	V	V
F	**.OR.**	F	F
F	**.OR.**	V	V
V	**.OR.**	V	V
V	**.EQV.**	V	V
F	**.EQV.**	F	V
F	**.EQV.**	V	F
V	**.EQV.**	F	F
V	**.NEQV.**	F	V
F	**.NEQV.**	V	V
V	**.NEQV.**	V	F
F	**.NEQV.**	F	F

Na operação com **.AND.** o resultado será verdadeiro somente se ambos os valores relacionados forem verdadeiros (V). No caso do operador **.OR.** o que interessa é que um dos valores relacionados seja verdadeiro (V), para o resultado ser verdadeiro.

Em Fortran 95 os resultados de variáveis lógicas para verdadeiro (V) e falso (F) são, respectivamente, **T** e **F**, de **.TRUE.** e **.FALSE.**. É importante salientar que a atribuição a uma variável lógica é realizada pelos valores **.TRUE.** para verdadeiro e **.FALSE.** para falso. Por exemplo:

```
LOGICAL :: a = .TRUE.
LOGICAL :: b = .FALSE.
```

Digite o programa, compile-o e execute-o para verificar o resultado.

Programa 3.1 – Exemplo com operadores lógicos.

```
1  PROGRAM logica
2  IMPLICIT NONE
3  LOGICAL :: a, b, c=.TRUE.
4     a = 5>7; b = 5<7
5     IF(c) THEN
6        WRITE(*,*)".TRUE. ou verdadeiro"
7     ELSE
8        WRITE(*,*)".FALSE. ou falso"
9     END IF
10    WRITE(*,*)"Resultados de 5>7 e 5<7: ",a,b
11 END PROGRAM logica
```

3.1.3 Hierarquia dos Operadores

A ordem de prioridades entre os operadores utilizada na linguagem Fortran é:

1° Operações que estão no interior de parênteses, iniciando sempre dos parênteses mais internos para os mais externos.

2° Todas as potenciações (**) e depois as radiciações (**SQRT**).

Obs.: a radiciação pode ser transformada numa potenciação, por exemplo, $\sqrt{6} \rightarrow (6)^{1/2}$*, que na linguagem Fortran fica* **6******(1./2.)**. Deve-se tomar muito cuidado nesta operação, porque numa potenciação o expoente é sempre inteiro. Assim, neste caso o expoente para não ser nulo, já que $1/2 = 0$, na aritmética de inteiros, é preciso fazer uma divisão por reais. Quando isto for realizado a operação é trocada por:

$$a^b = e^{b \ln a} \quad \Rightarrow \quad 6^{1/2} = e^{1/2 \ln(6)} ,$$

e o cuidado está na base, que não poderá ser negativa, porque o logaritmo de número negativo não é definido. A potenciação sempre será realizada da direita para a esquerda, $(3^4)^2 = 3^8 = 6561$, mas cuidado com $3^{4^2} = 3^{16} = 43046721$, que é diferente da anterior. Os resultados anteriores não contém pontos decimais porque são valores inteiros.

3° Todas as multiplicações (*) e divisões (/), partindo da esquerda para a direita.

4° Todas as adições (+) e subtrações (−), realizando as operações da esquerda para a direita.

5° Todas as operações relacionais (==, /=, >, <, >=, <=), iniciando **sempre** da esquerda para a direita.

6° Todos os operadores **.NOT.**, iniciando da esquerda para a direita.

7° Todos os operadores **.AND.**, iniciando da esquerda para a direita.

8° Todos os operadores **.OR.** , iniciando da esquerda para a direita.

3.2 ESTRUTURAS COM DECISÃO (OU SELEÇÃO)

Estruturas com Decisão (ou Seleção) são procedimentos que permitem-nos selecionar e executar seções específicas do código (que serão chamadas *blocos*) ou até mesmo desviar para outra parte do código. Elas são as variações da instrução **IF**, mais a estrutura **SELECT CASE**.

3.2.1 Estrutura Condicional Simples – (IF...THEN)

A forma mais comum da instrução de decisão **IF** é a *estrutura condicional simples*. Esta estrutura permite que um bloco de procedimentos seja executado se e somente se uma dada expressão lógica for verdadeira.

A sua forma geral é:

```
[<rótulo>:] IF (<condição lógica>) THEN
    <procedimentos executáveis>
END IF [<rótulo>]
```

em que:

IF: especifica o início da estrutura de decisão **IF...END IF**.

<rótulo>: é uma identificação opcional do bloco de decisão.

(<condição lógica>): determina um teste lógico, que se for verdadeira *então* (**THEN**) executa os **<procedimentos executáveis>**. Se for falsa, *então* (**THEN**) o fluxo de execução é desviado incondicionalmente para a linha seguinte ao **END IF**. A sequência de instruções **IF...THEN** obrigatoriamente é escrita na mesma linha.

END IF: especifica o final do bloco de decisão **<rótulo>**. Como o rótulo é opcional, não é comum utilizá-lo para um único bloco.

INDENTAÇÃO

Observe a indentação utilizada na definição acima e no exemplo a seguir. Estes afastamentos facilitam a visualização dos blocos ou estruturas no programa. Este estilo é adotado ao longo do livro e é uma **boa prática de programação**

No exemplo a seguir uma estrutura de decisão simples testa um número para identificar se é ímpar ou não. Para a solução é necessário utilizar a função intrínseca **MOD**, que fornece o resto da divisão entre números inteiros. As funções intrínsecas ou procedimentos intrínsecos estão descritos na página 31, no Capítulo 1.

Programa 3.2 – Exemplo de estrutura por decisão simples.

```
1 PROGRAM no_impar
2 INTEGER :: a
3    WRITE(*,*)"Digite um numero inteiro: "
4    READ(*,*) a
5    IF (MOD(a,2) == 1) THEN
6       WRITE(*,*)'O numero digitado eh impar'
7    END IF
8 END PROGRAM no_impar
```

Neste caso, se a expressão lógica **(MOD(a,2) == 1)** for verdadeira, uma instrução de saída imprimirá uma frase na tela do monitor indicando que o número digitado é ímpar. Se for falsa, a execução continua na linha seguinte ao **END IF**.

3.2.2 Estrutura Condicional Composta – `(IF...THEN...ELSE IF)`

Na estrutura simples definida acima, o bloco de procedimentos abaixo do **IF** só será executado se a condição lógica for verdadeira. Se a condição de controle for falsa, todos os procedimentos do bloco serão ignorados e a execução do programa seguirá logo após o **END IF**.

Algumas vezes é necessário executar um conjunto de procedimentos se uma dada condição lógica for verdadeira e outro conjunto de procedimentos para outra condição lógica verdadeira ou, até mesmo no caso da expressão lógica ser falsa. A estrutura condicional composta fornece esta possibilidade e a sua forma geral, que utilizaremos nos nossos algoritmos, é:

```
[<rótulo>:] IF (<condição lógica 1>) THEN
      <procedimentos bloco 1>
  ELSE IF (<condição lógica 2>) THEN
      <procedimentos bloco 2>
         . . .
  [ELSE IF (<condição lógica i>) THEN]
      <procedimentos bloco i>
  [ELSE]
      <procedimento padrão>
END IF [<rótulo>]
```

em que:

IF: especifica o início da estrutura de decisão **IF...THEN...ELSE IF**.

<rótulo>: é uma identificação opcional do bloco de decisão.

(<condição lógica>): determina um teste lógico.
Vejamos como trabalha todo o bloco interno ao **IF**:
inicialmente, se a **<condição lógica 1>** for verdadeira *então* (**THEN**) o programa executa os **<procedimentos bloco 1>** e imediatamente após é desviado para a primeira linha após o **END IF**. Se for falsa, a linha seguinte (**ELSE IF (<condição lógica 2>) THEN**) será executada. Nesta linha, novamente um teste lógico é realizado, e se for verdadeiro os **<procedimentos bloco 2>** serão executados e na sequência é desviado para a primeira linha após o **END IF**. Se for falsa, o próximo **ELSE IF** será executado e assim sucessivamente.
Se nenhuma condição lógica anterior for verdadeira, ou por outras palavras, se todas as **i**-ésimas condições lógicas forem falsas o **procedimento padrão** (*default*), logo abaixo o **ELSE**, será executado.

END IF: especifica o final do bloco de decisão **<rótulo>**. Como o rótulo é opcional, não é comum utilizá-lo para um único bloco.

Observe que pelo menos um **ELSE IF** ou um **ELSE** é necessário para ser considerado uma estrutura condicional composta.

No exemplo a seguir uma estrutura testa o discriminante de uma equação quadrática ($b^2 - 4ac$), para cálculo das raízes, com o objetivo de identificar as raízes.

Programa 3.3 – Exemplo de estrutura por múltiplas decisões.

```
1  PROGRAM raizes
2  REAL :: a, b, c
3     WRITE(*,*)"Digite os coeficientes da eq.: "
4     READ(*,*) a, b, c
5     IF ((b**2 - 4.0*a*c) < 0.0) THEN
6         WRITE(*,*)'Raizes complexas'
7         STOP
8       ELSE IF((b**2 - 4.0*a*c) == 0.0) THEN
9         WRITE(*,*) "A solucao tera 2 raizes e iguais"
10      ELSE
11        WRITE(*,*) "A solucao tera 2 raizes desiguais"
12    END IF
13 END PROGRAM raizes
```

Neste caso, se a expressão lógica **(b**2 - 4*a*c < 0.0)** for verdadeira, a execução do programa será cancelada, devido ao **STOP**. Neste caso, não há interesse em calcular as raízes complexas. Se for falsa, a execução continua na linha seguinte ao **STOP** e se **(b**2 - 4*a*c == 0.0)** uma frase será impressa na tela do monitor e, a execução é desviada para a primeira linha após o **END IF**. Se for falsa, a condição *default* (do **ELSE**) será executada e a execução continua na linha seguinte após **END IF**.

A instrução **STOP** é útil quando necessitamos abortar a execução em alguma parte do programa que não seja o final. Neste caso, a execução desviará para o **END PROGRAM**, encerrando o programa.

As estruturas de decisão podem ser "aninhadas", ou seja, podem conter uma dentro da outra, quantas forem necessárias, como por exemplo:

```
ext: IF (...) THEN
        ...
        int1: IF (...) THEN
            ...
            int2: IF (...) THEN
                ...
            END IF int2
            ...
        END IF int1
        ...
     END IF ext
```

A estrutura ou bloco mais interno é identificado pelo rótulo **int2** e o mais externo por **ext**. E, o bloco intermediário é identificado por **int1**. Nos casos de "aninhamentos" de estruturas de decisão é uma **boa prática de programação** o uso dos rótulos, para tornar claro que procedimentos são executados em cada bloco.

3.2.3 A instrução `IF` Lógico

É uma forma alternativa da estrutura condicional **`IF`** e consiste de um único procedimento com estrutura:

```
IF (<expressão lógica>) <procedimento>
```

em que:

`IF`: especifica o início do comando lógico.

`(<expressão lógica>)`: determina um teste lógico, que se o resultado for verdadeiro executa o **`<procedimento>`**. Caso contrário, o fluxo de execução continua na linha seguinte.

Esta instrução é utilizada nas estruturas de repetição sem contador fixo, nos chamados *loops* infinitos, que veremos na página 74. O exemplo abaixo demonstra o uso dessa instrução.

Programa 3.4 – Exemplo de decisão lógica em uma linha.

```
1 PROGRAM if_logico
2 IMPLICIT NONE
3 INTEGER :: a
4    WRITE(*,*)"Digite um numero inteiro: "
5    READ(*,*) a
6    IF(a >= 10) a =  a + 25
7    WRITE(*,*)"O Resultado final eh: ",a
8 END PROGRAM if_logico
```

Neste caso, se for digitado um número maior ou igual a 10, o resultado final será o valor digitado somado de 25. Caso contrário a saída será igual a entrada.

3.2.4 A estrutura de seleção direta (`SELECT CASE...CASE`)

A estrutura **`SELECT CASE`** é outra forma de ramificação numa tomada de decisão. Ela permite o programador selecionar um bloco específico baseado no valor de uma única variável, que poderá ser inteira, caracter ou lógica. A forma geral que adotaremos para representar esta estrutura é:

```
SELECT CASE (<expressão>)
   CASE(<seletor 1>)
      <procedimentos bloco 1>
   CASE(<seletor 2>)
      <procedimentos bloco 2>
         . . .
   CASE(<seletor i>)
      <procedimentos bloco i>
   CASE DEFAULT
      <procedimento padrão>
END SELECT
```

em que:

SELECT CASE: especifica o início da estrutura de seleção direta.

<expressão>: é a expressão que servirá de chave de seleçao. Deverá ser do tipo inteiro ou caracter ou lógico;

CASE: é a instrução que instruirá o código a testar o **<seletor i>**;

<seletor i>: é o valor ou intervalo que será comparado com a **<expressão>**

CASE DEFAULT: é uma instrução optativa. O objetivo é ativá-la quando nenhum dos testes anteriores realizados pela instrução **CASE**, for satisfeito.

O FUNCIONAMENTO:
Se o valor da **expressão** está dentro dos valores incluídos no **seletor 1** então os procedimentos do bloco 1 serão executados e, a seguir o fluxo de execução é desviado para o **END SELECT**. De forma semelhante, se o valor de **expressão** estiver entre os valores do **seletor 2**, os procedimentos do bloco 2 serão executados. A mesma idéia se aplica aos demais outros casos desta estrutura. O **CASE DEFAULT** é opcional, mas se estiver presente, só será executado se todos os *seletor i* anteriores forem falsos. Ainda, se o algoritmo não contiver o **CASE DEFAULT** e mesmo assim todos os **seletor i** anteriores forem falsos, nenhum bloco da estrutura **SELECT CASE** será executado.

O programa a seguir imprime mensagens conforme o valor da temperatura.

Programa 3.5 – Exemplo de estrutura por seleção direta.

```
1  PROGRAM clima_temp
2  INTEGER :: temp_c
3     WRITE(*,*) "Informe a temperatura ambiente, em Celsius"
4     READ(*,*) temp_c
5     SELECT CASE (temp_c)
6       CASE (:-1)
7         WRITE(*,*) "Hoje esta muito frio"
8       CASE (0)
9         WRITE(*,*) "Temperatura de gelo da agua"
10      CASE (1:20)
11        WRITE(*,*) "Hoje esta frio, mas suportavel"
12      CASE (21:25)
13        WRITE(*,*) "Hoje o clima esta agradavel"
14      CASE (26:35)
15        WRITE(*,*) "Hoje está quente"
16      CASE DEFAULT
17        WRITE(*,*) "Hoje o dia esta muito quente"
18    END SELECT
19 END PROGRAM clima_temp
```

O valor de **temp_c** controlará o fluxo do algoritmo. Se a temperatura for menor ou igual a -1 (ou menor que 0), então o primeiro bloco será executado, isto é, sairá na tela do monitor a informação **Hoje esta muito frio**, e o fluxo incondicionalmente é desviado

para **END SELECT**. O bloco é abandonado. Se não for o caso, isto é, a temperatura for igual a zero, então o segundo bloco será executado e não o primeiro, e novamente após ser executado o procedimento do bloco, o fluxo é desviado para o final do bloco. E, assim sucessivamente. Observe que os selecionadores (valores atribuídos para comparação com **temp_c**) não se superpõem–um valor de temperatura só pode aparecer uma e somente uma vez no selecionador.

É importante salientar que os **<seletor i>** podem assumir:

<valor_único>	Executa o bloco se **<valor_único>** = **<expressão>**
:<valor_máximo>	Executa o bloco para todos os valores de **<expressão>** menores ou igual a **<valor_máximo>**
<valor_mínimo>:	Executa o bloco para todos os valores de **<expressão>** maiores ou igual a **<valor_mínimo>**
<valor_mínimo>:<valor_máximo>	Executa o bloco para todos os valores da **<expressão>** entre **<valor_mínimo>** inclusive e **<valor_máximo>** inclusive

3.3 ESTRUTURAS DE REPETIÇÃO (*LOOPS*)

Veremos dois tipos de estruturas de repetição: o **DO...END DO** e o **DO...IF... END DO**, conhecido também por **DO** *infinito*. Em ambos a caracterítisca principal é repetir procedimentos, conforme uma lógica estabelecida.

3.3.1 A Estrutura de repetição DO...END DO

A sua forma geral é:

```
[<rótulo>:] DO <contador> = <valor_i>, <valor_f>, [<incremento>]
     <procedimentos>
END DO [<rótulo>]
```

em que:

DO: especifica o início da repetição dos procedimentos internos ao bloco **DO...END DO**, e inicia a execução dos **<procedimentos>**.

<rótulo>: é uma identificação opcional da estrutura de repetição.

<contador> = <valor_i>, <valor_f>, [<incremento>]: é o controle das repetições pela variável de controle **contador**, sempre do tipo inteiro. Mais:

<valor_i>: é um valor inteiro que determina o início do contador.

<valor_f>: é um valor inteiro que determina o final do contador, e assim o fim do bloco de repetição.

<incremento>: é um número que determina o "salto" na repetição. Por padrão é igual a um, isto é, se for omitido o valor da variável **<contador>** será acrescida de 1 a cada *loop*. Poderá assumir valores negativos, para por exemplo, fazer regedir o **<contador>**, desde que o **<valor_i>** seja maior que o **<valor_f>**.

Importante: estes três valores podem ser substituídos por variável, do tipo inteiro, desde que incializadas antes do **DO**. Por exemplo:

```
INTEGER :: i, x=1, y=200, z=2
   DO i = x,y,z
```

END DO: especifica o final do *loop* **<rótulo>**. Como o rótulo é opcional, não é comum utilizá-lo para um único bloco.

