MINICURSO
Introdução ao Fortran 90/95

Gilberto Orengo
Centro Universitário Franciscano

Semana Acadêmica
6-8 de junho de 2001

Este Material Didático pode ser manuseado e reproduzido
livremente, desde que seja mantida a versão original.

MINICURSO
Introdução ao Fortran 90/95

Centro Universitário Franciscano
Semana Acadêmica
6-8 de junho de 2001

Copyright © 2001 by Gilberto Orengo. This material or parts thereof may be reproduced, since that the original version is kept.

Caro Leitor

Uma vez que tenha lido este material, de reprodução liberada, eu peço que responda através de e-mail, as seguintes questões:

- O que foi difícil de entender?
- O que foi enfadonho?
- O que você ou seus amigos/colegas esperavam?
- Você encontrou algum erro?

Claro, quaisquer outras sugestões são bem-vindas. Este material é o embrião de um futuro livro sobre o uso e manuseio da linguagem Fortran. Uma certeza: serão necessários anos de trabalho, mas a semente já está plantada. Caso queira efetivamente contribuir para a construção deste livro envie um e-mail.

Agradecido!

Gilberto Orengo
Registered Linux User ID: 205346

g.orengo@via-rs.net
orengo@unifra.br

Arquivo mcf90.tex

Conteúdo

1 Considerações Iniciais **1**
1.1 Familiarizando-se com a Terminologia 1
1.2 Como Trabalha a Memória do Computador? 3
1.3 A "Despensa Numérica" no Fortran 90 4
1.3.1 Dados Inteiros - INTEGER 5
1.3.2 Dados Reais ou de Pontos Flutuantes - REAL 5
– Precisão Simples (Single Precision) 5
– Precisão Dupla (Double Precision) 6
1.3.3 Os Números Complexos - COMPLEX 6
1.4 As Unidades de Programa 7
1.5 A Declaração dos Dados 8

2 As Funções Intrínsecas KIND e SELECTED_REAL_KIND **9**
2.1 Selecionando Precisão de Maneira Independente do Processador 10

3 O Uso da Alocação Dinâmica de Memória **13**
3.1 O Atributo ALLOCATABLE e as Declarações ALLOCATE e DEALLOCATE 14
3.2 Quando Devemos Usar uma Array? 15
3.3 Manipulação entre Arrays 16

4 Os Módulos – MODULE **17**
4.1 A Declaração COMMON 17
4.2 A Declaração MODULE 19
4.2.1 Compartilhando Dados usando o MODULE 19
4.3 Os Procedimentos MODULE 20
4.3.1 Usando Módulos para Criar Interfaces Explícitas 20
4.3.2 A Acessibilidade PUBLIC e PRIVATE 22

Apêndice **23**

A A Programação Estruturada no Fortran **23**
A.1 As Sub-rotinas – SUBROUTINE 23
A.2 As Funções – FUNCTION 25

Referências Bibliográficas **27**

Índice **28**

Capítulo 1

Considerações Iniciais

Para início de conversa, é importante salientar que este texto foi integralmente escrito em LaTeX [1]–[5]. Desta forma, é aconselhável quem ainda não manteve contato com LaTeX, que o faça o mais breve possível. Este, é uma ferramenta muito poderosa para uso no meio acadêmico e científico.

Infelizmente, o assunto sobre Fortran 90/95 [6]–[10] é extenso para ser tratado num minicurso de apenas três dias. Sendo assim, entre as várias evoluções sofridas pelo Fortran (relativas ao FORTRAN 77), daremos ênfase a três: a função intrínseca `SELECTED_REAL_KIND`, que permite maior portabilidade entre computadores e compiladores Fortran; a declaração de variáveis `ALLOCATABLE` – que habilita a alocação dinâmica de memória e; as Declarações e Procedimentos do tipo `MODULE` – que, entre outras coisas, substitui com primazia os confusos e perigosos `COMMON`. Como também não haverá tempo para aplicações, apenas para alguns exercícios, uma boa referência sobre o uso da linguagem Fortran em Física encontra-se no livro de DeVries [11].

