Lógica

de

Programação

Índice

Capítulo 1 – Conceitos Iniciais

Introdução

Muitos anos se passaram desde os primórdios da história da computação, mas apesar de já termos vivido vários paradigmas de programação, existe uma base de conhecimento que não mudou e não mudará nunca – a Lógica de Programação.

Faço uma associação direta da Lógica de Programação com o Raciocínio Matemático, onde o importante é a interpretação de um problema e a utilização correta de uma fórmula, e não a sintaxe pré-definida da mesma. O saber da Lógica está no "praticar".

Não existem "fórmulas" em Informática, o que existe é o aperfeiçoamento de nossa forma de pensar e raciocinar sobre um problema, podendo extrair do mesmo uma solução eficiente e eficaz, sob um determinado ângulo de visão. Assim, verificamos que é preciso aprender a pensar sobre os problemas, extraindo deles o máximo de informações.

A solução que criamos para um determinado problema necessita ser exteriorizada e expressa numa linguagem publicamente conhecida. Assim, utilizamos a **lógica de programação** para desenvolver nossas soluções e os **algoritmos** para apresentar essas soluções ao mundo.

Venho acompanhando nos últimos anos vários livros de Algoritmos e Estruturas de Dados. Todos ensinam como representamos estruturas de controle e atribuições, ou como declaramos variáveis, mas nenhum deles – que eu tenha lido até o momento –, orientou o aluno na forma de **pensar**. Precisamos mais do que "fórmulas", precisamos aprender a pensar.

Os princípios da programação estruturada surgida no final da década de 60 – introduzidos por Dijkstra – levaram a necessidade de se ter uma linguagem que implementasse essas idéias, já que as linguagens de época (FORTRAN, COBOL e BASIC) não permitiam aplicar claramente as técnicas ensinadas. Assim, o professor Niklaus Wirth e seus colegas da Universidade Técnica de Zurique (Suíça) desenvolveram, no início dos anos 70, a linguagem PASCAL – uma derivação da linguagem ALGOL 60, porém de implementação mais simples e com uma estrutura de dados mais poderosa. O nome Pascal foi uma homenagem a Blaise Pascal, famoso matemático, que criou a calculadora baseada em discos de madeira, que foi a predecessora da calculadora de mesa e serviu de inspiração para diversos computadores.

Assim, nossa apostila oferecerá, inicialmente, conceitos gerais sobre Programação. Posteriormente, vocês terão exercícios de Raciocínio Matemático que lhes exercitarão o poder de PENSAR! Em seguida, apresentaremos como desenvolver algoritmos de soluções para Sistemas. E por último, vamos conhecer a Linguagem Pascal, a fim de vermos nossos algoritmos funcionando – ao vivo e à cores !

Programação Estruturada

A Programação Estruturada pode ser entendida como uma forma de programar que visa facilitar a escrita, entendimento, validação e manutenção de programas. Para Dijkstra, "*a arte de programar consiste na arte de organizar e dominar a complexidade*".

A Programação Estruturada procura reduzir o nível de complexidade através de três níveis:

- desenvolvimento do programa em diferentes fases por refinamento sucessivo (desenvolvimento top-down);
- decomposição do programa total em *módulos funcionais*, organizados de preferência num *sistema hierárquico*;
- uso de um número limitado de estruturas básicas de fluxo de controle dentro de cada módulo.

Desenvolvimento Top-Down

Na Programação Estruturada, ao desenvolvermos um algoritmo, temos como objeto um produto final – o programa. Todavia, para termos esta transição, passamos por várias fases, no sentido "cima para baixo", onde cada fase é documentada e principalmente obtida por "refinamento" da fase anterior, até chegarmos a um nível de detalhamento que permita implementar o algoritmo diretamente na linguagem de programação.

Modularização

A solução final de um problema é obtida através de soluções de subproblemas, o que permite dividir o programa em módulos com subfunções claramente delimitadas, que podem, inclusive, ser implementados separadamente, por diversos programadores de uma equipe.