O FUNCIONAMENTO DO CONTROLE DAS REPETIÇÕES:

ao iniciar um **DO** a variável **<contador>** recebe o **<valor_i>**, executa os procedimentos internos e ao encontrar o **END DO** retorna a linha inicial do bloco. Adiciona o **<incremento>** à variável **<contador>** e testa se **<valor_i>** $\geq$ **<valor_f>**. Se verdadeiro executa novamente os procedimentos. Este processo se repete até que **<valor_i>** $\geq$ **<valor_f>** seja falso, condição que desviará a execução para a primeira linha após o **END DO**.

ATENÇÃO COM OS CONTADORES

Se precisar utilizar o valor do **<contador>** depois do bloco **DO**, tenha cuidado, porque o **<contador>** sairá com o último valor que não é igual ao **<valor_f>**, porque foi acrescido do **<incremento>** antes do teste lógico, para verificar se **<valor_i>** $\geq$ **<valor_f>**

3.3.2 A Estrutura de repetição DO...IF...END DO ou DO *infinito*

Até o momento, todos os blocos de repetição (**DO...END DO**) são executados sabendo-se previamente quantas vezes será repetida uma dada tarefa. Mas, algumas vezes é necessário executar uma tarefa no interior de um bloco de repetição sem sabermos quantas vezes a mesma será repetida. Isto é muito comum, por exemplo, nos casos de busca de convergência num processo iterativo. Desta forma, internamente ao bloco de repetição é que será tomada a decisão de abandoná-lo. Quando isto ocorrer a estrutura **DO...IF...END DO** deverá ser usada, com a instrução de controle **EXIT**.

fortran 95

Há uma outra situação na qual, devido a um controle lógico, somente parte dos procedimentos internos ao bloco sejam executados, mas que não abandone até que outro controle lógico seja satisfeito. Neste caso, juntamente com o **EXIT**, usamos a instrução de controle **CYCLE**.

fortran 95

A forma geral deste bloco de repetição é:

```
[<rótulo>:] DO
      [<procedimentos 1>]
   IF (<condição lógica>) <instrução de controle> [<rótulo>]
      [<procedimentos 2>]
END DO [<rótulo>]
```

em que:

DO: especifica o início da repetição *infinita* dos procedimentos internos ao bloco **DO...IF...END DO**. Neste caso, se existirem, os **<procedimentos 1>** serão executados.

<rótulo>: é uma identificação opcional dada a estrutra de repetição.

IF (<condição lógica>): determina um teste lógico, pela *(condição lógica)*, que caso seja verdadeira executa a **<instrução de controle>**. Se for falsa, o fluxo do bloco continua na próxima linha, executando os **<procedimentos 2>**. Não há restrição ao número de testes lógicos, mas que exista pelo menos um, neste caso de saída da estrutura de repetição. E, podem estar localizados em qualquer ponto no interior do **DO...IF...END DO**.

<instrução de controle>: pode ser:

EXIT: determina a saída da estrutra de repetição **<rótulo>**, em que o **<rótulo>** poderá ser de outra estrutura de repetição, sempre, externa a atual. Esta rotulagem é opcional, portanto, não é comum utilizá-la para um único bloco.

Quando esta instrução de controle for executada o fluxo de execução do bloco (e do programa) é transferido para a primeira linha após o **END DO**. Observe que neste caso os **<procedimentos 2>** não serão executados.

CYCLE: determina o retorno ao início da estrutra de repetição **<rótulo>**, em que, novamente, o **<rótulo>** poderá ser de outra estrutura de repetição, sempre, externa a atual.

END DO: especifica o final do *loop*.

A estrutura, como consta no quadro acima, funciona da seguinte maneira, por exemplo, com a instrução de controle **EXIT**:

1. ao entrar no **DO** os **<procedimentos 1>** serão executados, caso existam,

2. a seguir a instrução **IF** será executada, e caso seja falsa os **<procedimentos 2>** serão executados e,

3. na sequência retorna a executar os **<procedimentos 1>** e a instrução **IF**. Este ciclo será executado sucessivamente, até que saia do bloco **DO**, quando a **<condição lógica>** for verdadeira, e neste caso, o último *loop* os **<procedimentos 2>** não serão executados.

Os **<procedimentos 1>** e **<procedimentos 2>** são opcionais e o uso de um ou de outro, ou ainda de ambos, dependerá das tarefas a serem executadas no programa.

Este tipo de estrutura também recebe o nome de *loop infinito* (repetições infinitas), porque se não existir uma condição de saída ou se a **condição lógica** para um **EXIT** nunca for verdadeira ele permanecerá *indefinidamente* neste processo repetitivo. Portanto, deve-se ter muito cuidado no uso dessa estrutura de repetição.

O USO DO `WHILE`

Esta estrutura de repetição pode ser substituída por outra que contenha a instrução **`WHILE`**, como segue:

```
[<rótulo>:] DO WHILE (<condição lógica>)
     [<procedimentos>]
END DO [<rótulo>]
```

Neste caso o **`DO`** estará ativo até que a **`<condição lógica>`** do **`WHILE`** seja falsa.

Vejamos um exemplo do uso de estruturas de repetição. O programa identifica e imprime os primeiros 100 pares no intervalo entre 1 e 1000.

Programa 3.6 – Exemplo de estrutura de repetição.

```
1  PROGRAM numeros_pares
2  IMPLICIT NONE
3  INTEGER :: i,j
4     j=0
5     DO i=1,1000
6        WRITE(*,*)"Contador: ", i
7        IF (MOD(i,2)==0) THEN
8           WRITE (*,*) "Eh um numero par: ", i
9           j=j+1
10          IF (j==100) EXIT
11       END IF
12    END DO
13 END PROGRAM numeros_pares
```

As estruturas de repetição, assim como as de decisão (**`IF`**), também podem ser aninhadas, ou seja, podem conter uma dentro da outra. Por exemplo:

```
ext: DO i=1,20
        ...
        int1: DO
            IF (a > b) EXIT int1
            ...
            int2: DO j=200,1,-1
               ...
            END DO int2
            ...
        END DO int1
        ...
     END DO ext
```

■ ■ ■ ■ ■

É importante que faça os exercícios a seguir. As respostas e comentários a respeito dos exercícios estão no site oficial do livro: *http://www.orengonline.com/fortran95/*.

EXERCÍCIOS

3.1) Criar um programa que entre com um número inteiro e informe se ele é ou não divisível por 5. *Dica: aqui será necessário testar se o resto de uma divisão por 5 é zero ou não. Em Fortran 95 isto feito pela função* **`MOD(a,b)`**, *conforme tabela 1.1, na página 31.*

3.2) Criar um programa, que entre com um número inteiro e informe se ele é divisível por 3 e por 7. *Obs.: aqui será necessário o uso dos operadores lógicos.*

3.3) Criar um programa em Fortran 95, que entre com um número inteiro e informe se ele é par ou ímpar.

3.4) Segundo uma tabela médica, o peso ideal está relacionado com a altura e o sexo. Fazer um programa em Fortran 95 que receba a altura (h) e o sexo de uma pessoa. A seguir calcule e imprima o seu peso ideal, utilizando as seguintes fórmulas: (a) para homens: (71.7 * h) - 58; (b) para mulheres: (62.1 * h) - 44.7 *Dica: não esqueça, a altura é um número real e é necessário saber se é homem ou mulher. Ou seja, podemos solicitar a informação por intermédio da indicação de "f" ou "F", ou "m" ou "M". Revise leitura de caracteres no Capítulo 2.*

3.5) Crie três programas em Fortran 95. O primeiro escreverá todos números de 100 até 1. O segundo escreverá os 100 primeiros pares e o último imprimirá os múltiplos de 5, no intervalo de 1 até 500.

3.6) Elabore um programa em Fortran 95, que escreva a soma dos números entre 25 e 200. *Dica: neste caso, a soma é realizada da seguinte forma:* **`soma1 = soma1 + 1`**. *Não esqueça de informar antes de começar o loop da soma (*`DO...END DO`*) o valor inicial de* **`soma1`**.

3.7) Crie um programa em Fortran 95, que escreva a soma dos números pares entre 25 e 200. *Dica: se baseie no exercício 3.6.*

3.8) Criar um programa em Fortran 95, que escreva o fatorial de um dado número. Importante: existe limite (de máquina) para o tamanho do número? *Dica: não esqueça que não é definido fatorial de número negativo e que 0!=1.*

3.9) Criar um programa que encontre o n-ésimo termo da série de Fibonacci. A série de Fibonacci é dada por: fib(n) = fib(n − 1) + fib(n − 2) para n > 1. Para n = 0 e n = 1, o valor é por definição fib(0) = 0 e fib(1) = 1.

3.10) Criar dois programas. O primeiro lerá 15 números e escreverá quantos números maiores que 30 foram digitados. O segundo receberá 20 números e escreverá a soma dos positivos e o total de números negativos.

3.11) Criar um programa que entre com um nome e escreva-o tantas vezes quantos forem seus caracteres.

3.12) Criar um programa que entre com uma palavra e escreva conforme o exemplo a seguir: Palavra: AMOR. A saída será AMOR, AMO, AM, A

3.13) Criar um programa que entre com dois números e escreva todos os números no intervalo fechado, do menor para o maior.

3.14) Criar um programa para ler o número de termos da série (N) e escrever o valor de H, sendo: $H = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + ... + \frac{1}{N}$.

3.15) Um restaurante faz uma promoção semanal de descontos para clientes de acordo com as iniciais do nome da pessoa. Projete um programa, em Fortran 95, que leia o primeiro nome do cliente, o valor de sua conta e se o nome iniciair com as letras A, D, M ou S, dar um desconto de 30 %. Para o cliente cujo nome não se inicia por nenhuma dessas letras, exibir a mensagem "Que pena. Nesta semana o desconto não é para seu nome, mas continue nos prestigiando que sua vez chegará".

3.16) Faça um programa que leia um número inteiro de 3 algarismos e imprima uma informação se o algarismo da casa das centenas é par ou ímpar.

3.17) Crie um programa para ler um número inteiro de 4 algarismos e imprimir se é ou não múltiplo de quatro o número formado pelos algarismos que estão nas casas da unidade e da centena.

3.18) Desenvolva um programa que leia o ano de nascimento de uma pessoa e o ano atual. A seguir imprima a idade da pessoa. *Não se esqueça de verificar se o ano de nascimento é um ano válido.*

3.19) Construir um programa em Fortran 95 que calcule a média aritmética de vários valores inteiros positivos, lidos externamente. O final da leitura acontecerá quando for lido um valor negativo. Mostrar o resultado ao final.

3.20) Escreva um programa em Fortran 95 que calcule a média dos números digitados pelo usuário, se eles forem pares. Termine a leitura se o usuário digitar 0 (zero). Mostrar o resultado ao final.

3.21) Desenvolva um programa em Fortran 95 que some os números fornecidos pelo usuário até que o número lido seja igual a zero e mostre a soma.

3.22) Chico tem 1,5 metros e cresce 2 centímetros por ano, enquanto Zé tem 1,1 metros e cresce 3 centímetros por ano. Faça um programa que calcule e mostre quantos anos serão necessários para que Zé seja maior que Chico.

3.23) Faça um programa em Fortran 95 que leia um número *n* que indicará quantos valores devem ser lidos a seguir e, para cada número lido, mostre uma tabela contendo o valor lido e o cubo do número somado a ele mesmo.

3.24) Elabore um programa em Fortran 90/95 que encontre um número real que mais se aproxima da raiz quadrada de um número fornecido pelo usuário. Indique também ao final quantos *loops* foram necessários. (*Obs.: não basta só calcular a raiz quadrada direta pela função* **`SQRT`**, *mas sim fazer um loop com a multiplicação de dois números iguais e que satisfaça ao teste de precisão entre o valor calculdado e o valor (exato) da raiz*.)

CAPÍTULO 4

AS VARIÁVEIS COMPOSTAS: Vetores e Matrizes (*Arranjos*)

Neste capítulo você encontrará:

Saudade (09/09/2004 - 11:00)

Eternidade: tanto faz
dia após dia se faz
Saudade: esta faz
o tempo ser sagaz

Gilberto Orengo
1961 –

4.1 INTRODUÇÃO

Até o momento escrevemos programas cujas variáveis armazenam um único valor, chamadas de *variáveis simples*. Por exemplo,

```
INTEGER :: a
REAL :: total
   ...
   a = 34
   total = 3.78
```

Existem casos que precisamos armazenar diferentes valores para uma mesma variável. Estas variáveis são conhecidas como ***arrays***, em inglês, ou simplesmente de **variáveis compostas**. As variáveis compostas mais conhecidas são os *vetores* e *matrizes*. Um vetor pode ser considerado uma matriz sob certas condições, e neste caso é chamado de *matriz coluna* ou *matriz linha*. A seguir veremos cada um em detalhes, iniciando pelos vetores.

4.2

OS VETORES

Os vetores são variáveis compostas unidimensionais. Por exemplo, seja o vetor $\mathbf{R} = 5\hat{\mathbf{i}} - 1\hat{\mathbf{j}} + 7\hat{\mathbf{k}}$, que tem seus componentes dados por:

$$R = \{5, -1, 7\}$$

em que $r_x = 5$; $r_y = -1$ e $r_z = 7$. De uma forma geral, um vetor $\mathbf{A}$ terá componentes descritos da seguinte maneira:

$$A = \{a_1, a_2, a_3, \cdots, a_n\}$$

O termo *variável composta unidimensional* é reconhecido pelo o **único índice** necessário para descrever a variável, ou seja, no exemplo acima a variável A tem um índice (n) para indicar a posição no conjunto de valores atribuídos a ela. Os valores assumido pelo índice n determinará a quantidade de elementos do vetor.

Na memória, a variável R armazenará seus componentes representada esquematicamente como segue, na Figura 4.1.

	1	2	3	4	5	6	7	8	9
		R					nome		
		5	-1	7			João	Maria	

Figura 4.1 – Representação na memória das variáveis *R* e *nome*.

Na representação acima foi acrescentado um exemplo de um vetor que armazena caracteres. O vetor *nome* armazena os caracteres (ou *string*) *João* e *Maria*. Já a variável R armazena valores inteiros. Um vetor só armazenará valores de um mesmo tipo, isto é, nunca poderemos armazenar num vetor, por exemplo, valores reais e inteiros ao mesmo tempo. Teremos que criar uma variável que armazene os valores reais e outra para os inteiros.

4.2.1 A declaração de um vetor

A declaração de um vetor, ou de uma variável composta unidimensional, segue a seguinte estrutura, conhecida como *arrays de forma-explícita*:

```
<tipo> [([KIND=]<par_rep>)], [<atrib>] [::] <var>(<forma>)
```

ou

```
<tipo> [([KIND=]<par_rep>)], DIMENSION(<forma>),[<atrib>][::] <var>
```

em que:

<tipo>: é um dos tipos de dados estudados anteriormente e podem ser: **INTEGER**, **REAL**, **COMPLEX**, **LOGICAL** e **CHARACTER**, respectivamente representando, inteiros, reais, complexos, lógicos e caracteres.

([KIND=]<par_rep>): em Fortran 95, cada um dos cinco tipos intrínsecos anteriores possui um valor inteiro não negativo denominado *parâmetro de representação* do tipo de dado. Este parâmetro é o valor correspondente em bytes disponibilizados para sua representação, como vimos em detalhes para os inteiros e reais, a partir da página 23.
Na normatização da linguagem Fortran 90/95 ficou estabelecido um padrão: qualquer processador deve suportar pelo menos dois parâmetros de representação (o **KIND**) para os tipos de dados **REAL** e **COMPLEX** e pelo menos um parâmetro para os tipos de dados **INTEGER**, **CHARACTER** e **LOGICAL**.

A tabela B.1, no Apêndice B, contém todos os tipos de dados, seus parâmetros de representação e intervalos de armazenamento, baseado no compilador G95.

DIMENSION(<forma>): indica a forma de uma *array*, em que **(<forma>)** é unidimensional, ou seja, com um único valor indica que é um vetor, cujo valor informa o número de elementos do vetor.

<atrib>: são informações adicionais a respeito do tipo de dado e podem ser, entre outros:

PARAMETER: indica que o dado será constante ao longo de todo o programa, sem possibilidades de alterá-lo,

ALLOCATABLE: é usado juntamente com o **DIMENSION** e indica que o tamanho da *array* será informado ao longo do programa e não no momento da declaração da variável. É a chamada *alocação dinâmica de memória* e será estudada no Capítulo 5.

Obs.:
– Podem ser declarados mais de um atributo para a mesma variável, os quais serão separados por vírgula.

::: os uso dos dois "dois pontos" é opcional, mas será obrigatório caso deseja-se inicializar a variável no momento da sua declaração.

<var>: são os nomes das variáveis que armazenarão os dados, separados por vírgulas. O nome de uma variável poderá ter até 31 caracteres e iniciará sempre com uma letra do alfabeto, jamais com um algarismo, que poderá ser utilizado a partir da segunda posição, assim como o *underscore* (_). Não conterá caracter especial (" , ( ){ }[ ]!˜ . : @ # $ % ˆ & *) em qualquer posição do nome, bem como letras acentuadas ou cedilhadas.