Será dedicado mais tempo à parte inicial, pois formará a base de todo minicurso. Para maiores informações e futuros avanços no aprendizado é aconselhado o livro do Chapman [6]. Não esqueçam, sempre que estiverem usando um dado compilador Fortran, uma valiosa fonte de informação encontra-se no seu Guia (ou Manual) do Usuário (*User's Guide*) e no Manual de Referência da Linguagem (*Language Reference Manual*).

1.1 Familiarizando-se com a Terminologia

É necessário conhecermos alguns termos usados nesta área da computação. As memórias dos computadores são compostas de milhões de "interruptores eletrônicos" individuais, cada um podendo assumir ON ou OFF ("ligado" ou "desligado"), nunca num estado intermediário. Cada um destes interruptores representa um **dígito binário** (também conhecido como **bit** – de *digit binary*), onde o estado ON é interpretado como o binário 1 e o estado OFF como o binário 0. Diversos bits agrupados juntos são usados para representar o sistema binário de números ou simplesmente o sistema de base dois.

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 daí, temos:

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

1024 KBytes = 1 MByte (1 MegaByte ou 1MB)

1024 MBytes = 1 GByte (1 GigaByte ou 1GB) e, já estamos ouvindo falar em TeraByte,

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

Para compreendermos um pouco mais o sistema binário, busquemos algo familiar, o nosso sistema decimal de cada dia (ou sistema de base 10, $\{0, 1, 2, \ldots, 8, 9\}$). Representamos um número nesta base (como exemplo, o 152) da seguinte forma:

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

E na base numérica do computador ou base 2, $\{0, 1\}$, como representamos um número? Bem, fazemos da mesma forma que na base 10. Vejamos o exemplo do número 101 :

NOTA: A pronúncia dos números na base 2 não é igual ao da base 10; neste caso o número 101 é pronunciado como UM ZERO 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, 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 destes valores são reservados para representar números negativos e a outra metade para os positivos. No caso de 1 Byte (8 bits) é utilizado usualmente para representar números entre -128 e $+127$, inclusive. Um sistema típico para representar os caracteres (de linguagens Não-Orientais) deve incluir os seguintes símbolos:

- As 26 letras maiúsculas (A, B, ..., Z).
- As 26 letras maiúsculas (a, b, ..., z).
- Os dez dígitos (0, 1, ..., 9).
- Símbolos comuns, tais como, " , ( ){ }[ ]!˜ _ @ # $ % ˆ & *.
- Algumas letras especiais ou símbolos, tais como, à ç ë £.

Embora o número total de caracteres e símbolos requeridos é menor do que 256, é usado 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 sendo desenvolvido um outro sistema de código mais geral, chamado *Unicode*, que contempla algumas linguagens orientais.

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, um Pentium©, no caso, é baseado em 32 bits (4 Bytes). Isto será importante mais adiante.

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**.

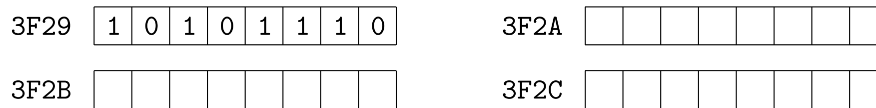
Para finalizar esta breve introdução, se fala muito em processamento paralelo (ou vetorização). O processamento paralelo ocorre quando duas ou mais CPUs trabalham **simultaneamente** na solução de um mesmo problema. Com isto se obtém, na grande maioria das vezes, maior velocidade de processamento computacional. Para fazermos uso desta otimização, é necessário que o compilador Fortran 90/95 nos habilite tal procedimento e que o programa seja feito com este objetivo, i.e., implementando as declarações intrínsecas para o processamento paralelo. Como exemplo de compiladores que habilitam a vetorização temos, entre outros, o *Lahey-Fujitsu Fortran 95–Pro v6.0 for GNU/Linux* da Lahey Computer Systems, Inc. [7] (a versão Windows© não contém esta característica) e o *PGHPF* da Portland Group [8], este por sinal é um excelente compilador.