Estruturas de Controle

São representadas pela seqüência simples, o comando condicional e o comando repetitivo, e fornecem ao programador um aumento da legibilidade e compreensão de cada módulo de programa. Assim, temos como uma das principais normas da Programação Estruturada : não usar comandos de desvio (GOTO).

Confiabilidade

Medimos a confiabilidade de um sistema através de sua resposta ao uso constante, no tocante a:

- não apresentar erros; e,
- corresponder às especificações.

Atualmente, a sociedade está totalmente dependente dos sistemas de computação. Assim, aumenta exponencialmente a importância do nosso trabalho.

Nos fins dos anos 60, constatou-se que as sistemáticas usadas pelos programadores eram os grandes responsáveis pela baixa confiabilidade dos programas. Como solução destes problemas, surgiu a Programação Estruturada (PE).

Manutenibilidade

As revisões sofridas por um programa (releases) tanto para correção de erros quanto para mudanças de especificação, são consideradas como manutenção de software.

Os programas devem passar por testes exaustivos de confiabilidade antes de serem colocados em produção. Falhas nesta fase levam a altos níveis de manutenção, que conseqüentemente, levam a altos custos.

PseudoLinguagem - PORTUGOL

Sabemos que ao desenvolver programas necessitamos de nossa criatividade, para que tenhamos soluções eficazes e eficientes. Todavia, não podemos representar nossas soluções em algoritmos totalmente escritos em português. Em programação, todas as vezes que executarmos um algoritmo a partir de um estado inicial **x**, devemos sempre obter o mesmo estado final **y**. Desta forma, é fácil perceber que a linguagem natural, não formalizada, geraria ambigüidades.

Assim, temos o PORTUGOL, que é uma pseudolinguagem de programação (simbiose do Português com o ALGOL e PASCAL), que permite pensarmos no problema e não na máquina que vai executar o algoritmo. Além disso, não perdemos a flexibilidade e continuamos a ter a proximidade com a linguagem humana, facilitando, portanto, a interpretação.

Raciocínio Matemático

As crianças aprendem facilmente como adicionar e subtrair valores. Suas dificuldades começam no momento em que elas se deparam com problemas e necessitam identificar quais operações trarão soluções para os mesmos.

Vejamos alguns exercícios de Raciocínio Matemático, que ajudarão a "exercitar" nosso cérebro. No final desta apostila, vocês encontrarão o gabarito, mas não olhem antes de tentar resolvê-los. Todos serão corrigidos em sala de aula.

Exercícios

1) Há três suspeitos de um crime: o cozinheiro, a governanta e o mordomo. Sabe-se que o crime foi efetivamente cometido por um ou por mais de um deles, já que podem ter agido individualmente ou não. Sabe-se, ainda que:

   a. se o cozinheiro é inocente, então a governanta é culpada;
   b. ou o mordomo é culpado ou a governanta é culpada, mas não os dois;
   c. o mordomo não é inocente.

   Logo:
   a. a governanta e o mordomo são os culpados
   b. o cozinheiro e o mordomo são os culpados
   c. somente a governanta é culpada
   d. somente o cozinheiro é inocente
   e. somente o mordomo é culpado.

2) Qual o número que completa a seqüência: 1, 3, 6, 10, ...

   a. 13
   b. 15
   c. 12
   d. 11
   e. 18

3) Um frasco contém um casal de melgas. As melgas reproduzem-se e o seu número dobra todos os dias. Em 50 dias o frasco está cheio. Em que dia o frasco esteve meio cheio ?

   a. 25
   b. 24
   c. 26
   d. 49
   e. 2

4) Qual o número que completa a seqüência: 1, 1, 2, 3, 5, ...

   a. 5
   b. 6
   c. 7
   d. 8
   e. 9

5) Num concurso de saltos, Maria foi, simultaneamente, a 13ª melhor e 13ª pior. Quantas pessoas estavam em competição?

   a. 13
   b. 25
   c. 26
   d. 27
   e. 28

6) Bruno é mais alto que Joaquim. Renato é mais baixo que o Bruno. Então, Joaquim é o mais alto dos três.