São exemplos de declaração de vetores

```
INTEGER :: A(5), R(3)
REAL :: h(10)
CHARACTER(LEN=5) :: nome(2)
```

ou por:

```
INTEGER, DIMENSION(5) :: A
INTEGER, DIMENSION(3) :: R
REAL, DIMENSION(10) :: h
CHARACTER(LEN=5), DIMENSION(2) :: nome
```

Na primeira representação, a variável **A** armazenará 5 valores inteiros, ou com outras palavras, o vetor **A** possui 5 elementos. Desta forma, o vetor **h** poderá armazenar 10 valores do tipo real. O termo "poderá" está bem colocado, pois a variável declarada como vetor terá um limite que armazenará, podendo não ser completado. Por exemplo, no caso da variável **A** se informarmos 4 valores, o vetor passará a ter 4 elementos e não 5. Neste caso, a memória será ocupada de forma não otimizada, pois requisitou-se que reservasse um espaço para 5 elementos e estamos só ocupando 4. Opostamente, se informarmos 6 elementos no lugar de 5, estaremos cometendo um erro, pois a variável irá armazenar somente nos espaços permitidos na declaração.

Nos nossos exemplos anteriores, na pág. 80 (Figura 4.1), a variável **R** armazenará 3 valores (elementos) inteiros. A variável **nome** armazenará dois valores, de tamanho igual a 5 caracteres cada. O argumento **LEN** indica o tamanho de cada variável do tipo caracter. Se este argumento for omitido, será assumido o valor *default* igual a 1. No exemplo abaixo,

```
CHARACTER(LEN=1), DIMENSION(3) :: nome1
```

é equivalente a

```
CHARACTER, DIMENSION(3) :: nome1
```

e a variável composta (*array*) armazenará 3 valores de tamanho igual a 1 cada.

Na segunda representação, é inserido o argumento **DIMENSION**, no qual, entre parênteses, indicamos a quantidade de elementos do vetor, ou seja, a sua dimensão.

BOA PRÁTICA DE PROGRAMAÇÃO

Adotar a última representação para a declaração de variáveis compostas (com DIMENSION), pois torna-se mais claro para o programador e para o "compilador".

4.2.2 Preenchendo com valores os elementos de um vetor

O procedimento para preencher uma variável composta é diferente da simples, pois temos que informar todos os elementos de uma variável com o mesmo nome. Existem diferentes formas de fazer esta tarefa, veremos algumas.

Preenchendo todos os elementos de uma única vez

Para preencher uma variável composta, por exemplo **A**, de uma única vez o procedimento é:

```
A =10
```

ou ainda, na própria declaração da variável:

```
INTEGER, DIMENSION(5) :: A = 10
```

Assim, todos os elementos da variável **A**, do tipo real, serão preenchidos pelo valor igual a 10.

O uso de uma estrutura de repetição (*loops*), e da instrução `READ`

Este procedimento é executado como segue:

```
DO i=1,5
   READ(*,*)A(i)
END DO
```

Neste caso, usamos a estrutura de repetição **`DO ... END DO`**, associado com a instrução de leitura, via teclado, **`READ(*,*)`** para preencher os valores da variável composta **`A`**. O procedimento é o seguinte: no *loop*, para **`i=1`** o primeiro elemento **`A(1)`** será lido e armazenado; na sequência para **`i=2`** o segundo elemento **`A(2)`** e assim, sucessivamente, até o último elemento **`A(5)`** seja lido e armazenado.

O uso de uma estrutura de repetição, em conjunto com uma expressão

```
x = 0
DO i = 1,5
   A(i) = x + i**2
   x = x + 3
END DO
```

Assim, os valores dos elementos da variável serão: $A(1) = 1$, $A(2) = 7$, $A(3) = 15$, $A(4) = 25$ e $A(5) = 37$. A função utilizada foi $x+i^2$, em que i é o próprio contador, mas não é necessário. A função poderia ser simplesmente $x + x^5$, por exemplo.

4.2.3 A manipulação dos elementos de um vetor

Podemos manipular os elementos como nos exemplos abaixo:

```
y = A(3)
b = A(1) + A(5)
```

Nestes casos, tomando os valores atribuídos a variável composta **`A`**, os resultados serão **`y=15`** e **`b=38`**. As variáveis **`y`** e **`b`** são do tipo simples. Na última linha temos a soma de elementos do vetor.

Podemos multiplicar cada elemento de um vetor por um escalar como segue:

```
C = 5*A
```

A variável **`C`** é do mesmo tipo da variável **`A`**, ou seja, composta. Este procedimento também pode ser adotado para a divisão.

A soma ou subtração de vetores:

O Fortran 95 permite a operação em bloco de soma ou subtração de vetores, desde que os vetores sejam conformes, isto é, tenham a mesma dimensão (forma) ou mesmo número de elementos. Por exemplo:

```
INTEGER, DIMENSION(5) :: A = 10
INTEGER, DIMENSION(5) :: B = 2
INTEGER, DIMENSION(5) :: C
  ...
C = A + B
```

O resultado é um vetor com os elementos todos iguais a 12, já que todos os vetores tem dimensão igual a 5.

A multiplicação de vetores:

A multiplicação de vetores é possível desde que os vetores sejam conformes, isto é, tenham a mesma dimensão. Este procedimento é um produto interno (ou produto escalar) e em Fortran 95 é realizado com a instrução **DOT_PRODUCT(vetor1,vetor2)**, como segue:

```
INTEGER, DIMENSION(5) :: A=10, B=2
INTEGER :: C
C = DOT_PRODUCT(A,B)
```

que é equivalente a:

```
C = A(1)*B(1) + A(2)*B(2) + ... + A(5)*B(5)
```

ou a:

```
C = SUM(A*B)
```

em que, neste último, foi utilizado a instrução **SUM** para somar os elementos do produto de dois vetores. O resultado final é simplesmente um número, e neste exemplo o resultado é 100.

4.3 AS MATRIZES

As matrizes são variáveis compostas bidimensionais. Por exemplo, seja a matriz:

$$A = \begin{pmatrix} -2 & 3 \\ 10 & 6 \end{pmatrix}$$

em que $a_{11} = -2$; $a_{12} = 3$; $a_{21} = 10$ e $a_{22} = 6$, são seus componentes. De uma forma geral, a matriz $A_{2\times2}$, terá componentes descritos da seguinte maneira:

$$A = \{a_{11}, a_{12}, a_{21}, a_{22}\}$$

e neste caso disposta linhas por colunas.

Na memória, a variável A armazenará seus componentes representada esquematicamente como segue, na Figura 4.2, dispondo linhas por colunas.

A matriz A, armazena valores inteiros, ordenados por linha. Assim, na primeira linha temos os elementos -2 e 3. Perceba que desta forma, os elementos de uma matriz são armazenados na memória em uma sequência de linhas. Podemos efetuar o ordenamento por

	1	2	3	4	5	6	
		A					
		$A(1,1)$	$A(1,2)$	$A(2,1)$	$A(2,2)$		
		-2	3	10	6		

Figura 4.2 – Representação na memória da variável composta A.

coluna. A diferença no ordenamento por linha ou coluna está no momento da montagem da matriz, ou seja, na informação passada para a memória. Isto será visto logo a seguir.

Uma matriz, assim como um vetor, só armazenará valores de um mesmo tipo, isto é, nunca poderemos armazenar numa matriz valores reais e inteiros ao mesmo tempo. Teremos que criar uma variável que armazene os valores reais e outra para os inteiros.

4.3.1 A declaração de uma matriz

A declaração de uma matriz, ou de uma variável composta bidimensional, segue a seguinte estrutura:

```
1 INTEGER :: A(2,2)
2 REAL :: h(4,3)
3 CHARACTER :: nomes(2,5)
```

ou por:

```
1 INTEGER, DIMENSION(2,2) :: A
2 REAL, DIMENSION(4,3) :: h
3 CHARACTER, DIMENSION(2,5) :: nomes
```

A variável **A** terá 2 linhas por 2 colunas, isto é, a primeira informação entre parênteses se refere ao número de linhas, enquanto que a segunda se refere ao número de colunas. Assim, a matriz A armazenará 4 valores inteiros. Este resultado vem da multiplicação do número de linhas pelo número de colunas, ou seja, $2 \times 2 = 4$. A variável **h** armazenará 12 valores, em 4 linhas e 3 colunas; e a variável **nomes** terá 10 entradas, em 2 linhas e 5 colunas.

4.3.2 Preenchendo com valores os elementos de uma matriz

O procedimento para preencher uma variável composta é semelhante a de um vetor. Assim, como no caso dos vetores, existem diferentes formas de realizar esta tarefa.

O uso de uma estrutura de repetição

```
1 DO i = 1, 2
2    DO j = 1, 2
3       READ(*,*)A(i,j)
4    END DO
5 END DO
```

Neste caso, usamos a estrutura de repetição **DO ... END DO**, associado com a instrução de leitura **READ** para preencher os valores da variável composta **A**.

É importante ressaltar que o preenchimento da matriz se fez pelas linhas. Isto é, para cada valor de de **i** o valor da variável de controle **j**, que representa as colunas, varia de 1 até 2 para cada repetição externa. Se quisermos preencher por colunas, basta trocar a linha 1 pela 2. Assim, estaremos fixando o **j** e variando o **i**, para cada repetição externa.

O uso de uma estrutura de repetição, em conjunto com uma expressão

Da mesma forma utilizada para os vetores, temos:

```
1  x = 0
2  DO i = 1,2
3     DO j = 1,2
4        A(i,j) =  x + j
5        x = x + 3
6     END DO
7  END DO
```

Neste caso, os valores da variável serão: $A(1,1) = 1$, $A(1,2) = 5$, $A(2,1) = 7$ e $A(2,2) = 11$. Novamente aqui foi utilizado o contador interno **j** na expressão (função) matemática. Esta não é necessariamente uma prática de programação, poderíamos trocar por uma outra expressão.

4.3.3 A manipulação dos elementos de uma matriz

Elemento a elemento:

Podemos manipular os elementos como segue, atribuindo-as a variávies simples. Nos exemplos abaixo os valores dos elementos foram retirados da matriz anterior $A_{2\times2}$:

```
y = A(2,1)
b = A(1,1) + A(2,2)
```

Nestes casos, os resultados para as variáveis simples são, respectivamente, $y = 7$ e $b = 12$.

A soma ou subtração de matrizes:

A soma ou subtração de matrizes é possível desde que as matrizes sejam conformes, isto é, que tenham mesma dimensão. Por exemplo:

```
1  INTEGER, DIMENSION(5,3) :: A = 10
2  INTEGER, DIMENSION(5,3) :: B = 12
3  INTEGER, DIMENSION(5,3) :: C
4          ⋮
5  C = A + B
```

O resultado é uma matriz com os elementos todos iguais a 22. A soma é realizada elemento a elemento, conforme é apresentado abaixo.

$$\begin{pmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{pmatrix}_{5\times3} + \begin{pmatrix} 12 & 12 & 12 \\ 12 & 12 & 12 \\ 12 & 12 & 12 \\ 12 & 12 & 12 \\ 12 & 12 & 12 \end{pmatrix}_{5\times3} = \begin{pmatrix} 10+12 & 10+12 & 10+12 \\ 10+12 & 10+12 & 10+12 \\ 10+12 & 10+12 & 10+12 \\ 10+12 & 10+12 & 10+12 \\ 10+12 & 10+12 & 10+12 \end{pmatrix}_{5\times3} = \begin{pmatrix} 22 & 22 & 22 \\ 22 & 22 & 22 \\ 22 & 22 & 22 \\ 22 & 22 & 22 \\ 22 & 22 & 22 \end{pmatrix}_{5\times3}$$

Este procedimento pode ser usado para multiplicarmos ou dividirmos elemento a elemento de matrizes conformes.

A multiplicação de matrizes:

A multiplicação de matrizes é possível desde que os matrizes sejam conformes, isto é, que o número de colunas da primeira matriz seja igual ao número de linhas da segunda matriz. Suponha duas matrizes **A(x,y)** e **B(m,n)**. Assim:

A(x,y) $\times$ **B(m,n)** só é possível se e somente se **y = m**. A matriz resultante será do tipo **x,n**.

B(m,n) $\times$ **A(x,y)** só é possível se e somente se **n = x**. A matriz resultante será do tipo **m,y**.

Isto é, com as matrizes $A_{4\times3}$ e $B_{3\times5}$, a matriz resultante será $C_{4\times5}$. Por exemplo:

```
1  INTEGER, DIMENSION(4,3) :: A = 10
2  INTEGER, DIMENSION(3,5) :: B = 2
3  INTEGER, DIMENSION(4,5) :: C
```

Matematicamente produzirá:

$$\begin{pmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{pmatrix}_{4\times3} \times \begin{pmatrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{pmatrix}_{3\times5} = \begin{pmatrix} 60 & 60 & 60 & 60 & 60 \\ 60 & 60 & 60 & 60 & 60 \\ 60 & 60 & 60 & 60 & 60 \\ 60 & 60 & 60 & 60 & 60 \end{pmatrix}_{4\times5}$$

A multiplicação em Fortran pode ser realizada por pelo menos maneiras. Especialmente em Fortran 90/95, podemos utilizar a instrução **MATMUL(matriz1,matriz2)**, como segue:

```
C = MATMUL(A,B)
```

que é equivalente a:

```
C(1,1) = A(1,1)*B(1,1) + A(1,2)*B(2,1) + A(1,3)*B(3,1)
   ⋮         ⋮
C(4,5) = A(4,1)*B(1,5) + A(4,2)*B(2,5) + A(4,3)*B(3,5)
```

ou a:

```
DO i = 1,4
   DO k = 1,5
      C(i,k)=0
      DO j = 1,3
         C(i,k) = C(i,k) + A(i,j)*B(j,k)
      END DO
   END DO
END DO
```

em que, neste último, foi utilizado uma estrutura de repetição **DO ... END DO** para multiplicar duas matrizes. O último procedimento é necessário nos programas escritos em no FORTRAN 77. No Fortran 90/95 este procedimento foi incorporado de forma intrínsica. Isto é, quando a instrução **MATMUL** é utilizada a sequência de *loops* anteriores é executada. Observe, atentamente, que os índices da matriz resultante do produto são para linhas e colunas, respectivamente, iguais a **i** e **k** e, o índice **j** é igual ao número de colunas da primeira matriz ou igual ao número de linhas da segunda matriz.

Em resumo: A multiplicação de matrizes é definida somente para duas matrizes que o número de colunas da primeira matriz é igual ao número de linhas da segunda matriz. Assim, se uma matriz A é uma matriz $M \times N$ e uma matriz B é uma matriz $N \times L$, então o produto $C = A \times B$ é uma matriz $M \times L$ cujos elementos são dados pela equação

$$C(i,k) = \sum_{j=1}^{N} A(i,j) * B(j,k) = A(i,1) * B(1,k) + \cdots + A(i,N) * B(N,k)$$

com $i = 1, \cdots, M$; $k = 1, \cdots, L$ e $j = 1, \cdots, N$.

IMPORTANTE .....

Digite o código abaixo e execute-o.

```
PROGRAM testabug
IMPLICIT NONE
REAL, DIMENSION(2,2) :: a=RESHAPE((/2., 3., 4., 5./) ,(/2,2/))
INTEGER :: i,j
    DO i=1,2
        WRITE(*,*)(a(i,j),j=1,2)
    END DO
    a(1,3)=6.0; a(2,3)=7.0; a(3,3)=10.0; a(3,1)=8.0; a(3,2)=9.0
    WRITE(*,*)""
    DO i=1,2
        WRITE(*,*)(a(i,j),j=1,2)
    END DO
    WRITE(*,*)""
    DO i=1,3
        WRITE(*,*)(a(i,j),j=1,3)
    END DO
END PROGRAM testabug
```

Executou? O resultado é inesperado, não!? Pois bem, uma vez definida a dimensão da array, o compilador Fortran mapeia a memória (para arrays) através da seguinte expressão:

$$\text{POSIÇÃO DO ELEMENTO DA ARRAY NA MEMÓRIA} = \lambda(j-1) + (i-1) \qquad (4.1)$$

onde, λ é o número de linhas declarado (no `DIMENSION`), i e j são as linhas e colunas da array, i.e., $a(i,j)$. Usando a Eq.(4.1), mostre o que ocorreu[N1]. O compilador Fortran posiciona os elementos de uma array na memória por coluna, ou seja, primeiro a coluna 1, depois a coluna 2 e assim, sucessivamente.

FALAR SOBRE O TEMPO DE MÁQUINA MANIPULANDO ARRANJOS POR LINHA OU POR COLUNA ....

■ ■ ■ ■ ■

A verificação de aprendizado é pela prática com exercícios e com a solução de problemas. É altamente recomendável que faça os exercícios a seguir, bem como os disponíveis no

[N1]**NOTA**: este tipo de erro pode ser chamado de *"estouro de índice"* para arranjos ou *"estouro de área de armazenamento"*. Este problema também ocorre no compilador C, e a Eq.(4.1) continua sendo válida trocando-se i por j e, o λ agora representa o número de colunas declarado. Este é um dos problemas dos programas feitos para o *rWindows*. Eles não possuem um bom controle de erro – por isso aquela tela *"Erro de Proteção Geral"* – Argh!!!. Este tipo de erro não ocorre, por exemplo, na linguagem *Java*.

site *http://www.orengonline.com/fortran95/*. Após este treinamento estará apto a resolver problemas.

As respostas e comentários a respeito dos exercícios encontram-se no referido endereço eletrônico, em *exercícios*. E, para verificar se está pronto para resolver problemas, um conjunto deles está disponível em *problemas*, também no site acima indicado.

É importante lembrar que este livro é introdutório e, portanto, não contém todas as capacidades mais avançadas das instruções Fortran. Para uma abordagem neste nível aconselha-se os livros listados nas referências biblioigráficas.

EXERCÍCIOS

4.1) Escreva um programa em Fortran 95 que leia 1 vetor de 15 elementos inteiros cada. A seguir mostre o terceiro elemento do vetor e imprima o resultado da multiplicação do quinto elemento pelo décimo elemento.