CPU: Central Processor Unit ou, no bom português, *Unidade Central de Processamento*

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

1.2 Como Trabalha a Memória do Computador?

Neste exemplo hipotético, o tamanho de uma 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.

Sistema Hexadecimal: 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, é 010001001100_2 no sistema decimal. Para evitar esta difícil manipulação, uma alternativa é quebrar o número binário em grupos de 3 e 4 bits e com isso obter novas bases, base 8 (**series octal**) ou base 16 (**series hexadecimal**), respectivamente. Compreendermos a última, 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: $0, 1, \ldots, 8, 9$ e de A, B, ..., E, F. Assim, $9_{16} = 9_{10}$; $A_{16} = 10_{10}$; $B_{16} = 11_{10}$; e assim por diante. Nós 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 90 possibilita substituí-las por nomes (em inglês).

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

Todas as CPU tem um **conjunto de instruções** (ou linguagem própria da máquina) para sua manipulação, que ela e nós "compreendemos". De maneira geral, todos os programas Fortran 90 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 3AF7". Esta é basicamente a maneira de como os programas trabalham.

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

```
LDA '3F2C'  ⇒  carregue (ou busque) os conteúdos de 3F2C
ADD '3F29'  ⇒  adicione estes conteúdos em 3F29
LDA '3F2A'  ⇒  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, então trabalhando na IBM, que propôs que um método econômico e eficiente

de programar deveria ser desenvolvido. 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:

$$\mathtt{K} = \mathtt{I} + \mathtt{J}\,.$$

Resumindo, um **compilador** tem a tarefa de converter um procedimento, como o da expressão acima, em instruções de código assembler. Um compilador Fortran é evocado por uma palavra chave, que depende de compilador para compilador. Por exemplo, no *Lahey/Fujitsu for GNU/Linux* é `lf95`, isto é, no terminal do GNU/Linux(†)[12] digitamos esta palavra chave seguida do nome do programa em Fortran 90/95:

```
home/orengo#> lf95 nome_programa.f90
```

Este comando:

1. verifica a sintaxe no programa,
2. gera um código objeto (programa objeto),
3. repassa o código objeto para o *"linkador"*(‡), que anexa bibliotecas (sistema, E/S, etc.) e gera um executável, com um nome *default* chamado `a.out`.

E/S = **E**ntrada/**S**aída. Em inglês, I/O = Imput/Output

Para dar um nome diferente para o arquivo executável é possível usar uma opção do compilador. Assim, temos para o exemplo acima

```
home/orengo#> lf95 -o nome_de_saida nome_programa.f90
```

Para outras opções do compilador, consulte o manual do Usuário, ou no terminal, digite:

```
home/orengo#> man lf95
```

para ler o manual.

O passo 3, acima, anexa ao código, entre outras coisas, cálculos matemáticos, entrada de dados via teclado e saída de resultados (dados) via monitor, por exemplo. Os arquivos executáveis são específicos para cada processador e/ou sistema operacional, i.e., código compilado num Intel Pentium não será executado numa Estação Sun SPARC e vice-versa. Assim, quando trocarmos de plataforma (processador e/ou sistema operacional), devemos compilar novamente o código.

Desta forma, encerramos uma breve discussão sobre os princípios básicos da manipulação de memória e a função de um compilador. A seguir veremos como funciona a "despensa numérica" do computador, isto é, como ele armazena números inteiros e reais (e por conseqüência os complexos).

1.3 A "Despensa Numérica" no Fortran 90

Em geral, são dois os tipos de números (dados) usados nos programas em Fortran 90: inteiros e reais, estes são conhecidos também como pontos flutuantes.

(†)**IMPORTANTE:** se você não trabalha com o sistema operacional GNU/Linux, um conselho: COMECE JÁ !!! É gratuito e não por isso ineficiente, pelo contrário, é altamente estável.

(‡)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"*.

1.3.1 Dados Inteiros - INTEGER

Os dados inteiros são armazenados "exatamentes" 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, que poderá ser de 1, 2, 4 ou 8 Bytes. O mais comum de ocorrer nos computadores atuais é de 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 pode ser representado num computador. Normalmente, o menor número inteiro que pode ser armazenado em n-bits inteiros é:

$$\text{Menor Valor Inteiro} = -2^{n-1} \tag{1.1}$$

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

$$\text{Maior Valor Inteiro} = 2^{n-1} - 1 \tag{1.2}$$

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 tentamos usar valores abaixo ou acima destes ocorre um erro chamado de *overflow condition*.

1.3.2 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 (por praticidade) escritos em notação científica. Por exemplo, a velocidade da luz no vácuo é aproximadamente 299.800.000m/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, pois o computador trabalha na base 2. Assim, se N-bits são dedicados para representar (e armazenar) um número real, parte é reservado 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), 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**,

i) 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 algarismos significativos 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 90 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**,

i) Expoente $\Rightarrow n' = 11$ bits (1 Byte)

$2^{n'} = 2^{11\,\text{bits}} = 2^{2048_{10}}$, 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 16 algarismos significativos 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*.

1.3.3 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 $c = a + bi$, onde 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.)`	$1 + 0i$ (real puro)
`(0.7071,0.7071)`	$0.7071 + 0.7071i$
`(1.01E6,0.5E2)`	$1010000 + 50i$
`(0,-1)`	$-i$ (imaginário puro)

Desta forma, o que vimos para os reais, é válido para os complexos. A diferença está no procedimento Fortran, que é feito através da declaração `COMPLEX`, que veremos adiante.

Quando estivermos programando em Fortran 90, 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[(§)]. Então, como poderemos escrever programas que possam ser facilmente portáveis entre processadores diferentes, com tamanho de palavra (*wordsize*) diferentes e assim mesmo funcionarem corretamente? O Fortran 90 possibilita modificarmos a mantissa e o expoente, conforme a conveniência e, com isso também obter 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`, que veremos no próximo capítulo.

[(§)] Exemplos da dependência da combinação Processador/Compilador: num Supercomputador Cray T90/CF90[13]– [14], a precisão simples é 64 bits e a dupla 128 bits; já num PC/Lahey Fortran 90, a precisão simples é 32 bits e a dupla 64 bits.

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 (*Main Program*)
- Sub-Programa Externo FUNCTION
- Sub-Programa Externo SUBROUTINE
- Unidade de Programa BLOCK DATA
- Unidade de Programa MODULE

A seguir veremos o primeiro tipo. Os subprogramas SUBROUTINE e FUNCTION estão descritos no Apêndice A. A 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 executável, no programa principal, e finaliza com uma declaração STOP, localizado em qualquer lugar do programa ou com a declaração END do programa principal. A forma de um programa principal é

```
[PROGRAM] [nome_do_programa]        (1º)
[USE nome_do_use]                   (2º)
[IMPLICIT NONE]                     (3º)
[declaração dos dados ]             (4º)
[declarações executáveis]
[subprogramas internos]
END [nome_do_programa]              (último)
```

onde, os colchetes indicam que a declaração é opcional. Nos parênteses, está indicado a ordem obrigatória na seqüencia das declarações.

Vejamos um exemplo de programa em Fortran 90/95, que transforma o valor do ângulo em graus para radianos:

```
PROGRAM graus_to_rad
IMPLICIT NONE
!
! Este programa converte angulos em graus para radianos
!
REAL(KIND=8), PARAMETER :: pi=3.141592653589793_8
REAL(KIND=8) :: theta, rad
    WRITE(*,'(" Indique um angulo em graus:  ")'&
    ,ADVANCE='NO')
    READ(*,*)theta
    rad = theta*pi/90.0_8  ! Aqui ocorre a conversao
    WRITE(*,*) 'O angulo ',theta,', em graus, vale',rad,' radianos'
    WRITE(*,*) 'cos(theta) = ',cos(rad)
END PROGRAM graus_to_rad
```

IMPORTANTE: é uma boa prática de programação colocar declaração PROGRAM (sempre na primeira linha) seguido de um nome. Já o IMPLICIT NONE obriga-nos a declarar todas

a variáveis do problema, ajudando com isso a depurar eventuais erros tipo, de escrita ou de dupla declaração. Outra boa prática de programação: comente o máximo possível o seu programa. Isto é feito com o caracter !, colocado no início do comentário. O uso de comentários evita o esquecimento do significado de cada variável, ou ainda para que serve o programa e/ou as sub-rotinas, fato que é comum com o passar do tempo (*O nosso cérebro nem sempre funciona ....*). O caracter &, colocado no final da linha (e/ou no início da outra linha) indica que a linha continuará na linha seguinte. Se um nome, palavra-chave, constante ou rótulo é quebrado por um &, o primeiro caracter não branco da próxima linha deve ser um &, seguido do restante do nome, palavra-chave, constante ou rótulo. O número máximo de continuações é de 39 linhas. O & não funciona no interior dos comentários (!).

1.5 A Declaração dos Dados

Em Fortran, todos os dados intrínsecos (ou variáveis) devem ser declarados. São cinco os tipos de dados intrínsecos: `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL` e `CHARACTER`. O tipo de declaração `DOUBLE PRECISION`, disponível no FORTRAN 77, é ainda suportado pelo Fortran 90/95, mas é considerado um subconjunto (ou um tipo – *kind*) do `REAL`. Os dados inteiros e reais já foram descritos anteriormente. Os demais tipos de declarações não serão tratados em detalhes aqui.

Vejamos exemplos de declarações, 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 :: 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
```

As declarações acima, por si só se explicam, mas veremos algumas considerações importantes. Inicialmente, 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. As variáveis `a1` e `b1` são *arrays*, i.e., matrizes, com a declaração `DIMENSION` explicitada. No primeiro caso, 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.

★ **Lista de Exercícios 1**, mãos na massa!!!!

Capítulo 2

As Funções Intrínsecas KIND e SELECTED_REAL_KIND

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, podemos 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 2.1 é apresentado os resultados da execução deste programa, em quatro diferentes combinações de Processador/Compilador.

Tabela 2.1: 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 !!!!

2.1 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 2.2 descreve estas funções.

Tabela 2.2: 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 2.2, 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 3

O Uso da Alocação Dinâmica de Memória (ALLOCATABLE)

Vimos no primeiro capítulo como declarar variáveis. Aqui nos deteremos um pouco nas arrays(*), 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:

```
CHARACTER(len=20), DIMENSION(50) :: nomes
```

Isto é, cada elemento desta array deverá ter até 20 caracteres, e endereçado como `nomes(1)`,

(*) 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.

`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 declarar-mos 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[(†)], 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.

3.1 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)
```

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,

[(†)] **Open Source**: é uma boa prática abrirmos o código fonte de nossos programas, através da licença GPL–General Public License[12]. Eles se tornarão mais eficientes, pois outros programadores poderão fazer alterações e nos avisar das mesmas.

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.

3.2 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.

3.3 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 4

Os Módulos – 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.

4.1 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:

$$\texttt{COMMON}\ /r_1/\mathit{lista\ de\ identificadores}_1 \cdots /r_N/\mathit{lista\ de\ identificadores}_N$$

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`.

4.2 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.

4.2.1 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 18 (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.

4.3 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
```

4.3.1 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(*) 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 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.

(*) 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.

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.

4.3.2 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[15] e Fortran 90/95. Vale a pena !!!!

★ **Lista de Exercícios 3**. Ufa!! é a última.