   ( ) Verdadeiro
   ( ) Falso

7) O preço de um produto foi reduzido em 20% numa liquidação. Qual deverá ser a percentagem de aumento do preço do mesmo produto para que ele volte a ter o preço original ?

   a. 15%
   b. 20%
   c. 25%
   d. 30%
   e. 40%

Use a descrição abaixo para resolver os exercícios 8 e 9.

Chapeuzinho Vermelho ao entrar na floresta, perdeu a noção dos dias da semana. A Raposa e o Lobo Mau eram duas estranhas criaturas que freqüentavam a floresta. A Raposa mentia às segundas, terças e quartas-feiras, e falava a verdade nos outros dias da semana. O Lobo Mau mentia às quintas, sextas e sábados, mas falava a verdade nos outros dias da semana.

Um dia Chapeuzinho Vermelho encontrou a Raposa e o Lobo Mau descansando à sombra de uma árvore. Eles disseram:

Raposa: "Ontem foi um dos meus dias de mentir"

Lobo Mau: "Ontem foi um dos meus dias de mentir"

8) A partir dessas afirmações, Chapeuzinho Vermelho descobriu qual era o dia da semana. Qual era?

9) Em qual dia da semana é possível a Raposa fazer as seguintes afirmações?

   Eu menti ontem.

   Eu mentirei amanhã.

10) José quer ir ao cinema assistir ao filme "Fogo Contra Fogo", mas não tem certeza se o mesmo está sendo exibido. Seus amigos, Maria, Luis e Julio têm opiniões discordantes sobre se o filme está ou não em cartaz. Se Maria estiver certa, então Julio está enganado. Se Julio estiver enganado, então Luís está enganado. Se Luis estiver enganado, então o filme não está sendo exibido. Ora, ou o filme "Fogo conta Fogo" está sendo exibido, ou José não irá ao cinema. Verificou-se que Maria está certa. Logo,

    a. O filme "Fogo contra Fogo" está sendo exibido
    b. Luis e Julio não estão enganados
    c. Julio está enganado, mas Luis não.
    d. Luis está enganado, mas Julio não.
    e. José não irá ao cinema.

Capítulo 2 - Algoritmos

Segundo Wirth, "programas são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados".

De forma bem simples, um algoritmo pode ser definido como "um conjunto de passos lógicos, bem definidos, que descreve a solução de um problema".

Ao pensarmos na solução de um problema, encontramos ações imperativas que são expressas por **comandos**. Os algoritmos não são aplicados apenas ao mundo da Informática; pelo contrário, usamos – até sem perceber – algoritmos em todos os momentos de nossa vida. Uma receita de cozinha é claramente um algoritmo.

Veja um exemplo:

Faça um algoritmo para "Ir de casa para o trabalho de ônibus"

Solução 1

Algoritmo Trajeto_Casa_Trabalho_V1
início
Andar até o ponto de ônibus
Aguardar o ônibus
Ao avistar o ônibus correto, fazer sinal
Entrar no ônibus pela porta traseira
Pagar passagem
Escolher um assento e sentar
Quando chegar próximo do local a saltar, dar o sinal para descida
No ponto, descer do ônibus, pela porta dianteira
Andar até o trabalho
fim

Veja que resolvemos esse algoritmo em 9 passos, todavia se pedirmos que **n** pessoas resolva o mesmo problema, provavelmente, teremos **n** respostas diferentes. Isto ocorre pois, normalmente, abstraímos um problema, de ângulos diferentes, com maior ou menor riqueza de detalhes.

Por outro lado, devemos perceber que o algoritmo descrito revela uma situação perfeita, sem condicionais, sem exceções. Assim como na nossa rotina é improvável termos situações perfeitas, essas exceções também ocorrem nos programas de computador.

Vamos refazer este algoritmo de forma a introduzir algumas condições.