4.2) Escreva um programa em Fortran 95 que leia 2 vetores de 10 elementos inteiros cada. Criar um terceiro vetor que seja a união dos dois primeiros. Mostrar o vetor resultante.

4.3) Escreva um programa em Fortran 95 que leia 2 vetores de 8 elementos inteiros cada e, a seguir, efetue a multiplicação dos vetores e mostre o resultado.

4.4) Criar um programa em Fortran 90 que leia um conjunto de 30 valores e os coloque em 2 vetores conforme forem pares ou ímpares. O tamanho de cada vetor é de 5 posições. Se algum vetor estiver cheio, escrevê-lo. Terminada a leitura, escrever o conteúdo dos dois vetores. Cada vetor pode ser preenchido tantas vezes quantas forem necessárias. *Obs.: o **zero** não é classificado nem como número par e nem como ímpar.*

4.5) Fazer um programa em Fortran 90 que leia um conjunto de 10 valores inteiros, armazene-os em um vetor e escreva-os ao contrário da ordem de leitura. A saída dos dados deve ser num arquivo com o nome *saida.txt*.

4.6) Criar um programa em Fortran 90 que entre com 4 nomes e imprima uma listagem contendo todos os nomes. Considere que cada nome poderá ter até 10 caracteres. A saída dos dados deve ser num arquivo com o nome *nomes.txt*.

4.7) Criar um programa em Fortran 90 que leia 6 números reais e os ordene em ordem decrescente (isto é, do maior para o menor valor). ***Este dará trabalho – É difícil**. Dica: monte um conjunto de 5 elementos e pense numa estratégia "na ponta do lápis" antes de se aventurar no programa. !!!!*

4.8) Repita o exercício 4.7, mas lendo os 6 valores de entrada de um arquivo de nome *dados1.txt*. *Dica: importante, não esqueça de criar o arquivo e digitar nele os 6 valores reais*.

4.9) Uma empresa deseja aumentar seus preços em 20%. Fazer um programa em Fortran 90/95 que leia o código e o preço de custo de cada produto e calcule o novo preço; calcule também a média dos preços com e sem aumento; e mostre o código e o preço novo de cada produto e, no final, as médias. A entrada de dados deve terminar quando for lido um código de produto nulo.

4.10) Elaborar um programa em Fortran 90/95 que leia um conjunto de 8 valores e os escreva em um vetor. A seguir, separe os valores que são pares e ao final indique quantos são pares e ímpares.

4.11) Escreva um programa em Fortran 90/95 que leia um conjunto de 9 valores reais e os escreva em uma matriz 3×3. A seguir, divida cada elemento da diagonal da matriz por 2.0 e escreva o resultado.

CAPÍTULO 5

A Alocação Dinâmica de Memória (ALLOCATABLE)

Vimos no primeiro capítulo como declarar variáveis. Aqui nos deteremos um pouco nas arrays[N1], ou matrizes. Uma array é um grupo de variáveis ou constantes, todas do mesmo tipo, que são referidas por um único nome. Os valores no grupo ocupam localizações consecutivas na memória do computador. Um valor individual dentro da array é chamado de elemento da array e, sua identificação ocorre pelo nome da array juntamente com um subscrito, que aponta para sua posição dentro da array. Por exemplo, seja uma array de 3 elementos, cujo nome é `hoje`, teremos como seus elementos `hoje(1)`, `hoje(2)` e `hoje(3)`. A sua declaração será:

```
REAL(KIND=8), DIMENSION(3) :: hoje
```

ou

```
REAL(KIND=8) :: hoje(3)
```

Isto é, a array `hoje` tem 3 elementos e cada elemento é do tipo real de precisão dupla, conforme o `KIND=8`. Ou, a array poderia ser de valores inteiros:

```
INTEGER(KIND=4), DIMENSION(3) :: hoje
```

Assim, quando queremos nos referir a um elemento da array, fazemos `hoje(3)`, que representa um dado valor numérico. As arrays acima são do tipo unidimensional ou *rank-1*. As arrays bidimensionais ou *rank-2* são, por exemplo:

```
REAL(KIND=8), DIMENSION(4,4) :: ontem
```

E nos referimos a um elemento deste tipo de array da mesma forma que o unidimensional (só que com 2 subscritos), p.ex., `ontem(1,2)`. Existem arrays de dimensões superiores, caso seja necessário. Podemos também ter arrays de caracteres, no lugar de números:

[N1]Será usado *array(s)* e não matriz(es), por ser de uso corrente no meio computacional, e portanto, mais específico. Até mesmo para não confundirmos com matrizes da Matemática.

```
CHARACTER(len=20), DIMENSION(50) :: nomes
```

Isto é, cada elemento desta array deverá ter até 20 caracteres, e endereçado como `nomes(1)`, `nomes(2)`, até `nomes(50)`.

Mas, o que as arrays acima têm em comum? O tamanho de cada array foi declarado no início do programa. Este tipo de declaração de array é chamado de **alocação estática de memória**, porque o tamanho de cada array deve ser grande o suficiente para conter o maior valor do problema que o programa irá resolver. Isto pode trazer sérias limitações. Se declararmos uma array, sem saber ao certo seu futuro tamanho, poderemos estar sobrecarregando a memória do computador, caso venhamos a usar, por exemplo só 20 ou 30% da memória alocada para a array. Com isso tornaremos a execução mais lenta ou até mesmo sem memória suficiente para executar o programa. No outro extremo está o caso de dimensionarmos a array abaixo do que ela necessitará de alocação de memória. Desta forma, o programa não poderá resolver problemas maiores. Então, como o programador resolverá este problema? Se o mesmo possuir o programa fonte[N2], poderá alterar a declaração e recompilá-lo. Mas, isto nem sempre é possível. E, como faremos com os programas proprietários?

A melhor solução é projetar o programa com **alocação dinâmica de memória**. O tamanho da array será dinamicamente alocada quando for necessário e no tamanho exato. Assim, otimizamos e controlamos melhor o uso da memória e, podemos executar problemas tanto com arrays grandes, quanto pequenas.

5.1 O ATRIBUTO ALLOCATABLE E AS DECLARAÇÕES ALLOCATE E DEALLOCATE

No Fortran 90/95, uma array alocada dinamicamente é declarada com o atributo `ALLOCATABLE` e alocada no programa através da declaração `ALLOCATE`. Quando não precisamos mais da array, a desalocamos da memória através da declaração `DEALLOCATE`.

A estrutura de uma declaração típica de array alocada dinamicamente é:

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

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

Observe que os dois pontos (`:`) são usados no lugar das declarações estáticas, pois ainda não sabemos o tamanho da array. O *rank* da array é declarado, mas não o seu tamanho.

Quando o programa é executado, o tamanho da array será especificado pela declaração `ALLOCATE`. A forma desta declaração é

```
ALLOCATE(lista das variáveis a serem alocadas, STAT=nome do status)
```

[N2] **Open Source**: é uma boa prática abrirmos o código fonte de nossos programas, através da licença GPL–General Public License[13]. Eles se tornarão mais eficientes, pois outros programadores poderão fazer alterações e nos avisar das mesmas.

Um exemplo:

```
ALLOCATE(ontem(100,0:10), STATUS=info)
```

Este procedimento aloca uma array de 100 × 11, quando for necessário. O `STATUS=info` é opcional. Se estiver presente, ele retornará um inteiro. Será 0 para sucesso na alocação ou número positivo (valor que dependerá do compilador) para falha na alocação. É uma boa prática de programação usar o `STATUS`, pois caso esteja ausente e a alocação falhar, p.ex., por falta de memória ou por outro erro qualquer (como nome errado de variável), a execução do programa será abortada. O seu uso é feito através de um controle de fluxo, tipo `IF`. Para o caso acima, temos:

```
1  IF (info == 0) THEN
2     amanha = ontem*10
3  ELSE
4     WRITE(*,*) 'Erro na Alocacao de Memoria. Verifique !!'
5     STOP
6  END IF
```

Uma array alocável não poderá ser utilizada num dado ponto do programa até que sua memória seja alocada para tal. Qualquer tentativa de usar uma array que não esteja alocada produzirá um erro e com isso sua execução será abortada. O Fortran 90/95 inclui a função lógica intrínseca `ALLOCATED()`, para habilitar o programa testar o estado da alocação de uma dada array, antes de tentar usá-la. Por exemplo, as seguintes linhas de um código computacional testam o estado de alocação da array `input_data`, antes de tentar realmente utilizá-la:

```
1  REAL, ALLOCATABLE, DIMENSION(:) :: input_data
2  ......
3  IF (ALLOCATED(imput_data)) THEN
4     READ(8,*) input_data
5  ELSE
6     WRITE(*,*) 'AVISO: Array nao Alocada !!'
7     STOP
8  END IF
```

Esta função pode ser útil em grandes programas, envolvendo muitos procedimentos de alocação dinâmica de memória.

No final do programa ou mesmo quando não precisamos mais da array, devemos desalocá-la da memória, com a declaração `DEALLOCATE`, liberando memória para ser reutilizada. A sua estrutura é

`DEALLOCATE`(lista das variáveis a serem desalocadas, `STAT`=nome do status)

Um exemplo:

```
DEALLOCATE(ontem(100,0:10), STATUS=info)
```

onde o `STATUS` tem o mesmo significado e uso que tem na declaração `ALLOCATE`. Após desalocar a array, os dados que a ela pertenciam não existem mais na memória. Então, tenha

muito cuidado. Devemos sempre desalocar qualquer array, uma vez que tenha terminado o seu uso. Esta prática é especialmente importante em SUBROUTINE e FUNCTION.

5.2 QUANDO DEVEMOS USAR UMA ARRAY?

Em programação, principalmente em Fortran, se fala muito em arrays, mas talvez nunca nos perguntamos: quando devemos usá-las? Em geral, se muitos ou todos os dados devem estar na memória ao mesmo tempo para resolver um problema eficientemente, então o uso de arrays para armazenar estes dados será apropriado, para este problema. Por outro lado, arrays não serão necessárias. O exemplo abaixo (parte de um programa) mostra como nem sempre é preciso usar uma array.

```
1   ....
2  DO i = 1,n  ! Le valores
3      WRITE(*,*) 'Entre com o numero: '
4      READ(*,*) x
5      WRITE(*,*) 'O numero eh: ',x
6      sum_x=sum_x + x ! Acumulando a soma
7      sum_x2=sum_x2 + x**2
8  END DO
9  ! Agora calcula a media (x_bar) e o desvio padrao (std_dev)
10 x_bar = sum_x/real(n)
11 std_dev = SQRT((real(n)*sum_x2 - sum_x**2)/(real(n)*real(n-1)))
12  ....
```

Perceba que os valores de x não foram armazenados, para cálculo da média (x_bar) e do desvio padrão (std_dev). Neste caso os dados foram lidos via teclado (linha 5). Estes mesmos dados poderiam ser lidos de um arquivo.

Os dois maiores problemas associados com uso de arrays desnecessárias são:

1. *Arrays desnecessárias desperdiçam memória.* Arrays desnecessárias podem "consumir" uma grande quantidade de memória, gerando com isso um programa maior do que ele necessita ser. Um programa grande requer mais memória para executá-lo, e portanto requer mais disponibilidade do computador. Em alguns casos, o tamanho extra do programa pode não ser executado num dado computador.
2. *Arrays desnecessárias restringem a eficiência do programa.* Para entender este ponto, vamos considerar o programa-exemplo acima, que calcula a média e o desvio-padrão de um conjunto de dados. Se o programa é projetado com 1000 elementos estáticos como entrada da array, então ele somente trabalhará para um conjunto de dados de até 1000 elementos. Se nós encontramos um conjunto de dados maior do que 1000 elementos, o programa terá que ser recompilado e *relinked* com um tamanho maior para a array. Por outro lado, um programa que calcula a média e o desvio-padrão de um conjunto de dados, que são "lidos" de um arquivo, não terá limite para o tamanho do conjunto de dados.

5.3

MANIPULAÇÃO ENTRE ARRAYS

Rapidamente veremos outra característica do Fortran 90/95, que é o fato de podermos operar com arrays, tal como fazemos com números. Isto é, quando operamos $a + b = c$, se $a = 5$ e $b = 6$, c será 11. Se as arrays são conformes (mesma forma), este tipo de operação fica subentendida. Vejamos o caso abaixo: (Digite e execute-o!)

```
1  PROGRAM operacao_array
2  IMPLICIT NONE
3  INTEGER :: i
4  REAL, DIMENSION(4) :: a = (/1., 2., 3., 4./)
5  REAL, DIMENSION(4) :: b = (/5., 6., 7., 8./)
6  REAL, DIMENSION(4) :: c, d
7      DO i = 1,4
8          c(i) = a(i) + b(i)
9      END DO
10     d = a + b
11     WRITE(*,100)'c', c
12     WRITE(*,100)'d', d
13 100 FORMAT (' ', A, ' = ', 5(F6.1,1X))
14 END PROGRAM operacao_array
```

Neste exemplo, a array c resulta da soma dos elementos conformes da array a com os da array b. Já a array d é obtida usando a nova instrução do Fortran 90/95, que faz implicitamente a descrição anterior.

★ **Lista de Exercícios 2**, só assim, exercitando, saberemos de nossas limitações!!!!

CAPÍTULO 6

As Sub-rotinas e Funções

Neste capítulo você encontrará:

Para reflexão !!

"Só existem duas coisas infinitas: o universo e a estupidez humana. E não estou muito seguro da primeira."

Albert Einstein
1879 – 1955

6.1 INTRODUÇÃO

É possível escrever um programa completo em Fortran em um único programa principal. Mas, se o código é complexo ou muito extenso, pode ser que um determinado conjunto de instruções seja realizado repetidas vezes, em pontos distintos do programa. Neste caso, é melhor quebrar o programa em unidades distintas, chamadas de subprogramas. Os subprogramas são unidades de programa que realizam tarefas específicas. Podem ser chamados pelo nome a partir do programa principal ou de outros subprogramas e, até mesmo por ele próprio, conhecida como *chamada recursiva*.

Cada uma dessas unidades de programa corresponde a um conjunto completo e consistente de tarefas que podem ser escritas, compiladas e testadas individualmente. Posteriormente são incluídas no programa principal para gerar um arquivo executável.

Outra importante função dos subprogramas é a possibilidade de depurar erros. Assim, os programadores podem criar o subprograma como uma unidade de programa, também chamada de **procedimento** em Fortran. Depois de aprovada, isto é, após compilar, testar e depurar os erros, o subprograma pode ser agregado ao programa principal. Em Fortran há dois tipos de subprogramas ou procedimentos que se encaixam nesta categoria: sub-rotinas e funções. A agregação pode ser realizada por procedimento interno ou externo.

PROCEDIMENTOS EXTERNOS

Os procedimentos externos se localizam externamente ao programa principal, isto é, após o **`END PROGRAM`** ou ainda em um arquivo separado, e neste caso podem ser de outra linguagem. Nestes últimos, o procedimento externo (sub-rotina ou função), deve ser compilado separadamente e após anexado ao executável principal, por intermédio de opções de compi-

lação do programa principal, que veremos nesta seção. Os procedimentos externos de outra linguagem muito utilizados são os da linguagem C.

A estrutura geral e localização é, para o caso do procedimento externo pertencer ao mesmo arquivo do programa principal:

```
[PROGRAM nome_do_programa]
[USE nome_do_use]
[IMPLICIT NONE]
[declaração global dos dados]
   declarações executáveis
END [PROGRAM nome_do_programa]
  [procedimentos ou subprogramas externos]        ⟸
```

Um programa principal pode ter quantos procedimentos externos forem necessários. Se o procedimento externo não estiver contido no mesmo arquivo (.f90) que está o programa principal é necessário anexá-lo no executável final. Isto é realizado compilando-se da seguinte forma. Sejam dois procedimentos externos localizados, respectivamente, nos arquivos **`sub1.f90`** e **`sub2.f90`**, e o programa fonte principal no **`prog.f90`**. Para gerar um executavel, de nome **`calculo1`**, que contenha os procedimentos externos anexados ao programa principal, executamos o seguinte comando:

```
g95 -o calculo1 prog.f90 sub1.f90 sub2.f90
```

que compila múltiplos arquivos de código fonte e os anexa para produzir um arquivo executável chamado **`calculo1`** no linux ou unix, ou **`calculo1.exe`** no sistema MS Windows.

PROCEDIMENTOS INTERNOS

O Fortran 95 também permite procedimentos internos, os quais se localizam no programa principal após a instrução CONTAINS, a qual finaliza os procedimentos executáveis do programa principal.

fortran 95

A estrutura geral e localização de um procedimento interno é:

```
[PROGRAM nome_do_programa]
[USE nome_do_use]
[IMPLICIT NONE]
[declaração global dos dados]
    declarações executáveis
CONTAINS                                          ⟸
  [procedimentos ou subprogramas internos]        ⟸
END [PROGRAM nome_do_programa]
```

Um programa principal pode ter quantos procedimentos internos forem necessários.

O Fortran tem dois tipos de procedimentos ou subprogramas: **sub-rotinas** (**`SUBROUTINE`**) e **funções** (**`FUNCTION`**). A diferença fundamental entre esses dois tipos é que, as sub-rotinas são chamadas pela instrução **`CALL`**, com o respectivo nome, e podem retornar *múltiplos resultados* através de seus argumentos. Já as funções são ativadas pelo seu nome na expressão e, o seu resultado é *um único valor* passado diretamente pelo uso da função, como

ocorre por exemplo, com o cálculo da função seno ou cosseno, nas funções intrínsecas. Ambos os procedimentos serão descritos a seguir.