Apêndice A

A Programação Estruturada no Fortran

O Fortran tem um importante mecanismo que permite criar subtarefas (ou subprogramas), que são usados para desenvolver e depurar erros, isoladamente, antes de construir o programa final. Isto constitui a chamada *Programação Estruturada*. Assim, os programadores podem criar cada subtarefa como uma unidade de programa, chamada de **procedimento externo**, e então compilá-las, testá-las e ainda depurar os erros para cada procedimento externo independentemente de todas as outras subtarefas. Depois de aprovada, a subtarefa pode ser agregada ao programa principal.

O Fortran tem dois tipos de procedimentos externos: **sub-rotinas** (`SUBROUTINE`) e **funções** (`FUNCTION`). As sub-rotinas são chamadas pelo respectivo nome, através da declaração `CALL`, podendo retornar múltiplos resultados através de seus argumentos. As funções são ativadas pelo seu nome na expressão e, o seu resultado é um único valor que é usado no cálculo da expressão. Ambos os procedimentos serão descritos rapidamente a seguir. O Fortran também permite procedimentos internos, mas não serão tratados aqui. Para maiores esclarecimentos, pesquise nas ref.[6][10]. Para alimentar a curiosidade, pode-se adiantar que a declaração `CONTAINS` joga um papel imprescindível.

Os benefícios dos subprogramas são principalmente:

1. **Testes Independentes das Subtarefas.** Cada subtarefa pode ser codificada e compilada como uma unidade independente, antes de ser incorporada 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 a mesma subtarefa. 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.

O uso de subprogramas é uma boa prática de programação em programas (códigos) muito grandes.

A.1 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*)
>
> ...
>
> *Declarações*
>
> ...
>
> *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[(*)]. Para chamar uma sub-rotina é usado a declaração CALL, da seguinte maneira:

> CALL *nome_da_sub-rotina* (*lista_de_argumentos*)

onde, 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. Um exemplo simples ilustra melhor o que é uma sub-rotina.

```
1  SUBROUTINE exemplo_sub (lado1, lado2, hipotenusa)
2  ! Calcula hippotenusa
3  IMPLICIT NONE
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(tem)
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). 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 do programa. Esta sub-rotina é usada num programa ou noutra sub-rotina, através da declaração CALL exemplo_sub (lado1, lado2, hipotenusa), como no exemplo abaixo:

[(*)] Uma sub-rotina pode chamar outra sub-rotina, mas não a si mesmo, a menos que seja declarada *recursiva*. Maiores informações sobre sub-rotinas recursivas são obtidas nas ref. [6][10].

```
1  PROGRAM testa_sub
2  IMPLICIT NONE
3  REAL :: s1, s2, hip
4  ...
5      CALL exemplo_sub (s1, s2, hip)
6      WRITE(*,*) 'A hipotenusa eh: ',hip
7      ...
8  ...
9  END PROGRAM test_sub
```

Outras características importantes, tal como alocação de memória automática para arrays, estão descritos detalhadamente nas referências indicadas anteriormente.

A.2 As Funções – FUNCTION

Uma função Fortran é um procedimento que só pode ser ativado em uma expressão pertencente a um comando de programa. A função retorna (resulta) num 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 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*
>
> ...
>
> *Procedimentos Executáveis*
>
> *nome_da_função = expressão*
>
> RETURN
>
> END FUNCTION *[nome_da_função]*

A função definida-usuário (ou simplesmente função) deve ser iniciada com a declaração FUNCTION e finalizada com uma declaração END FUNCTION. O nome da função deve seguir, como nas sub-rotinas, os padrões do Fortran, i.e., 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 declaração 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, CARACTER 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.