Solução 2

Algoritmo Trajeto_Casa_Trabalho_V2
início
1. Andar até o ponto de ônibus
2. Aguardar o ônibus
3. Quando avistar o ônibus correto, fazer sinal
se o ônibus não parar, então
Em pensamento, xingar o motorista
Reclamar para si que vai chegar atrasado
se estiver muito atrasado então
Pegar uma Van
senão
Voltar para o Passo 2
fim-se
senão
se Pessoa >= 65 anos então
Entrar pela porta dianteira
senão
Entrar pela porta traseira
Pagar passagem
se houver troco então
Aguardar troco
fim-se
fim-se
se houver lugar disponível então
Sentar
senão
Escolher o melhor lugar em pé e ali permanecer
fim-se

Quando chegar próximo do local a saltar, dar o sinal para descida
No ponto, descer do ônibus, pela porta dianteira
Andar até o trabalho
fim-se
fim

Com certeza, a brincadeira que fiz da condição "Se o ônibus não parar" deve ter levado vocês a pensarem em inúmeras novas condições, como por exemplo: qual seria a sua reação, se num dia de chuva, o ônibus passasse por sobre uma poça e lhe sujasse toda a roupa?

Veja quão complexo pode se tornar um "simples" algoritmo. Devemos lembrar que detalhes são essenciais na confecção de um algoritmo, todavia, eles devem estar de acordo com o **contexto**. Além disso, é importante que venhamos a relatar apenas os detalhes relevantes.

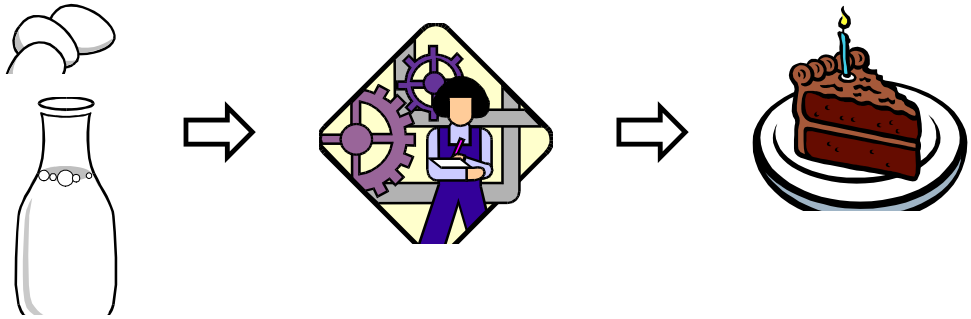
0

Por exemplo, a solução 2 está apropriada para ensinarmos uma pessoa que não está acostumada a andar de ônibus. Todavia, este algoritmo causaria problemas se estivéssemos programando um robô. Considerando esta situação, deveríamos ser mais precisos no passo "Quando chegar próximo do local a saltar, dar o sinal de descida". Nesse caso, deveríamos dizer

"A x metros do local a saltar, dar o sinal de descida" ou "Na altura x da Rua y ...".

Assim, lembrem-se de usar o BOM SENSO!

Podemos pensar também num algoritmo como um "mecanismo" de transformação de entradas em saídas. Assim, um algoritmo ao ser "executado", receberá algumas entradas, que serão processadas e nos devolverá algumas saídas.



Fluxo de Controle em Algoritmos

Um algoritmo é um texto estático, onde temos vários passos que são lidos e interpretados de cima para baixo. Para que venhamos a obter o(s) resultado(s) deste algoritmo, necessitamos "executá-lo", o que resulta em um processo dinâmico.

No fluxo de controle identificamos em cada passo da execução qual é o próximo comando a ser executado.

A compreensão da lógica de programação de um algoritmo está diretamente ligada a compreensão de seu fluxo de controle. A partir de uma compreensão correta, podemos traçar as diversas execuções possíveis de um algoritmo. Se testarmos todas essas possibilidades, e obtivermos resultados corretos, podemos ter certeza de estar entregando um produto final confiável.

Os iniciantes no mundo da programação encontram alguma dificuldade em diminuir a distância conceitual que separa a representação estática de um algoritmo do(s) processo(s) dinâmico(s) de sua execução. É importante frisar que quando nos propomos a entender um algoritmo, lidamos fisicamente com um texto, mas mentalmente temos processos.