Os benefícios dos subprogramas são principalmente:

1. **Testes Independentes.** Cada subprograma pode ser codificado e compilado como uma unidade independente, antes de ser incorporado ao programa principal. Este passo é conhecido como uma *unidade de teste*.
2. **Procedimentos Re-utilizáveis.** Em muitos casos, diferentes partes de um programa podem usar o mesmo subprograma. Com isto reduz o esforço de programação e também simplifica a depuração dos erros.
3. **Isolamento do restante do Programa.** As únicas variáveis no programa principal que podem se comunicar (e também serem trocadas) pelo procedimento são as que estão declaradas nos argumentos.
3. **Cooperativismo.** Os subprogramas podem ser desenvolvidos por diferentes programadores e em diferentes partes do mundo, e após disponibilizados para uso. Um exemplo é o repositório de subprogramas da biblioteca matemáica LAPACK, que será estudada na pág. 103.

O uso de subprogramas é uma **boa prática de programação** em códigos muito extensos.

6.2 AS SUB-ROTINAS – `SUBROUTINE`

Uma sub-rotina é um procedimento Fortran que é chamado pela declaração **`CALL`**, que recebe valores de entrada e retorna valores de saída através de uma lista de argumentos. A forma geral de uma sub-rotina é:

```
SUBROUTINE <nome_da_sub-rotina>(<lista_de_argumentos>)
IMPLICIT NONE
   <declarações de variáveis locais>
     ...
   <procedimentos executáveis>
     ...
  [RETURN]
END SUBROUTINE <nome_da_sub-rotina>
```

A declaração **`SUBROUTINE`** marca o início de uma sub-rotina. O nome da sub-rotina deve seguir os padrões do Fortran: deve ter até 31 caracteres e pode ter tanto letras do alfabeto como números, mas o primeiro caracter deve ser - obrigatoriamente - uma letra. A lista de argumentos contém uma lista de variáveis, arrays ou ambas que são passadas para a sub-rotina quando a mesma é ativada. Estas variáveis são chamadas **argumentos mudos** (*dummy arguments*), porque a sub-rotina não aloca memória para elas. A alocação será efetivada quando os argumentos forem passados na chamada da sub-rotina.

Qualquer unidade de programa pode chamar uma sub-rotina, até mesmo outra sub-

rotina[N1]. Para usar ou "chamar" uma sub-rotina é usado a declaração **CALL**, da seguinte maneira:

```
CALL <nome_da_sub-rotina>(<lista_de_argumentos>)
```

em que, a ordem e tipo dos argumentos na **lista_de_argumentos** devem corresponder a ordem e tipo dos argumentos mudos declarados na sub-rotina. A sub-rotina finaliza sua execução quando encontra um **RETURN** ou um **END SUBROUTINE** e, retorna ao programa que a requisitou na linha seguinte ao **CALL**. Quando a execução encontra um **RETURN**, imediatamente o fluxo de execução é desviado para a linha que contém o **END SUBROUTINE**. Um exemplo simples ilustra melhor o que é uma sub-rotina.

Programa 6.1 – Um exemplo de sub-rotina.

```
1  SUBROUTINE exemplo_sub(lado1, lado2, hipotenusa)
2  IMPLICIT NONE
3  ! Calcula hippotenusa de um triangulo retangulo
4  ! Declaracao dos parametros de chamada
5  REAL, INTENT(IN) :: lado1        ! Dado de entrada da sub-rotina
6  REAL, INTENT(IN) :: lado2        ! Dado de entrada da sub-rotina
7  REAL, INTENT(OUT) :: hipotenusa  ! Dado de saida da sub-rotina
8  ! Declaracao das variaveis locais (internamente a sub-rotina)
9  REAL :: temp
10    temp = lado1**2 + lado2**2
11    hipotenusa = SQRT(temp)
12 RETURN
13 END SUBROUTINE exemplo_sub
```

Neste exemplo, que calcula a hipotenusa de um triângulo retângulo, três argumentos são passados para a sub-rotina. Dois argumentos são de entrada (**lado1** e **lado2**) e um de saída (**hipotenusa**). Aqui é introduzida uma novidade do Fortran 95: o atributo INTENT, que especifica o uso pretendido dos argumentos da sub-rotina (e também das funções), descritos a seguir:

fortran 95

INTENT(IN): este atributo especifica que o argumento do subprograma seja recebido como **dado de entrada** pelo subprograma, quando é chamado por uma unidade de programa. Assim, o argumento não poderá ser redefinido ou ficar indefinido durante a execução do procedimento (subprograma).

INTENT(OUT): este atributo especifica que o argumento do subprograma é um **dado de saída** do subprograma, quando é chamado por uma unidade de programa. Cada argumento associado dessa forma deve ser definido ou atribuido um valor.

INTENT(INOUT): este atributo especifica que o argumento do subprograma seja utilizado para ambas possibilidades, isto é, como como **dado de entrada** e **dado de saída** do subprograma, quando é chamado por uma unidade de programa. Cada argumento associado dessa forma deve ser definido ou atribuido um valor.

A variável **temp** é definida somente para uso interno, i.e., ela não será acessada externamente a sub-rotina. Esta característica é importante porque poderemos usar nomes iguais para outros procedimentos, desde que um seja interno a(s) sub-rotina(s) e o outro no corpo

[N1]Uma sub-rotina pode chamar outra sub-rotina, mas não a si mesmo, a menos que seja declarada como *recursiva*. Mais informações sobre sub-rotinas recursivas são obtidas nas referências [6][10].

do programa. Esta sub-rotina é usada num programa ou em outra sub-rotina, por intermédio da declaração **`CALL exemplo_sub(lado1, lado2, hipotenusa)`**, como no exemplo abaixo:

Programa 6.2 – Um programa para testar a sub-rotina do Programa 6.1.

```
1  PROGRAM testa_sub
2  IMPLICIT NONE
3  REAL :: s1, s2, hip
4     WRITE(*,*)'Indique um dos lados de um triangulo retangulo: '
5     READ(*,*)s1
6     WRITE(*,*)'Indique o outro lado do triangulo retangulo: '
7     READ(*,*)s2
8     CALL exemplo_sub(s1, s2, hip)
9     WRITE(*,*) 'O valor da hipotenusa do triangulo eh: ',hip
10 END PROGRAM testa_sub
```

Outras características importantes, tal como alocação de memória automática para arrays, estão descritos detalhadamente nas referências indicadas anteriormente.

6.3 AS FUNÇÕES – `FUNCTION`

Uma função Fortran é um procedimento que é ativado em uma expressão que pertence a um comando de programa. A função retorna um único valor numérico, ou lógico, ou caracter ou uma array. O Fortran tem dois tipos de funções: **funções intrínsecas** e **funções definidas pelo usuário** *(funções definida-usuário)*.

Funções intrínsecas, estudadas no Capítulo 1, na página 30, são próprias (latentes) da linguagem Fortran, tais como **`SIN(X)`**, **`COS(X)`**, **`SQRT(X)`**, entre outras. Para saber quais são as funções intrínsecas consulte o Manual do Usuário.

As funções definida-usuário são funções que o programador cria para executar uma tarefa específica. A forma geral de uma função definida-usuário é:

```
[Tipo] FUNCTION <nome_da_função>(<lista_de_argumentos>)
<declarações de variáveis locais>
    ...
<procedimentos executáveis>
    ...
  <nome_da_função = expressão>
[RETURN]
END FUNCTION <nome_da_função>
```

A função definida-usuário (ou simplesmente função) deve ser iniciada com a instrução **`FUNCTION`** e finalizada com uma instrução **`END FUNCTION`**. O nome da função deve seguir, como nas sub-rotinas, os padrões do Fortran, isto é, deve ter até 31 caracteres e pode ter tanto letras do alfabeto como números, mas o primeiro caracter deve ser – obrigatoriamente – uma letra. A função é ativada pelo seu nome, em uma expressão e, sua execução começa no topo da função e termina quando encontra um **`RETURN`** ou **`END FUNCTION`**. A instrução **`RETURN`** é opcional e é raramente utilizada, pois a execução sempre termina num **`END FUNCTION`**. A

declaração **Tipo** é opcional se a declaração **IMPLICIT NONE** estiver presente. Caso contrário, é necessário declarar o tipo de função. Estes tipos podem ser **REAL**, **INTEGER**, **COMPLEX**, **CHARACTER** ou **LOGICAL**. Após ser executada, a função retorna um valor que será usado para continuar a execução da expressão na qual a função foi chamada. Um exemplo de função definida-usuário é mostrado abaixo, a qual calcula o valor da função

$$f(x) = ax^2 + bx + c\,,$$

num ponto x.

Programa 6.3 – Um exemplo de função.

```
1  REAL FUNCTION func(x,a,b,c)
2  ! Objetivo: calcular um polinomio quadratico do tipo
3  !           a*x**2 + b*x + c
4  IMPLICIT NONE
5  REAL, INTENT(IN) :: x, a, b, c     ! Dados de entrada da funcao
6  ! Calculo da expressao
7      func = a*x**2 + b*x + c
8  END FUNCTION func
```

Esta função produz um resultado real. Observe que o atributo **INTENT** não é usado com a declaração do nome da função **func**, porque ela sempre será usada somente como saída. Note também que, se não fosse declarada como real, a variável **func** deveria ser declarada no corpo da **FUNCTION**, como de hábito. Um programa que usa esta função pode ser:

Programa 6.4 – Um programa para testar função definida-usuário do Programa 6.3.

```
1  PROGRAM testa_func
2  IMPLICIT NONE
3  ! Testa a funcao que calcula f(x) = ax**2 + b*x + c
4  REAL :: func
5  REAL :: a, b, c, x
6      WRITE(*,*) 'Entre com os coef. quadraticos a, b e c: '
7      WRITE(*,*) 'Digite o coef. a: '
8      READ(*,*) a
9      WRITE(*,*) 'Digite o coef. b: '
10     READ(*,*) b
11     WRITE(*,*) 'Digite o coef. c: '
12     READ(*,*) c
13     WRITE(*,*) 'Entre com a localizacao na qual quer fazer o calculo: '
14     READ(*,*) x
15     WRITE(*,100) ' Calculo em (',x,') = ', func(x,a,b,c)
16 100 FORMAT(A,F10.4,A,F12.4)
17 END PROGRAM testa_func
```

Note que a função **func** é declarada como tipo real tanto na própria função, como no programa principal da qual é ativada. Para mais informações, procure pela literatura indicada nas Referências Bibliográficas.

6.4
AS BIBLIOTECAS DE SUB-ROTINAS E FUNÇÕES

A Netlib é um conjunto de códigos matemáticos, artigos e base de dados. O endereço eletrônico é *http://www.netlib.org/*. O uso das subrotinas e funçõe prontas, como LAPACK e BLAS, são ótimos exemplos.

6.4.1 A biblioteca LAPACK

A biblioteca LAPACK é escrita em FORTRAN 77 e fornece rotinas para resolver equações lineares simultâneas, soluções de sistemas de equações lineares por mínimos-quadrados, problemas de autovalores, e problemas de valores singulares. Também estão disponibilizadas soluções para fatorização de matrizes associadas (LU, Cholesky, QR, SVD, Schur, Schur generalizada). Soluções com matrizes densas e esparsas também estão disponíveis, bem como funcionalidades para tratar com matrizes reais e complexas, tanto em precisão simples como dupla.

Esta parte do livro está em fase de elaboração e finalização. Desculpe!!!!!

Programa 6.5 – A sub-rotina **DGESV** da LAPACK.

```
      SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )
*
*  -- LAPACK driver routine (version 3.1) --
*     Univ. of Tennessee, Univ. of California Berkeley and NAG Ltd..
*     November 2006
*
*     .. Scalar Arguments ..
      INTEGER            INFO, LDA, LDB, N, NRHS
*     ..
*     .. Array Arguments ..
      INTEGER            IPIV( * )
      DOUBLE PRECISION   A( LDA, * ), B( LDB, * )
*     ..
*
*  Purpose
*  =======
*
*  DGESV computes the solution to a real system of linear equations
*     A * X = B,
*  where A is an N-by-N matrix and X and B are N-by-NRHS matrices.
*
*  The LU decomposition with partial pivoting and row interchanges is
*  used to factor A as
*     A = P * L * U,
*  where P is a permutation matrix, L is unit lower triangular, and U is
*  upper triangular.  The factored form of A is then used to solve the
*  system of equations A * X = B.
*
*  Arguments
*  =========
*
*  N       (input) INTEGER
*          The number of linear equations, i.e., the order of the
*          matrix A.  N >= 0.
*
*  NRHS    (input) INTEGER
*          The number of right hand sides, i.e., the number of columns
*          of the matrix B.  NRHS >= 0.
```

```
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the N-by-N coefficient matrix A.
*          On exit, the factors L and U from the factorization
*          A = P*L*U; the unit diagonal elements of L are not stored.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*
*  IPIV    (output) INTEGER array, dimension (N)
*          The pivot indices that define the permutation matrix P;
*          row i of the matrix was interchanged with row IPIV(i).
*
*  B       (input/output) DOUBLE PRECISION array, dimension (LDB,NRHS)
*          On entry, the N-by-NRHS matrix of right hand side matrix B.
*          On exit, if INFO = 0, the N-by-NRHS solution matrix X.
*
*  LDB     (input) INTEGER
*          The leading dimension of the array B.  LDB >= max(1,N).
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value
*          > 0:  if INFO = i, U(i,i) is exactly zero.  The factorization
*                has been completed, but the factor U is exactly
*                singular, so the solution could not be computed.
*
*  =====================================================================
*
*     .. External Subroutines ..
      EXTERNAL           DGETRF, DGETRS, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX
*     ..
*     .. Executable Statements ..
*
*     Test the input parameters.
*
      INFO = 0
      IF( N.LT.0 ) THEN
         INFO = -1
      ELSE IF( NRHS.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -4
      ELSE IF( LDB.LT.MAX( 1, N ) ) THEN
         INFO = -7
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'DGESV ', -INFO )
         RETURN
      END IF
*
*     Compute the LU factorization of A.
*
      CALL DGETRF( N, N, A, LDA, IPIV, INFO )
      IF( INFO.EQ.0 ) THEN
*
*        Solve the system A*X = B, overwriting B with X.
*
         CALL DGETRS( 'No transpose', N, NRHS, A, LDA, IPIV, B, LDB,
     $                INFO )
      END IF
      RETURN
*
*     End of DGESV
```

```
106 *
107       END
```

Programa 6–1: A sub-rotina **DGESV** da LAPACK

```
1         SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )
2   *
3   *  -- LAPACK driver routine (version 3.1) --
4   *     Univ. of Tennessee, Univ. of California Berkeley and NAG Ltd..
5   *     November 2006
6   *
7   *     .. Scalar Arguments ..
8         INTEGER            INFO, LDA, LDB, N, NRHS
9   *     ..
10  *     .. Array Arguments ..
11        INTEGER            IPIV( * )
12        DOUBLE PRECISION   A( LDA, * ), B( LDB, * )
13  *     ..
14  *
15  *  Purpose
16  *  =======
17  *
18  *  DGESV computes the solution to a real system of linear equations
19  *     A * X = B,
20  *  where A is an N-by-N matrix and X and B are N-by-NRHS matrices.
21  *
22  *  The LU decomposition with partial pivoting and row interchanges is
23  *  used to factor A as
24  *     A = P * L * U,
25  *  where P is a permutation matrix, L is unit lower triangular, and U is
26  *  upper triangular.  The factored form of A is then used to solve the
27  *  system of equations A * X = B.
28  *
29  *  Arguments
30  *  =========
31  *
32  *  N       (input) INTEGER
33  *          The number of linear equations, i.e., the order of the
34  *          matrix A.  N >= 0.
35  *
36  *  NRHS    (input) INTEGER
37  *          The number of right hand sides, i.e., the number of columns
38  *          of the matrix B.  NRHS >= 0.
39  *
40  *  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
41  *          On entry, the N-by-N coefficient matrix A.
42  *          On exit, the factors L and U from the factorization
43  *          A = P*L*U; the unit diagonal elements of L are not stored.
44  *
45  *  LDA     (input) INTEGER
46  *          The leading dimension of the array A.  LDA >= max(1,N).
47  *
48  *  IPIV    (output) INTEGER array, dimension (N)
49  *          The pivot indices that define the permutation matrix P;
50  *          row i of the matrix was interchanged with row IPIV(i).
51  *
52  *  B       (input/output) DOUBLE PRECISION array, dimension (LDB,NRHS)
53  *          On entry, the N-by-NRHS matrix of right hand side matrix B.
54  *          On exit, if INFO = 0, the N-by-NRHS solution matrix X.
55  *
56  *  LDB     (input) INTEGER
57  *          The leading dimension of the array B.  LDB >= max(1,N).
58  *
59  *  INFO    (output) INTEGER
60  *          = 0:  successful exit
61  *          < 0:  if INFO = -i, the i-th argument had an illegal value
```

Continuação do Programa 6–1 ...

```
62  *          > 0:  if INFO = i, U(i,i) is exactly zero.  The factorization
63  *                has been completed, but the factor U is exactly
64  *                singular, so the solution could not be computed.
65  *
66  *  =====================================================================
67  *
68  *     .. External Subroutines ..
69        EXTERNAL           DGETRF, DGETRS, XERBLA
70  *     ..
71  *     .. Intrinsic Functions ..
72        INTRINSIC          MAX
73  *     ..
74  *     .. Executable Statements ..
75  *
76  *     Test the input parameters.
77  *
78        INFO = 0
79        IF( N.LT.0 ) THEN
80           INFO = -1
81        ELSE IF( NRHS.LT.0 ) THEN
82           INFO = -2
83        ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
84           INFO = -4
85        ELSE IF( LDB.LT.MAX( 1, N ) ) THEN
86           INFO = -7
87        END IF
88        IF( INFO.NE.0 ) THEN
89           CALL XERBLA( 'DGESV ', -INFO )
90           RETURN
91        END IF
92  *
93  *        Solve the system A*X = B, overwriting B with X.
94  *
95           CALL DGETRS( 'No transpose', N, NRHS, A, LDA, IPIV, B, LDB,
96       $               INFO )
97        END IF
98        RETURN
99  *
100 *     End of DGESV
101 *
102       END
```

CAPÍTULO 7

As Funções Intrínsecas SELECTED_REAL_KIND e SELECTED_INT_KIND

Neste capítulo você encontrará:

Para reflexão !!