```
1  REAL FUNCTION exemplo_func
2  ! Objetivo: calcular um polinomio quadratico do tipo
3  !           a*x**2 + b*x + c
```

```
4  IMPLICIT NONE
5  REAL, INTENT(IN) :: x
6  REAL, INTENT(IN) :: a
7  REAL, INTENT(IN) :: b
8  REAL, INTENT(IN) :: c
9  ! Calcula a expressao
10     exemplo_func = a*x**2 + b*x + c
11 END FUNCTION exemplo_func
```

Esta função produz um resultado real. Observe que o atributo INTENT não é usado com a declaração do nome da função exemplo_func, porque ela sempre será usada somente como saída. Note também que, se não fosse declarada como real, a variável exemplo_func deveria ser declarada no corpo da FUNCTION, como de hábito. Um programa que usa esta função pode ser:

```
1  PROGRAM testa_func
2  IMPLICIT NONE
3  REAL :: exemplo_func
4  REAL :: a, b, c, x
5      WRITE(*,*) 'Entre com os coef. quadraticos a, b e c: '
6      READ(*,*) a, b, c
7      WRITE(*,*) 'Entre com a localizacao na qual quer fazer o calculo: '
8      READ(*,*) x
9      WRITE(*,100) ' Calculo em (',x, ') = ', exemplo_func(x,a,b,c)
10 100 FORMAT(A,F10.4,A,F12.4)
11 END PROGRAM testa_func
```

Note que a função exemplo_func é declarada como tipo real tanto na própria função, como no programa principal da qual é ativada. Para maiores informações, procure pela literatura indicada nas 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 2_ε: 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] DEVRIES, P. L. *A first course in computational physics*. New York: John Wiley & Sons, Inc., 1994.

[12] http://www.linux.org/. Último acesso: 03 de junho de 2001.

[13] Cray Research, Inc., Eagan – USA. *Optimizing application code on cray pvp systems: Tr–vopt 2.0(a), volume i*, 1996.

[14] Cray Research, Inc., 2360 Pilot Knob Road – Mendota Heights, MN 55120, USA. *Cf90^{TM} commands and directives – reference manual*, 1994. SR-3901 1.1.

[15] KERNIGHAN, B. W., RITCHIE, D. M. *C, a linguagem de programação: Padrão ansi*. Rio de Janeiro: Campus, 1989.

Índice

ALLOCATABLE, 13, 14
ALLOCATED, 15
Alocação de Memória, 3
Alocação de Memória Automática, 25
Alocação Dinâmica de Memória, 13, 14
    ALLOCATABLE, 13, 14
Alocação Estática de Memória, 14
Array, 15
    Quando usá-la?, 15
ASCII, 2
Assembler, Código, 3

Bit, 1
Byte, 1
    GByte, 1
    KByte, 1
    MByte, 1
    TByte, 1

CALL, 24
COMMON, 17
    Compartilhamento de Memória, 17
Compartilhamento de Memória, 17
Compilador, 2, **4**
    Lahey, 2
    PGHPF, 2
Complexos, Números, 6
CONTAINS, 19
CPU, 2
Cray, Supercomputador, 6

Declaração de Dados, 8
Dígito binário, 1

Executável, Arquivo, 4
Expoente, 5

Fortran 90/95, 2
    Lahey, 2
    PGHPF, 2
FORTRAN, Significado, 4
Funções, 25
    Definida-Usuário, 25
FUNCTION, 25

Hexadecimal, 3

IMPLICIT NONE, 7
INTEGER, 5
Inteiros, 5
INTENT
    INOUT, 21
    IN, 21
    OUT, 21
Interfaces, 20
    Explícitas, 20
    Implícitas, 21

John Backus, 3

KIND, 9

LaTeX, 1
Linux/GNU, 4

Mantissa, 5
MegaFlops, 2
MODULE, 17, **19**
    Declaração, 17
    Procedimentos, 20
Módulos, 17

Objeto, Código, 4

Ponto Flutuante, 5
    Reais, 5
Precisão Dupla, 6
Precisão Simples, 5
PRIVATE, 22
Procedimentos Externos, 23
Processamento Paralelo, 2
    Vetorização, 2
Programa Principal, 7
PUBLIC, 22

Reais, 5
    Pontos Flutuantes , 5
    REAL, 5
RETURN, 24

SAVE, 19
SELECTED_INT_KIND, 11
SELECTED_REAL_KIND, 6, **11**
Subprograma, 23
SUBROUTINE, 23

Unicode, 2
USE, 19

Vetorização, 2
    Processamento Paralelo, 2