Criando Algoritmos

Toda linguagem é composta de sintaxe e semântica, onde a sintaxe corresponde à forma e a semântica corresponde ao conteúdo.

Vocês devem aprender a sintaxe dos comandos, mas a principal preocupação deve ser de "como usar esses comandos".

Regras para criação de bons algoritmos

Use comentários com freqüência. Isto torna o algoritmo mais legível e facilita o entendimento da lógica empregada. Seus algoritmos deverão ser lidos e entendidos por outras pessoas (e por você mesmo) de tal forma que possam ser corrigidos e receber manutenção.

Obs: Não se esqueça de atualizar os comentários, em caso de manutenção. Pior do que um programa sem comentários, é um programa com comentários errados.

Use comentários, também, no cabeçalho do algoritmo, incluindo, principalmente:

- descrição do que faz o algoritmo
- autor
- data de criação

Escolha nomes de variáveis significativos, todavia evite nomes muito longos.

Ex: Prefira SalBruto ou SalarioBruto ao invés de SB ou VAR1

Prefira TotAlunosAprovDireta ao invés de TotalAlunosAprovacaoDireta

Destaque as palavras-chave das estruturas de controle e comandos com sublinhado.

```
Ex: se media >= 7 então
        ...
    senão
        ...
    fim-se
```

Utilize espaços e linhas em branco para melhorar a legibilidade.

Coloque apenas um comando por linha. Vários comandos em uma linha causa ilegibilidade e dificulta a depuração.

Utilize parênteses para aumentar a legibilidade e prevenir-se de erros.

Use identação nos comandos de acordo com o nível que estejam, ou seja, alinhe comandos de mesmo nível e desloque comandos de nível inferior.

Ex.:

```
início
|       comando 1;
|       se condicao1 então
|       |       comando2;
|       |       comando3;
|       senão
|       |       comando4;
|       |       comando5;
|       fim-se
|       comando6;
fim
```

Método para desenvolvimento de algoritmos

Faça uma leitura de todo o problema até o final, a fim de formar a primeira impressão. A seguir, releia o problema e faça anotações sobre os pontos principais.

Verifique se o problema foi bem entendido. Questione, se preciso, ao autor da especificação sobre suas dúvidas. Releia o problema quantas vezes for preciso para tentar entendê-lo.

Extraia do problema todas as suas saídas.

Extraia do problema todas as suas entradas.

Identifique qual é o processamento principal.

Verifique se será necessário algum valor intermediário que auxilie a transformação das entradas em saídas. Esta etapa pode parecer obscura no início, mas com certeza no desenrolar do algoritmo, estes valores aparecerão naturalmente.

Teste cada passo do algoritmo, com todos os seus caminhos para verificar se o processamento está gerando os resultados esperados.

Crie valores de teste para submeter ao algoritmo.

Reveja o algoritmo, checando as boas normas de criação.

Conselho: Só tente conseguir o ótimo, depois de realizar o bom.

Identificadores

As variáveis, funções e procedimentos que usamos em nossos algoritmos precisam receber um nome (rótulo). Estes nomes são chamados de Identificadores e possuem algumas regras de formação:

O primeiro caractere deve ser, obrigatoriamente, uma letra.

Do segundo caractere em diante são permitidos números e letras. O símbolo de underscore (_) pode ser usado para separar nomes compostos. Portanto, não são permitidos espaços, caracteres acentuados e símbolos especiais na composição do nome de um identificador;

Palavras reservadas (em inglês ou português) não podem ser usadas com identificadores. (Exemplo: begin, end, for, var, inicio, fim, para, etc...)

Não há distinção entre maiúsculo e minúsculo, na forma como os identificadores são escritos;

Exemplos de nomes de identificadores:

SalarioBruto

Preco_Unitario

BuscaValor

NOTA1

Nota1

Variáveis

Vocês sabem que os computadores possuem CPU e memória, certo ? Sim. A CPU (Unidade Central de Processamento) é responsável pelo controle e processamento dos cálculos matemáticos e das resoluções de expressões lógicas. Todavia, os dados que são usados num processamento precisam ser armazenados em algum lugar – este lugar é a memória principal. Ela funciona como um "armário" que guarda nossos pertences. Todavia, como um armário, não podemos simplesmente ir guardando nossos pertences sem nenhuma arrumação. Para isso, existem "caixas" na memória (posições de memória), que nos permitem organizar essas informações. Essas caixas, conceitualmente recebem nomes e são conhecidas como **variáveis**.