"Você não pode provar uma definição. O que você pode fazer, é mostrar que ela faz sentido".

Albert Einstein
1879 – 1955

Vimos que na maioria dos computadores, a variável real *default* é **precisão simples**, a qual usualmente tem 4 Bytes, divididos em duas partes: mantissa e expoente. Para a **precisão dupla**, usualmente, é dedicado 8 Bytes. Usando estas declarações ficaremos dependentes da combinação compilador/processador. Podemos começar a alterar esta dependência, usando o parâmetro `KIND` na declaração de variáveis. Assim, precisão simples e dupla tem valores específicos neste parâmetro. Vejamos os exemplos:

default = na omissão de declaração.

```
REAL(KIND=1) :: valor_1
REAL(KIND=4) :: valor_2
REAL(KIND=8), DIMENSION(20) :: matriz_a
REAL(4) :: temp
```

O tipo de valor real é especificado nos parênteses após o `REAL`, com ou sem `KIND=`. Uma variável declarada com este tipo de parâmetro é chamado de *variável parametrizada*. Se nenhum tipo é especificado, então o tipo real *default* é usado. Mas afinal, que significa o tipo de parâmetro (em `KIND`)? Infelizmente, não temos como saber. Cada compilador é livre para atribuir um número para cada tamanho de variável. Por exemplo, em alguns compiladores, o valor real com 32 bits é igual a `KIND=1` e o valor com 64 bits é `KIND=2`, que é o caso da combinação PC/NAGWare FTN90. Em outros compiladores, como PC/Lahey-Fujitsu Fortran 90/95 e PC/Microsoft PowerStation 4.0, temos `KIND=4` e `KIND=8`, para respectivamente, 32 bits e 64 bits.

Portanto, para tornar nossos programas portáveis, entre compiladores e máquinas diferentes, devemos sempre fornecer o valor correto para o tipo de parâmetro. Para isso, pode-

mos usar a função intrínseca KIND, que retorna o número que específica o tipo de parâmetro usado para simples e dupla precisão. Uma vez descoberto estes valores, podemos usá-los nas declarações das variáveis reais. Vejamos como funciona a função intrínseca KIND, através de um programa:

```
1  PROGRAM kinds
2  ! Proposito: determinar os tipos de parametros de simples e
3  !            dupla precisao num dado computador e compilador
4  IMPLICIT NONE
5  ! Escreve na tela os tipos de parâmetros
6  WRITE(*,'(" O KIND para Precisao Simples eh ",I2)')KIND(0.0)
7  WRITE(*,'(" O KIND para Precisao Dupla   eh ",I2)')KIND(0.0D0)
8  END PROGRAM kinds
```

Na tabela 7.1 é apresentado os resultados da execução deste programa, em quatro diferentes combinações de Processador/Compilador.

Tabela 7.1 – Valores de KIND para valores reais em alguns compiladores

	KIND		
	32 bits	64 bits	128 bits
PC-Pentium/Lahey-Fujitsu Fortran 90/95	4	8	16
PC-Pentium/NAGWare Fortran 90	1	2	N/D
Cray T90 Supercompuador/CF90	N/D	8	16
SPARC/CF90	4	8	N/D

A partir destes resultados podemos migrar entre estas máquinas e compiladores, simplesmente trocando os parâmetros do KIND. Vejamos um exemplo de programa que use este procedimento, com os dados do *PC-Pentium/Lahey-Fujitsu Fortran 90/95*:

```
1  PROGRAM uso_do_kind
2  !
3  ! Proposito: usar o KIND como parametro
4  !
5  IMPLICIT NONE
6  INTEGER, PARAMETER :: single = 4
7  INTEGER, PARAMETER :: double = 8
8  REAL(KIND=single) :: valor_1
9  REAL(KIND=double), DIMENSION(20) :: matriz_1
10 REAL(single) :: temp
11   .....
12   executaveis
13   .....
14 END PROGRAM uso_do_kind
```

Se trocarmos de máquina e/ou compilador, basta trocarmos os valores do `single` e `double`, para os correspondentes tipos para simples e dupla precisão, respectivamente. Mas, o melhor vem agora !!!!

7.1 SELECIONANDO PRECISÃO DE MANEIRA INDEPENDENTE DO PROCESSADOR

Como já sabemos, o maior problema encontrado quando portamos um programa Fortran de um computador para outro é o fato que os termos *precisão simples* e *precisão dupla* não são precisamente definidos. Os valores com precisão dupla tem, aproximadamente, duas vezes o valor dos valores com precisão simples, mas o número de bits associado com cada tipo de número real dependerá de cada compilador. Também já sabemos que em muitos computadores, 32 bits está associado com a precisão simples e 64 bits com a dupla precisão. Num computador Cray é diferente, conforme tabela anterior.

Então, como podemos escrever programas que possam ser facilmente portáveis entre processadores diferentes, com definições de precisão simples e dupla diferentes e assim mesmo funcionar corretamente? A resposta está num dos avanços da linguagem Fortran. Agora, é possível especificarmos valores para a mantissa e o expoente, conforme a conveniência e, com isso também obtermos maior portabilidade do programa. Isto é feito através de uma função intrínseca que seleciona automaticamente o tipo de valor real para usar quando se troca de computador. Esta função é chamada `SELECTED_REAL_KIND`. A forma geral desta função é

`SELECTED_REAL_KIND(p=`*precisão*`,r=`*expoente(ou range)*`)`

onde *precisão* é o número de dígitos decimais requerido e *range* é o tamanho do expoente requerido da potência de 10. Os dois argumentos *precisão* e *range* são argumentos opcionais; um deles ou ambos podem ser informados. Vejamos os exemplos abaixo:

```
kind_number = SELECTED_REAL_KIND(p=6,r=37)
kind_number = SELECTED_REAL_KIND(p=12)
kind_number = SELECTED_REAL_KIND(r=100)
kind_number = SELECTED_REAL_KIND(13,200)
kind_number = SELECTED_REAL_KIND(13)
kind_number = SELECTED_REAL_KIND(p=17)
```

Num computador com processador PC-Pentium e usando o compilador *Lahey-Fujitsu Fortran 90/95*, a primeira função retornará um 4, (para precisão simples) e as outras quatro funções retornarão um 8 (precisão dupla). A última função retornará 16, mas para o compilador da Portland (PGHPF), retornará um -1, porque não existe este tipo de dado real no processador Pentium-PC. Outros, retornarão valores distintos, tente você mesmo descobrir.

Observe que, dos exemplos, que tanto o `p=` e `r=` são opcionais e, `p=` é opcional se somente a precisão é desejada.

A função `SELECTED_REAL_KIND` deve ser usada com precaução, pois a especificação desejada no seu programa pode aumentar o tamanho do mesmo e com isso sua execução pode ficar mais lento. Por exemplo, computadores com 32 bits tem entre 6 e 7 dígitos decimais de precisão, para as varáveis com precisão simples. Assim, se foi especificado

SELECTED_REAL_KIND(6), então nestas máquinas será precisão simples. Entretanto, se especificar SELECTED_REAL_KIND(7), será dupla precisão.

É possível usar outras funções intrínsecas para determinar o tipo (KIND) de uma variável real e, sua precisão e expoente, num dado computador. A tabela 7.2 descreve estas funções.

Tabela 7.2 – Funções Intrínsecas relacionadas com o KIND

Função	Descrição
SELECTED_REAL_KIND(p,r)	Retorna o menor tipo de parâmetro real com um valor mínimo de p dígitos decimais de precisão e máximo intervalo $\geq 10^r$.
SELECTED_INT_KIND(r)	Retorna o menor tipo de parâmetro inteiro com máximo intervalo $\geq 10^r$.
KIND(X)	Retorna o número que especifica o tipo de parâmetro de X, onde X é uma variável ou constante de algum tipo intrínseco.
PRECISION(X)	Retorna a precisão decimal de X, onde X é um valor real ou complexo.
RANGE(X)	Retorna o expoente da potência de 10 para X, onde X é um valor inteiro, real ou complexo.

Observe, pela tabela 7.2, que o procedimento de escolha de precisão é também válido para os números inteiros. A função para isto é SELECTED_INT_KIND(r), e o exemplo abaixo ilustra seu uso:

```
kind_number = SELECTED_INT_KIND(3)
kind_number = SELECTED_INT_KIND(9)
kind_number = SELECTED_INT_KIND(12)
```

Usando um processador PC-Pentium e o compilador da Lahey/Fujitsu, a primeira função retornará um 2 (para 2 Bytes inteiros), representando um intervalo de representação entre -32.768 e 32.767. Igualmente, a segunda função retornará um 4 (4 Bytes), que fornecerá um intervalo entre -2.147.483.648 e 2.147.483.647. A última função retornará um 8 (8 Bytes), com intervalo entre -9.223.372.036.854.775.808 e 9.223.372.036.854.775.807. Em outros compiladores, este último pode retornar -1, pois poderá fugir de sua representatividade.

CAPÍTULO 8

Os Procedimentos MODULE

A linguagem Fortran surgiu na década de 50, sendo a primeira linguagem de alto nível a ser criada. Embora seja a precursora das linguagens, ela foi projetada com os conceitos da programação estruturada. No que diz respeito à modularização de programas, a linguagem Fortran oferece facilidades através de sub-rotinas (SUBROUTINE) e funções (FUNCTION), o que torna possível a implementação de programas modulares e estruturados. No Fortran 90/95, esta modularização teve um avanço significativo através das declarações e procedimentos MODULE, tanto que esta declaração tem *status* de programa. Como veremos, esta característica é muito importante.

Um dos usos da declaração MODULE é substituir as declarações COMMON, no compartilhamento de dados. Antes de estudarmos esta utilidade, veremos qual a função do COMMON nos programas Fortran.

8.1 A DECLARAÇÃO COMMON

Programas e subprograma em Fortran podem utilizar variáveis que são declaradas de forma a compartilhar uma mesma área de memória. Este compartilhamento tem a finalidade de economizar memória, pois variáveis de módulos (ou subprograma) diferentes ocuparão uma mesma posição de memória. Isto anos atrás era uma característica muito utilizada, pois era visível o problema de memória. Hoje, este problema pode até ser amenizado, mas sempre que pudermos economizar memória, melhor!! Assim, continuamos sempre otimizando o uso de memória e os COMMON ainda são usados.

O uso do COMMON, e o seu compartilhamento, torna possível a transferência de informações entre subprogramas, sem (ou de forma complementar) a utilização da passagem por parâmetros. A área de memória compartilhada pode ser dividida em blocos, onde cada um recebe um nome ou rótulo. A forma geral de se declarar variáveis com área compartilhada, conhecida como COMMON, é:

$$\texttt{COMMON}\ \ /r_1/\textit{lista de identificadores}_1 \cdots /r_N/\textit{lista de identificadores}_N$$

onde r_i são nomes dos rótulos comuns de variáveis, *lista de identificadores*$_i$ são nomes de variáveis simples ou compostas que não podem ser diferentes. Um exemplo, parcialmente

reproduzido de um programa:

```
1  PROGRAM uso_common
2  IMPLICIT NONE
3  INTEGER :: i,m,n1,n2,ue,us
4  COMMON /area1/n1,n2,m
5    .....
6    CALL mdc
7    .....
8  END PROGRAM uso_common
9  !
10 ! Aqui comecam as sub-rotinas
11 !
12 SUBROUTINE mdc
13 INTEGER :: a,aux1,b,m
14 COMMON /area1/a,b,m
15     m = b
16     aux1 = MOD(a,b)
17   .....
18 END SUBROUTINE mdc
19   .....
```

Neste exemplo, os parâmetros da sub-rotina foram substituídos pelas variáveis da área `area1` da declaração `COMMON`.

A utilização de variáveis em `COMMON` não constitui, no entanto, uma boa norma de programação. A transferência de valores entre os subprogramas deve ser feita de preferência através de parâmetros; com isto, os subprogramas se tornarão mais independentes, mais fáceis de serem entendidos e modificados.

Os `COMMON` devem ser usados com cautela para evitar problemas, pois estão sujeitos a dois tipos de erros. **Melhor é não usar mesmo!!** Porque? Bem...., analisemos um programa, reproduzido parcialmente, que usa `COMMON` e cuja alocação de memória se encontra representada ao lado.

```
1  PROGRAM erro_common
2  IMPLICIT NONE
3  REAL :: a, b
4  REAL, DIMENSION(5) :: c
5  INTEGER :: i
6  COMMON / common1 / a, b, c, i
7    .....
8    CALL cuidado
9    .....
10 END PROGRAM erro_common
11 !
12 ! Aqui comecam a sub-rotina
13 !
14 SUBROUTINE cuidado
15 REAL :: x
16 REAL, DIMENSION(5) :: y
17 INTEGER :: i, j
18 COMMON / common1 / x, y, i, j
19   .....
20 END SUBROUTINE cuidado
```

Representação da Alocação da Memória no `COMMON`

Endereço na Memória	Programa (`erro_common`)	Sub-rotina (`cuidado`)
0000	a	x
0001	b	y(1)
0002	c(1)	y(2)
0003	c(2)	y(3)
0004	c(3)	y(4)
0005	c(4)	y(5)
0006	c(5)	i
0007	i	j

1º tipo de erro: observe que os 5 elementos da array `c` no programa principal e o seus correspondentes na sub-rotina estão "desalinhados". Portanto, `c(1)`, no programa principal, será a mesma variável `y(2)`, na sub-rotina. Se as arrays `c` e `y` são supostamente as mesmas, este "desalinhamento" causará sérios problemas.

2º tipo de erro: o elemento real da array `c(5)` no programa principal é idêntico a variável inteira `i`, na sub-rotina. É extremamente improvável (e indesejável) que a variável real armazenada em `c(5)` seja usada como um inteiro na sub-rotina `cuidado`.

Estes tipos de erros podem ser evitados se usarmos a declaração `MODULE`, no lugar do `COMMON`.

8.2 A DECLARAÇÃO `MODULE`

A declaração `MODULE` (ou módulo, simplesmente) pode conter dados, procedimentos, ou ambos, que podemos compartilhar entre unidades de programas (programa principal, subprograma e em outros `MODULE`). Os dados e procedimentos estarão disponíveis para uso na unidade de programa através da declaração `USE`, seguida do nome do módulo. Ficará mais claro com um exemplo simples.

8.2.1 Compartilhando Dados usando o `MODULE`

O módulo abaixo será compartilhado com outras duas unidades de programas. Vejamos:

```
MODULE teste
!
! Declara dados para compartilhar entre duas rotinas
!
IMPLICIT NONE
SAVE
INTEGER, PARAMETER :: num_vals = 5
REAL, DIMENSION(num_vals) :: valores
END MODULE teste
```

A declaração `SAVE` garante que todos os dados declarados no módulo serão preservados quando forem acessados por outros procedimentos. Ele deve sempre incluído em qualquer módulo que declara dados compartilhados. Agora, vejamos como usar o módulo acima, através do seguinte programa:

SAVE: é um dos avanços do Fortran 90.

```
PROGRAM testa_module
!
! Ilustra o compartilhamento via MODULE
!
USE teste
IMPLICIT NONE
REAL, PARAMETER :: pi = 3.141592
    valores = pi*( /1., 2., 3., 4., 5. /)
    CALL sub1
```

```
CONTAINS
    SUBROUTINE sub1
    !
    ! Ilustra o compartilhamento via MODULE
    !
    USE teste
    IMPLICIT NONE
        WRITE(*,*) valores
    END SUBROUTINE sub1
END PROGRAM testa_module
```

CONTAINS: é outro avanço do Fortran 90 e, especifica que um módulo ou um programa contenham procedimentos internos.

Os conteúdos do módulo `teste` estão sendo compartilhados entre o programa principal e a sub-rotina `sub1`. Qualquer outra sub-rotina ou função dentro do programa também poderá ter acesso aos dados, simplesmente incluindo a declaração `USE`.

Módulos são especialmente úteis para compartilhar grandes volumes de dados entre unidades de programas.

Exercício: use o `MODULE` para evitar o erro descrito no exemplo da página 112 (programa `erro_common`).

Importante:

- A declaração `USE` é sempre a primeira declaração não comentada posicionada logo abaixo a declaração `PROGRAM`, `SUBROUTINE`, ou `FUNCTION`. Evidentemente, antes da declaração `IMPLICIT NONE`.
- O módulo deve ser **sempre** compilado antes de todas as outras unidades de programa que a usam. Ela pode estar no mesmo arquivo ou arquivo separado. Se estiver no mesmo arquivo, deve aparecer antes do programa principal. Muitos compiladores suportam a compilação separada e geram um arquivo `.mod` (ou similar), que contém informações sobre o módulo, para uso mais tarde com a declaração `USE`.