Assim, a variável é o local da memória onde guardamos os dados e o nome da variável é um identificador conforme definição anterior.

Exemplos de variáveis:

SalarioBruto

NomeFuncionario

Toda variável necessita ser declarada, ou seja, reserva-se um local da memória informando que tipo de dados residirão ali. Assim, a sintaxe de declaração de uma variável é:

```
variável  :  tipo de dados ;
```

ou

```
variável1, variável2, ..., variáveln  :  tipo de dados ;
```

Exemplo:

Se declararmos as variáveis A, B e C da seguinte forma:

declare

A: inteiro;

B : caracter;

C : lógico;

Estamos criando áreas na memória identificadas por A, B e C, que só poderão receber, respectivamente, valores inteiros, alfanuméricos e lógicos (Verdadeiro ou Falso).

Tipos Básicos de Dados

Ao armazenarmos variáveis na memória do computador, precisamos dizer que tipo elas são, para que seja reservado o espaço adequado, além de ser dado o trabalho correto a elas. Além dos tipos básicos de dados citados abaixo, podemos criar nossos próprios tipos.

INTEIRO: qualquer número inteiro, negativo, nulo ou positivo
Ex.: -15, 0, 101

REAL: qualquer número real, negativo, nulo ou positivo
Ex.: -1, -0.5, 0, 5, 9.5

CARACTER ou STRING: qualquer conjunto de caracteres alfanuméricos
Ex.: “AB”, “ 123”, “ A123” , “CASA”

LÓGICO ou BOOLEANO: conjunto de valores (FALSO ou VERDADEIRO)

Comentários

Comentários devem ser inseridos no algoritmo a fim de esclarecer o desenvolvimento do mesmo. Os comentários são inseridos entre { e }.

{ Texto de comentário delimitado por chaves }

Comando de Atribuição

Ao criarmos uma variável, partimos do princípio que em algum momento ou vários momentos dentro do nosso algoritmo, ela receberá valores, ou seja, armazenaremos dados na memória através de nossas variáveis.

Para atribuirmos um valor ou uma expressão a uma variável, utilizamos o comando de atribuição ←.

Assim, a sintaxe do comando é:

```
identificador ← expressão ;
```

Exemplo:

Salario ← 1000

Nome ← ‘Ana’

Operadores Aritméticos

É comum necessitarmos realizar cálculos matemáticos com as informações que estamos manipulando. Para isso, é necessário sabermos qual a representação dos símbolos de operações matemáticas. Vejamos:

Operador	Operação	Exemplo
+	Adição	10 + 15
-	Subtração	20 – 10
*	Multiplicação	3 * 5
/	Divisão (onde o resultado será um número real)	5 / 2 = 2,5
DIV	Divisão (onde o resultado será um número inteiro)	10 div 2 = 5 7 div 2 = 3
MOD	Resto de uma divisão	7 mod 2 = 1
** **ou exp(a, b)**	Exponenciação	5 ** 2 ou exp(5, 2)

Operadores Relacionais

Além de operações matemáticas, é freqüente nossa necessidade em comparar informações. Por exemplo: Se média for maior ou igual a 7. Para isso, utilizamos operadores relacionais.

Operador	Relação
=	Igualdade
≠ ou <>	Diferente
>	Maior que
≥ ou >=	Maior ou igual que
<	Menor que
≤ ou <=	Menor ou igual que

Operadores Lógicos

É freqüente precisarmos analisar expressões lógicas, aquelas que só possuem dois valores possíveis: Verdadeiro ou Falso. Os operadores usados em expressões lógicas são os Operadores Lógicos. Veja:

Operador	Relação
E (And)	E lógico
Ou (Or)	Ou lógico
Não (Not)	Negação lógica
Ou-X (Xor)	Ou 'Exclusivo'

Veja a seguinte tabela para entender melhor os operadores lógicos.