8.3 OS PROCEDIMENTOS `MODULE`

Além de dados, os módulos também podem conter sub-rotinas e funções, que são os **Procedimentos `MODULE` ou Módulos**. Estes procedimentos são compilados como uma parte do módulo e estarão disponíveis para as unidades de programa através da declaração `USE`. Os procedimentos que são incluídos dentro dos módulos devem vir após a declaração dos dados do módulo e precedidos por uma declaração `CONTAINS`. Esta declaração, tem a função de instruir o compilador que as declarações que a seguem são procedimentos incluídos no programa e, portanto, devem ser agregados na compilação.

No exemplo abaixo, a sub-rotina `sub1`, está contida no interior do módulo `mod_proc1`.

```
MODULE mod_proc1
IMPLICIT NONE
!
! Aqui sao declarados os dados
```

```
!
CONTAINS
    SUBROUTINE sub1(a, b, c, x, error)
    IMPLICIT NONE
    REAL, DIMENSION(3), INTENT(IN) :: a
    REAL, INTENT(IN) :: b, c
    REAL, INTENT(OUT) :: x
    LOGICAL, INTENT(OUT) :: error
    ......
    END SUBROUTINE sub1
END MODULE mod_proc1
```

A sub-rotina `sub1` estará disponível para uso numa unidade de programa através do `USE mod_proc1`, posicionado como vimos anteriormente. A sub-rotina é ativada com a declaração padrão `CALL`, por exemplo:

```
PROGRAM testa_mod_proc1
USE mod_proc1
IMPLICIT NONE
    ......
    CALL sub1(a, b, c, x, error)
    ......
END PROGRAM testa_mod_proc1
```

8.3.1 Usando Módulos para Criar Interfaces Explícitas

Mas porque nos darmos o trabalho de incluir procedimentos (sub-rotinas e funções) num módulo? Já sabemos que é possível compilar separadamente uma sub-rotina e chamá-la numa outra unidade programa, então porque passar por etapas extras, i.e., incluir uma sub-rotina num módulo, compilar o módulo, declarar o módulo através da declaração `USE`, e só aí chamar a sub-rotina?

A resposta é que quando um procedimento é compilado dentro de um módulo e o módulo é usado numa chamada de programa, todos os detalhes da interface de procedimentos estão disponíveis para o compilador. Assim, quando o programa que usa a sub-rotina é compilado, o compilador pode automaticamente verificar o número de argumentos na chamada do procedimento, o tipo de cada argumento, se cada argumento está ou não numa array, e o `INTENT`[N1] de cada argumento. Em resumo, o compilador pode capturar muito dos erros comuns que um programador pode cometer quando usa os procedimentos.

INTENT: outro avanço do Fortran 90. Esta declaração especifica a intenção de uso de um argumento mudo

Um procedimento compilado dentro de um módulo e acessado pelo `USE` é dito ter uma **Interface Explícita**. O compilador Fortran conhece todos os detalhes a respeito de cada argumento no procedimento sempre que o mesmo é utilizado, e o compilador verifica a interface para assegurar que está sendo usado adequadamente.

Ao contrário, procedimentos que não estão em módulos são chamados ter uma **Interface Implícita**. Desta forma, o compilador Fortran não tem informações a respeito destes

[N1] O `INTENT(xx)`, que especifica o tipo de uso do argumento mudo, onde o `xx` pode ser `IN`, `OUT` e `INOUT`. O atributo `INTENT(IN)` especifica que o argumento mudo é entrada na unidade de programa e não pode ser redefinido no seu interior; já o atributo `INTENT(OUT)` especifica que o argumento mudo é saída da unidade de programa e o atributo `INTENT(INOUT)` especifica que o argumento mudo é tanto de entrada como de saída na unidade de programa.

procedimentos, quando ele é compilado numa unidade programa, que o solicite. Assim, ele assume que o programador realmente verificou corretamente o número, o tipo, a intenção de uso, etc. dos argumentos. Se esta preocupação não foi tomada, numa seqüencia de chamada errada, o programa será executado com falha e será difícil de encontrá-la.

Nada melhor que um exemplo para dirimir dúvidas. O caso a seguir ilustra os efeitos da falta de concatenação quando a sub-rotina chamada está incluída num módulo. O módulo é dado por,

```
1  MODULE erro_interf
2  CONTAINS
3      SUBROUTINE bad_argumento (i)
4      IMPLICIT NONE
5      INTEGER, INTENT(IN) :: i
6      WRITE(*,*) ' I = ', i
7      END SUBROUTINE bad_argumento
8  END MODULE erro_interf
```

que será utilizado pelo programa a seguir:

```
1  PROGRAM bad_call
2  USE erro_interf
3  IMPLICIT NONE
4  REAL :: x = 1.
5      CALL bad_argumento (x)
6  END PROGRAM bad_call
```

Quando este programa é compilado, o compilador Fortran verificará e capturará o erro de declaração entre as duas unidades de programa, e nos avisará através de uma mensagem. Neste exemplo, que tem uma interface explícita entre o programa `bad_call` e a sub-rotina `bad_argumento`, um valor real (linha 4, do programa principal) foi passado para a sub-rotina quando um argumento inteiro (linha 5, do módulo) era esperado, e o número foi mal interpretado pela sub-rotina. Como foi dito, se este problema não estive numa interface explícita, o compilador Fortran não teria como verificar o erro na chamada do argumento.

Exercício: no exemplo acima, transforme a interface explícita em implícita, isto é, simplesmente elimine o módulo. Compile e execute! O que ocorrerá? *Dica: elimine o módulo, o* `CONTAINS` *e coloque a sub-rotina após o* `END PROGRAM` *e só aí compile.*

Existem outras maneiras de instruir o compilador Fortran para explicitar a verificação nos procedimentos por interface, é o bloco `INTERFACE`[6][10], que não será visto aqui.

8.3.2 A Acessibilidade `PUBLIC` e `PRIVATE`

Se não for especificado, todas as variáveis dos módulos estarão disponíveis para todas as unidades de programas, que contenham a declaração `USE` do referido módulo. Isto pode nem sempre ser desejado: é o caso se os procedimentos do módulo também contenham variáveis que pertençam só as suas próprias funções. Elas estarão mais a salvo se os usuários do pacote não interfiram com seus trabalhos internos. Por *default* todos os nomes num módulo são `PUBLIC`, mas isto pode ser trocado usando a declaração `PRIVATE`. Vejamos o exemplo:

```
1  MODULE change_ac
2  IMPLICIT NONE
3  PRIVATE
4  PUBLIC :: casa_1, hotel_rs
5  REAL :: casa_1, fazenda_rs
6  INTEGER :: apto_1, hotel_rs
7    .....
8  END MODULE change_ac
```

Neste caso uma unidade de programa, que use este módulo, não terá acesso as variáveis `fazenda_rs` e `apto_1`. Mas, terá acesso as variáveis `casa_1` e `hotel_rs`. Sobre módulos existem ainda outras características interessantes, mas isto que vimos já é o suficiente para mostrar a sua potencialidade.

Assim, chegamos ao final deste minicurso de Introdução ao Fortran 90/95. É evidente que o que foi apresentado pode ser aprofundado, principalmente sobre o último assunto: módulos e interfaces. Muitas outras novas instruções escaparam ao minicurso (por motivo óbvio!), tais como as instruções `FORALL` (específica para processamento paralelo), `WHERE`, `TYPE`, `CASE`, `POINTER` e `TARGET`, que entre outras, tornaram a linguagem Fortran mais poderosa ainda. A proposta inicial era de apresentar alguns avanços que a linguagem Fortran sofreu nestes últimos anos e acredito ter alcançado o objetivo. Agora, quando fores usar a linguagem Fortran, já sabes que a mesma não "morreu", como muitos apregoam. Pelo contrário, ela é constantemente atualizada e está, mais do que nunca, forte no seu principal uso: como ferramenta do meio científico.

Agora, já mais embasado, é interessante visitar o site (em inglês)

http://www.ibiblio.org/pub/languages/fortran/ch1-2.html,

que traz um texto, de Craig Burley, comparando as linguagens C[16] e Fortran 90/95. Vale a pena !!!!

★ **Lista de Exercícios 3**. Ufa!! é a última.

APÊNDICE A

A Tabela ASCII de Caracteres

A ASCII (*American Standard Code for Information Interchange*) é uma tabela de padrões de caracteres, reproduzida parcialmente a seguir. Para visualizar a tabela completa ASCII e outras, acesse o endereço eletrônico: *http://www.lookuptables.com/*.

Tabela A.1 – Tabela ASCII de Caracters

Binário	Decimal	Caracter ASCII	Binário	Decimal	Caracter ASCII
...	...	...	0100 0001	65	A
0010 0000	32	(espaço)	0100 0010	66	B
0010 0001	33	!	0100 0011	67	C
0010 0010	34	"	...	...	...
0010 0011	35	#	0101 1000	88	X
0010 0100	36	$	0101 1001	89	Y
0010 0101	37	%	0101 1010	90	Z
...	...	...	...	...	...
0011 0000	48	0	0110 0001	97	a
0011 0001	49	1	0110 0010	98	b
0011 0010	50	2	0110 0011	99	c
0011 0011	51	3	0110 0100	100	d
0011 0100	52	4	...	...	...
0011 0101	53	5	0111 1000	120	x
0011 0110	54	6	0111 1001	121	y
0011 0111	55	7	0111 1010	122	z
0011 1000	56	8	...	...	...
0011 1001	57	9	0111 1101	125	}
...	...	...	0111 1110	126	$\simeq$

Fonte: *http://en.wikipedia.org/wiki/ASCII*, ou em *http://www.lookuptables.com/*.

APÊNDICE B

Os tipos de Dados do Fortran 95

Tabela B.1 – Os tipos de Dados (intrínsecos) suportados pelo Fortran 95, baseado no compilador G95.

Tipo de variável (dado)	**Parâmetro de Representação** (em Bytes – **KIND**)	**Observações**
INTEGER	1	Intervalo: -127 até 127
INTEGER	2	Intervalo: -32.767 até 32.767
INTEGER	4*	Intervalo: $-2.147.483.647$ até 2.147.483.647
INTEGER	8	Intervalo: $-9.223.372.036.854.775.808$ até 9.223.372.036.854.775.807
REAL	4*	Intervalo: 1.18×10^{-38} até 3.40×10^{38} Precisão: 7–8 dígitos decimais
REAL	8	Intervalo: 2.23×10^{-308} até 1.79×10^{308} Precisão: 15–16 dígitos decimais
REAL	10	Intervalo: 10^{-4931} até 10^{4932} Precisão: aproximadamente 19 dígitos decimais
COMPLEX	4*	Intervalo: 1.18×10^{-38} até 3.40×10^{38} Precisão: 7–8 dígitos decimais
COMPLEX	8	Intervalo: 2.23×10^{-308} até 1.79×10^{308} Precisão: 15–16 dígitos decimais
COMPLEX	10	Intervalo: 10^{-4931} até 10^{4932} Precisão: aproximadamente 19 dígitos decimais
LOGICAL	1	Valores: **.TRUE.** e **.FALSE.**
LOGICAL	2	Valores: **.TRUE.** e **.FALSE.**
LOGICAL	4*	Valores: **.TRUE.** e **.FALSE.**
LOGICAL	8	Valores: **.TRUE.** e **.FALSE.**
CHARACTER	1*	Conjunto de códigos da ASCII Precisão: 1 caracter

Obs: * indica o caso *default*, isto é, assume-se o valor na ausência de indicação.

APÊNDICE C

Glossário

ATRIBUIÇÃO (assignment)
uma instrução na forma *variável = valor numérico* ou *expressão*. É designado ou identificado pelo símbolo de igualdade (=). Por exemplo, `z = 4`, indica que o valor inteiro 4 é atribuído a variável `z`. Outro exemplo: `x = y**2 + 2.0*y`, cujo valor só será atribuído a variável `x` após a solução numérica da expressão a direita.

DISPOSITIVO (device)
um meio material (ou equipamento), tal como uma unidade de disco (HD, disquete, CD,...), teclado, monitor, impressora, ou até mesmo a memória do computador.

INSTRUÇÃO (statement)
uma seqüência de *tokens*, ou seja, uma seqüência de códigos internos que substituem uma palavra reservada ou declaração de programa em uma linguagem de alto nível. P. ex., o `WRITE`, substitui uma instrução de máquina para imprimir, por intermédio de um dispositivo, o conteúdo contido em um referido endereço de memória.

INSTRUÇÃO DE ATRIBUIÇÃO (assignment statement)
é o mesmo que atribuição.

PROCEDIMENTO (procedure)
um cálculo que pode ser invocado durante a execução de um programa. E pode ser uma sub-rotina ou uma função. Pode ser um procedimento intrínsico (p. ex., `COS` ou `EXP`), um procedimento externo, um procedimento módulo, um procedimento interno (subprograma que vem após um `CONTAINS`), um procedimento mudo, ou uma função.

APÊNDICE D

Como Abordar um Problema de Programação

Este texto foi adaptado, com autorização do autor, do artigo *Como Abordar um Problema de Programação*, de Vinícius José Fortuna, obtido no endereço:
http://olimpiada.ic.unicamp.br/programacao/programacao/dicas

D.1 ANALISE O PROBLEMA (E PROJETE SEU PROGRAMA) ANTES DE PROGRAMÁ-LO

Nunca inicie a programar a partir do nada. É aconselhável esquematizar alguns pseudo-códigos (algoritmos) explicando o que o programa vai fazer (em um nível mais elevado) antes de escrever o programa. A exceção é quando se trata de um código que você já escreveu diversas vezes, por exemplo, encontrar um elemento em um vetor ou determinar se um número é par ou ímpar.

Ao escrever um programa é importante que se tenha pensado muito nele antes, com o objetivo de visualizá-lo como um todo. Criando um rascunho prévio do programa, podem aparecer várias abordagens do problema e as dificuldades ficam mais fáceis de serem superadas. Assim, esquematizar o programa ajuda a fixar exatamente o que se deseja e economiza-se tempo em frente ao monitor na tentativa de escrever um programa que execute o desejado.

D.2 ESCREVA UM CÓDIGO LEGÍVEL

Escrever um código legível é muito importante para facilitar o entendimento de um programa. Até para o próprio criador do código. Em programa claro e auto-explicativo fica mais difícil se perder e torna muito mais fácil a depuração.

D.2.1 Comente seu código enquanto escreve, não depois

Comentários são ferramentas muito úteis para tornar o código mais legível. É interessante comentar tudo que não seja muito claro. Não comente algo que seja óbvio, como por exemplo

```
i = 0  ! Atribui o valor 0 a variável i
```

Comente algo como:

```
x = LEN(frase) - LEN(frase)/2 ! x recebe a posicao para frase ficar centralizada
```

Em programas muito grandes ou complicados, é interessante criar um cabeçalho comentado em cada função, definindo exatamente o que espera-se que ela faça, quais suas entradas e quais suas saídas. O pseudo-código rascunhado pode ser muito útil para isso. Agindo assim, não se precisa ler diversas linhas de código para saber o que uma função faz.

É recomendável que se escreva os comentários enquanto se escreve o programa, porque é menos provável que se escreva alguma coisa útil ou significativa depois. Escreva enquanto programa e seus comentários serão muito mais completos.

D.2.2 Utilize margens e indentação apropriadamente

A cada novo loop, expressões condicionais, definição de funções e blocos de comandos, seu código deve ser indentado um nível mais à direita (pressione [TAB] ou a barra de espaço algumas vezes). Esteja certo de voltar ao nível de indentação anterior quando terminar o bloco.

Linhas em branco também são muito úteis para aumentar a legibilidade do seu código. Umas duas linhas entre as definições de funções e procedimentos e uma linha entre a definição de variáveis e o código irão separar claramente cada parte, o que torna a identificação delas mais rápida. Isso torna o código bem mais claro.

D.2.3 Use nomes sugestivos para variáveis, funções e procedimentos

O código fica incrivelmente mais difícil de ser depurado quando variáveis importantes se chamam p, t, ma1, qq, e assim por diante. Deve-se sempre utilizar nomes sugestivos para as variáveis, funções e procedimentos. O nome deve dar idéia do que a variável representa ou o que a função ou procedimento fazem. Por exemplo, se você quer armazenar o número de alunos em uma variável, pode-se usar num_alunos. Se for uma função que calcula o salário médio, pode-se nomeá-la calc_SalarioMedio().

D.2.4 Utilize funções e procedimentos curtos e objetivos

Evite sempre funções/procedimentos grandes que englobem todo tipo de processamento. Separe algoritmos distintos em suas próprias funções/procedimentos. Projete sua grande função/procedimento em várias pequenas, de forma que seu programa fique mais fácil de ler e entender.

Dessa forma, cada parte do seu programa fica bem definida e torna-se muito mais fácil escrevê-lo, pois pode-se fazê-lo passo a passo. Dessa forma, a cada parte que se termina,

pode-se verificar se ela está correta. Além disso a localização de um problema no programa também fica facilitada, pois ele se restringirá a um bloco menor de código.

Conclusão:

Lembre-se que a maior parte do tempo que se gasta programando é corrigindo e modificando código existente. Relativamente pouco tempo é realmente utilizado para adicionar coisas novas. Isso significa que você gastará muito tempo lendo o seu código, então faz sentido gastar algum tempo aprendendo a escrever um código legível. Código legível é fácil de escrever, fácil de depurar e fácil de manter. Você realmente sai ganhando!

D.3
SE ESTIVER CONFUSO NA HORA DA DEPURAÇÃO

Se você estiver confuso ao tentar procurar algum problema no seu programa, tente explicá-lo para você mesmo. Dessa forma é possível notar inconsistências ou fugas ao algoritmo planejado.

Caso isso não resolva, pode-se tentar executar o programa no papel. Isso se aplica tanto a códigos que você escreveu e não está mais entendendo como a códigos pegos de outros. Funciona da seguinte maneira: Pegue uma folha em branco e liste todas as variáveis usadas no programa. Siga linha por linha do código, escrevendo o valor das variáveis enquanto elas mudam, como se você fosse o computador. Pode-se usar uma calculadora para ajudar nas contas. Anote todas as saídas em uma folha à parte. Após algumas poucas iterações a estrutura básica do algoritmo e sua intenção devem ficar claras. Tome cuidado, pois nem sempre o código funciona do jeito que nós pensamos que funciona.

D.4
GUIA PRÁTICO PARA RESOLUÇÃO DE PROBLEMAS DE PROGRAMAÇÃO

1) Entender o problema

Esteja certo de que tenha entendido o problema;
O que é a entrada?
O que é a saída?

2) Resolver o problema à mão

Resolva pequenas instâncias do problema à mão;
O que acontece?
Pense em casos variados;
Pense em como (qual algoritmo) você utilizou para resolver o problema.

3) Definir o algoritmo

Defina precisamente o algoritmo a ser utilizado
Rascunhe as etapas do programa

4) Programar
Como escrever o algoritmo na linguagem utilizada?
Que estrutura de dado utilizar?1
Divida o programa em partes menores (modularizar);
Escreva um programa de fácil leitura;
Pense nos casos patológicos.2

4) Depurar
Explique o programa para si mesmo;
Por que funciona?
A leitura de dados está sendo feita corretamente?
Variáveis inicializadas?
Verificar casos patológicos;
Localizar o erro restringindo os blocos de códigos (cercando o erro)
Comandos e loops aninhados corretamente?

Observações:

1) Que estrutura utilizar?

Qual a melhor forma de representar as variáveis do problema. Variáveis simples? Vetores? Matrizes? Registros? Alguns vetores? Vetores de registro? Registros de vetores? São muitas as estruturas utilizáveis. Deve-se escolher uma que seja conveniente e que não venha trazer complicações mais adiante.

2) Pense nos casos patológicos

Os casos patológicos ocorrem quando a propriedade que seu programa utiliza não vale para alguns valores. Normalmente são o zero, um, valores iniciais ou finais. Por exemplo, em uma função que calcula a potência de um número n pelo expoente e. Para isso pode-se multiplicar o número n e vezes. Nesse caso pode-se ter problemas quando o valor de e for zero, caso que deve ser tratado especialmente (considerando a resposta padrão como 1, por exemplo). Para ilustrar melhor, imagine o caso em que deseja-se verificar se um vetor está ordenado em ordem não-decrescente. Para isso basta verificar se v[n]<=v[n+1] para todos os elementos, exceto o último, pois para ele essa propriedade não tem sentido. Os casos patológicos são causa de grande parte dos problemas, especialmente quando se trabalha com ponteiros.

Referências www.gamedev.net Skiena, Steven S. "The Algorithm Design Manual", Telos, 1997

APÊNDICE E

O LaTeX e este Livro

Com o objetivo didático e, também, de incentivo ao uso do LaTeX, serão apresentados algumas das linhas de programação em LaTeX, que foram utilizadas na elaboração deste livro.

Trabalho com LaTeX desde 1986, e as distribuições que eu utilizo atualmente são o *tetex* no Linux e *MikTeX* no Windows. Ambas funcionam perfeitamente, embora eu prefira a do Linux, por questão puramente ideológica. Como referido no Capítulo 1, o LaTeX é distribuído gratuitamente, mas não é de código aberto, embora as classes (**.cls**) e pacotes (**.sty**) o sejam. Elas são *freeware*. Para editar os documentos utilizo no Linux, o *kile* e no Windows, o *WinShell*, quem podem ser obtidos gratuitamente, respectivamente, nos endereços *(http://kile.sourceforge.net/)* e *(http://www.winshell.org)*. Esta escolha é uma questão pessoal, pois existem outros tantos, e estão listados no endereço *http://www.ctan.org*.

Para abrigar a maioria dos ambientes, comandos e instruções de formatação, que serão descritos abaixo foi criado o pacote **cfglivrof95.sty**, que significa ***c**on**f**i**g**urações do **livro** sobre **F**ortran **95***. Outros foram adicionados no documento fonte principal. É importante ressaltar que constarão somente as formatações que julgo sejam válidas para um aprendizado diferenciado sobre LaTeX. As formatações simples do tipo, distância entre linhas, tamanho de papel e de margens, entre outras, foram omitidas para não tornar enfadonho e longo este capítulo do Apêndice, e se encontram em qualquer texto sobre o assunto.

E.1 SOBRE O TEXTO

A fonte adotada no texto foi a *palatino*, tamanho 10pt, habilitada pelo pacote **palatino.sty**. Para aproveitamento do papel disponível na gráfica, o tamanho do papel escolhido foi de 21×25 cm, e deixando uma margem segura para corte foi adotado $20,7 \times 24,7$ cm, configurado no LaTeX pelo pacote **geometry.sty**.

Para possibilitar a digitação direta dos caracteres acentuados e cedilhados foram utilizados os pacotes e opções (entre colchetes) **[latin1]inputenc.sty** e **[T1]fontenc.sty**, sendo que este último possibilita usar no comando **hyphenation** palavras acentuadas e cedilhadas, permitindo assim melhor controle sobre a separação silábica.

OS CAPÍTULOS

Os títulos dos capítulos foram formatados com o pacote **fncychap.sty**, com a opção **Lenny**, desenvolvido por Ulf A. Lindgren. Para obter a formatação desejada as seguintes linhas, no pacote,

```
\ChNameVar{\fontsize{14}{16}\usefont{OT1}{phv}{m}{n}\selectfont}
\ChNumVar{\fontsize{60}{62}\usefont{OT1}{ptm}{m}{n}\selectfont}
\CNV\FmN{\@chapapp}\space\CNoV\thechapter%
```

foram substituídas pelas linhas

```
\ChNameVar{\color{branco}\fontsize{12}{14}\usefont{OT1}{phv}{m}{n}
\bfseries\selectfont}
\ChNumVar{\color{preto}\fontsize{60}{62}\sffamily\bfseries\selectfont}
\hspace{-.05cm}\CNV\FmN{\@chapapp}\space\CNoV\thechapter%
```

E foi introduzida no pacote, antes do \parbox[b]{\textwidth}{, a linha:

```
{\color{cinza4}\rule{1.5\textwidth}{.69cm}}\vspace{-1.53cm}
```

para gerar o traço horizontal cinza claro, que transpõe a margen direita.

Também foram alterados os comandos, estes localizados no preâmbulo do documento fonte principal:

```
\ChTitleVar{\Huge\bfseries\sffamily}
\ChRuleWidth{0pt}
\ChNameUpperCase
\ChTitleUpperCase
```

AS SEÇÕES

Os títulos das seções dos capítulos foram formatados com o pacote **titlesec.sty**, com as seguintes opções:

```
\titleformat{\section}[block]{\sffamily\Large\bfseries}
{{\bfseries\color{cinza4}\thesection}}{.5em}
{\\[.1mm]\sffamily\Large\bfseries\MakeUppercase}
% Avança o título para a próxima linha
\titlespacing{\section}{-.8cm}{.50cm}{1pc}
```

E.2
A DEFINIÇÃO DOS TONS DE CINZA

Com o pacote **color.sty** foi possível definir os seguintes tons de cinza:

```
\definecolor{cinza1}{cmyk}{0,0,0,0.1} % fraco +++
\definecolor{cinza15}{cmyk}{0,0,0,0.15} %
\definecolor{cinza2}{cmyk}{0,0,0,0.2} %
```

```
\definecolor{cinza3}{cmyk}{0,0,0,0.3} %
\definecolor{cinza4}{cmyk}{0,0,0,0.4} %
\definecolor{cinza5}{cmyk}{0,0,0,0.5} % medio
\definecolor{cinza6}{cmyk}{0,0,0,0.6} %
\definecolor{cinza7}{cmyk}{0,0,0,0.7} %
\definecolor{cinza8}{cmyk}{0,0,0,0.8} %
\definecolor{cinza9}{cmyk}{0,0,0,0.9} % forte +++
\definecolor{preto}{cmyk}{0,0,0,1}
\definecolor{branco}{cmyk}{0,0,0,0}
```

Também foram definidos o branco e o preto na base CMYK, no lugar da base RGB. Isto é importante no momento da impressão do livro na gráfica. Se definirmos tons de cinza com RGB, estaremos adotando pigmentos "coloridos" e a impressão sairá mais cara. O uso das definições de tons de cinza no texto é pelo comando \ **color{cinza4}** ou \ **textcolor{cinza4}**, como aparecerão nas outras definições de comandos a seguir. Um exemplo do uso é:

O texto ficará {\color{cinza4} cinza} assim. $\Longrightarrow$ O texto ficará cinza assim.

E.3
A NOTA DE MARGEM

A nota de margem para chamar atenção aos elementos novos do **Fortran 90**, foi gerada com o seguinte comando:

fortran 95

```
\newcommand{\FORTRANN}[1]{%
\textcolor{cinza4}{\bfseries #1}\marginpar{%
\sffamily \textcolor{cinza4}{\tiny fortran}\\[-.25cm]%
\bfseries\Large\xspace\textcolor{cinza4}{90}%
}%
}
```

E.4
AS NOTAS DE OBSERVAÇÕES NO TEXTO

As notas utilizadas ao longo do texto, como por exemplo o da página 2, foram geradas com o seguinte comando:

```
\newcommand{\NOTA}[2]{
\begin{flushright}
\vspace*{-.4cm}%
\begin{minipage}[t]{0.8\textwidth}
{\small\bfseries\sffamily\color{cinza4} #1}\\[-.2cm]
{\color{cinza4}\rule{\textwidth}{.1mm}}
\\[-.05cm]\color{cinza4}\scriptsize\sffamily #2
```

```
{\color{cinza4}\hrulefill}
\end{minipage}
\end{flushright}\vspace{.1cm}
}
```

E.5 OS QUADROS DAS INSTRUÇÕES FORTRAN 95

Os quadros que contêm as formas das sintaxes das instruções Fortran 95, como o exemplo abaixo,

```
WRITE([UNIT=]<unidade>,[FMT=]<formato>,[ADVANCE=<modo>])
```

foram introduzidas pelos ambientes

```
\begin{sintaxebox}
\begin{Verbatim}[fontfamily=tt,fontseries=b,commandchars=\\\{
\},numbers=none,xleftmargin=5mm,codes={\catcode'$=3\catcode'^=1}]
WRITE([UNIT=]<unidade>,[FMT=]<formato>,[ADVANCE=<modo>])
\end{Verbatim}
\end{sintaxebox}
```

sendo que o ambiente **sintaxebox**, é gerado pelas seguintes linhas:

```
\newlength\Linewidth
\def\findlength{\setlength\Linewidth\linewidth
\addtolength\Linewidth{-4\fboxrule}
\addtolength\Linewidth{-3\fboxsep}
}
 \newenvironment{sintaxebox}{\par\centering\begingroup%
  \setlength{\fboxsep}{5pt}\findlength%
  \setbox0=\vbox\bgroup\noindent%
  \hsize=1.0\Linewidth%
     \begin{minipage}{1.0\Linewidth}\small}%
    {\end{minipage}\egroup%
  \vspace{6pt}%
  \noindent\textcolor{cinza4}{\fboxsep2.5pt%
    {\fboxsep5pt\colorbox{cinza4}{\normalcolor\box0}}}%
  \endgroup\par\addvspace{6pt minus 3pt}\noindent%
  \normalcolor\ignorespacesafterend}
\let\Sintaxebox\sintaxebox
\let\endSintaxebox\endsintaxebox
```

O pacote **fancyvrb.sty** é que habilita o ambiente **Verbatim**.

E.6
OS EXEMPLOS DE PROGRAMAS FORTRAN 95

Os exemplos de programas Fortran 95, como o do primeiro programa,

Programa 1–2: O primeiro programa

```
WRITE(*,*) 'Ola mundo externo ....'
END
```

foram introduzidos da seguinte forma:

```
\begin{programa}{O primeiro programa}
\fvset{fontsize=\scriptsize,numbers=none,fontfamily=tt,
fontseries=b}
\VerbatimInput[xleftmargin=12.5mm]{ex_primeiro.f90}
\end{programa}
```

no qual foi utilizado novamente o pacote **fancyvrb.sty**, com a opção do comando **VerbatimInput**, que anexa um arquivo no texto, neste caso o **ex_primeiro.f90**.

O ambiente **programa** foi gerado pelas seguintes linhas:

```
\newcounter{np} % novo contador
\setcounter{np}{1}
\newenvironment{programa}[1]{\par\centering\begingroup%
  \setlength{\fboxsep}{5pt}\findlength%
  \setbox0=\vbox\bgroup\noindent%
  \hsize=1.0\Linewidth%
    \begin{minipage}{1.0\Linewidth}\footnotesize
          {\bfseries Programa~\thechapter--\arabic{np}:}~#1}%
  {\end{minipage}\egroup%
  \vspace{6pt}%
  \noindent\textcolor{cinza1}{\fboxsep2.5pt%
   {\fboxsep5pt\colorbox{cinza1}{\normalcolor\box0}}}%
  \endgroup\par\addvspace{6pt minus 3pt}\noindent%
  \normalcolor\ignorespacesafterend\refstepcounter{np}}
\let\Programa\programa
\let\endPrograma\endprograma
```

Para os exemplos de programas que apresentam continuação na página seguinte, alterou-se o nome do ambiente para **contprog** e substitui-se a linha

```
{\bfseries Programa~\thechapter--\arabic{np}:}~#1}%
```

pela

```
{\bfseries Continuação do Programa~\thechapter--\arabic{np}} ...}%
```

E.7
OS MINI-SUMÁRIOS DOS CAPÍTULOS

A idéia deste item no início de cada capítulo tem dois objetivos. Primeiro, informar ao leitor o que encontrará no decorrer do mesmo e, segundo, transcrever uma frase de um cientista conhecido, para possibilitar uma reflexão. O LATEX proporciona o pacote **minitoc.sty** para gerar mini-sumários, mas a opção em criá-los manualmente foi para ter controle sobre todo o conteúdo deste item, como por exemplo, filtrar os itens que constarão no mini-sumário.

O comando **INIZIO{#1}{#2}** gera o mini-sumário.

```
\newcommand{\INIZIO}[2]{\vspace{-.8cm}
{\color{cinza6}%
\rule{\textwidth}{.3mm}\\[.2cm]\noindent\begin{minipage}[t]{0.5\textwidth}%
\tiny
#1
\vspace{.1cm}\mbox{}
\end{minipage}
\hspace{0.05\textwidth}
\begin{minipage}[t]{0.4\textwidth}
\scriptsize\noindent\sffamily
#2
\vspace{.1cm}\mbox{}
\end{minipage}\\
\rule{\textwidth}{.3mm}
}
}
```

em que o primeiro argumento (**#1**) é a parte da lista de conteúdos do referido capítulo, copiada e editada do arquivo **nome_livro.toc**, gerado pelo LATEX. O segundo argumento (**#2**), contém a frase para reflexão.

E.8
AS REFERÊNCIAS BIBLIOGRÁFICAS

REFERÊNCIAS BIBLIOGRÁFICAS

[1] KNUTH, D. E. *The TEXbook*. Reading, Massachusetts: Addison-Wesley, 1984.

[2] LAMPORT, L. *LATEX: A document preparation system*. Reading, Massachusetts: Addison-Wesley, 1986.

[3] GOOSSENS, M., MITTELBACH, F., SAMARIN, A. *The LATEX companion*. Reading, Massachusetts: Addison-Wesley, 1994.

[4] KOPKA, H., DALY, P. W. *A guide to LATEX 2ε: Document preparation for beginners and advanced users*. Harlow, England: Addison-Wesley, 1997.

[5] http://www.miktex.org/. Site com links e distribuição gratuita de pacotes, para Windows, como: LaTeX, WinShel, GhostView, e outros – Último acesso: 02 de junho de 2001.

[6] CHAPMAN, S. J. *Fortran 90/95 for scientists and engineers*. Boston, Massachusetts: WCB McGraw-Hill, 1998.

[7] http://www.lahey.com/. Último acesso: 03 de junho de 2001.

[8] http://www.pgroup.com/. Último acesso: 03 de junho de 2001.

[9] Lahey Computer Systems, Inc., 865 Tahoe Boulevard – P.O. Box 6091 – Incline Village, NV 89450-6091. *Lahey/fujitsu fortran 95 express – user's guide/linux edition*, 1999. Revision A.

[10] ELLIS, T., PHILIPS, I. R., LAHEY, T. M. *Fortran 90 programming*. Harlow, England: Addison-Wesley, 1998.

[11] REID, J. The new features of fortran 2003. *ISO/IEC JTC1/SC22/WG5 N1579*, Benson, Oxon OX10 6LX, UK, v. 1, p. 1–38, 2003.

[12] DE ALENCAR PRICE, A. M., TOSCANI, S. S. *Implementação de linguagens de programação: compiladores*. Number 9 in Livros didáticos. Porto Alegre: Editora Sagra Luzzatto, 2000.

[13] http://www.linux.org/. Último acesso: 03 de junho de 2001.

[14] Cray Research, Inc., Eagan – USA. *Optimizing application code on cray pvp systems: Tr–vopt 2.0(a), volume i*, 1996.

[15] Cray Research, Inc., 2360 Pilot Knob Road – Mendota Heights, MN 55120, USA. *Cf90^{TM} commands and directives – reference manual*, 1994. SR-3901 1.1.

[16] KERNIGHAN, B. W., RITCHIE, D. M. *C, a linguagem de programação: Padrão ansi*. Rio de Janeiro: Campus, 1989.