P	Q	P e Q	P ou Q	P ou-X Q	não P
F	**F**	F	F	F	V
F	**V**	F	V	V	--
V	**F**	F	V	V	F
V	**V**	V	V	F	--

Prioridade na Avaliação de Expressões

1º Parênteses e funções (resolvidos da esquerda para a direita)

2º Multiplicação (*), Divisão (/ e div) e Resto (Mod)
(resolvidos da esquerda para a direita)

3º soma e subtração

4º Operadores relacionais: >, <, **≥, ≤, =, ≠**

5º Operador Lógico Não

6º Operador Lógico E

7º Operador Lógico Ou

Comandos de Entrada e Saída

Para atingirmos os objetivos de um algoritmo, necessitamos receber dados do mundo externo e precisamos exteriorizar as informações produzidas. No momento do desenvolvimento de um algoritmo, não nos interessa saber se os dados virão via teclado, ou pela leitura de um arquivo de dados ou por qualquer outro meio.

Para obtermos e exteriorizarmos esses dados, utilizamos os seguintes comandos de entrada e saída, LER, ESCREVER e IMPRIMIR. O comando ler espera receber um determinado dado (sem importar a origem). O comando escrever mostra a informação produzida no vídeo. O comando imprimir faz a impressão em papel da informação produzida.

Veja a sintaxe dos comandos:

```
ler (variável1, variável2, ... , variável n);
escrever (lista de constantes, variáveis e/ou expressões );
imprimir (lista de constantes, variáveis e/ou expressões );
```

Exemplos:

```
ler(numero1);
numero2 ← numero1 * 2;
escrever('O dobro do número é ', numero2);
imprimir('O triplo do número é ', numero1 * 3);
```

Funções

Em toda linguagem vocês encontram funções primitivas que realizam operação básicas com os tipos de dados.

Nome da Função	Descrição da Função	Exemplo
raiz (x)	Retorna a raiz quadrada de x	A = raiz(25) ⇨ A = 5
sqr (x)	Retorna x elevado ao quadrado	A = sqr(4) ⇨ A = 16
abs (x)	Retorna o valor absoluto de x	A = abs(-15) ⇨ A = 15
int (x)	Retorna a parte inteira de x	A = int(4,5) ⇨ A = 4

Operações com Strings

Nem só de números vive um programa, portanto, precisamos também poder manipular dados do tipo string. Vejamos:

Nome da Função/Operador	Descrição da Função	Exemplo
+	concatenação (união) de strings	A : string A = 'cris' + 'tina' ⇨ A = 'cristina'
Len	retorna o tamanho de uma string	A : inteiro A = tamanho('ana') ⇨ A = 3
Ord	retorna o código ASCII[1] de um caractere	A : inteiro A = ord('A') ⇨ A = 65
Chr	retorna o caractere correspondente ao código ASCII recebido por parâmetro	A : string A = chr(66) ⇨ A = 'B'
ucase	converte toda uma string para maiúsculo	A : string A = ucase('ana') ⇨ A = 'ANA'
lcase	converte toda uma string para minúsculo	A : string A = lcase('ANA') ⇨ A = 'ana'
pos	retorna a posição de uma substring dentro de uma string	A : inteiro A = pos('asa', 'casa') ⇨ A = 2
substring	retorna parte de uma string, a partir de uma determinada posição	A : string A = substring('casa', 2, 3) A = 'asa'

Estrutura de um Algoritmo

```
declare
        < declaração de variáveis, constantes e tipos >
início
        < comandos >
fim
```

Estruturas de Controle

Blocos

Delimitam um conjunto de comandos com uma função bem definida.

[1] ASCII - American Standard Code for Information Interchange

```
início
        <comandos>
fim
```

Seqüências Simples

Conjunto de comandos que serão executados numa seqüência linear de cima para baixo. Estes comandos podem aparecer em qualquer estrutura de controle, agrupados ou não por blocos.

Ao final de cada comando é obrigatório a colocação de um ponto-e-vírgula (;). Mais de um comando pode ser colocado por linha, mas isso não é aconselhável. Um comando pode ocupar mais de uma linha. Nesse caso, o ponto-e-vírgula só vai aparecer no final da última linha.

Veja a sintaxe:

```
comando 1;
comando 2;
início comando 3 ...
        meio comando 3 ...
        fim comando 3 ;
...
comando n;
```

Exemplo 1

Faça um algoritmo que leia dois números inteiros e mostre a soma deles.

```
Algoritmo SomaNumerosInteiros
declare
        num1, num2, soma : inteiro;
inicio
        ler (num1, num2);
        soma ← num1 + num2;
        escrever('A soma dos números é : ', soma);
fim
```

Exemplo 2

Faça um algoritmo que leia 3 nomes e mostre-os na ordem inversa de leitura

Obs: Veja como este algoritmo já apresenta mais detalhes.

```
Algoritmo LeituraNomes
declare
        nome1, nome2, nome3 : string;
inicio
        escrever (‘Entre com primeiro nome : ‘);
        ler (nome1);
        escrever (‘Entre com segundo nome : ‘);
        ler (nome2);
        escrever (‘Entre com terceiro nome : ‘);
        ler (nome3);
        escrever (‘A ordem inversa dos nomes é ‘);
        escrever (nome3);
        escrever (nome2);
        escrever (nome1);
fim
```

Exemplo 3

Fazer um algoritmo que leia uma palavra e mostre a primeira letra dela.

```
Algoritmo PrimeiraLetra
declare
        palavra, letra1 : caractere;
inicio
        escrever(‘Digite palavra : ‘ );
        ler(palavra);
        letra1 ← Substring(palavra, 1, 1);
        escrever(‘A primeira letra da palavra é : ‘ , letra1);
fim
```

Estruturas Condicionais

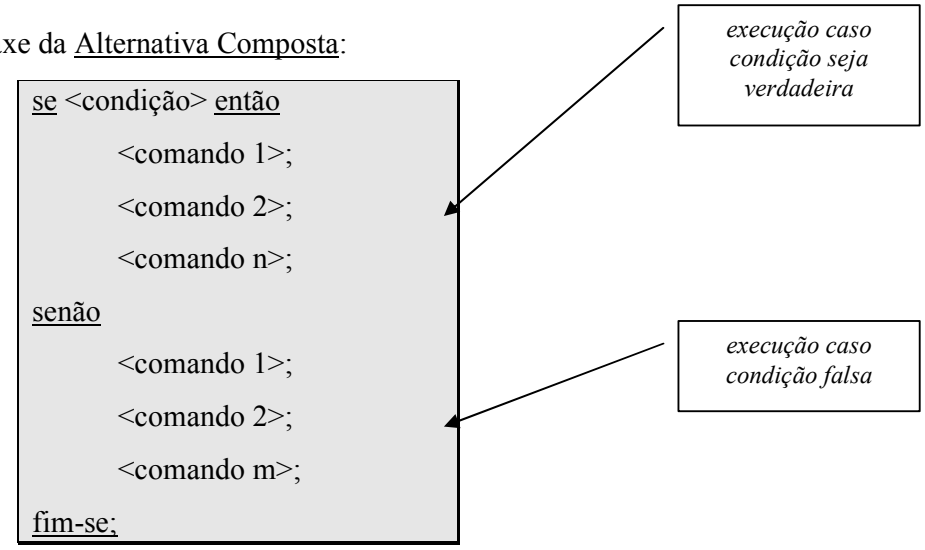
Quando uma ação para ser executada depender de uma inspeção ou teste, teremos uma **alternativa simples** ou **composta**.

Sintaxe da Alternativa Simples:

```
se <condição> então
        <comando 1>;
        <comando 2>;
        <comando n>;
fim-se;
```

execução caso condição seja verdadeira

Sintaxe da Alternativa Composta:



onde: <condição> é qualquer expressão cujo resultado seja Falso ou Verdadeiro.

Exemplo:

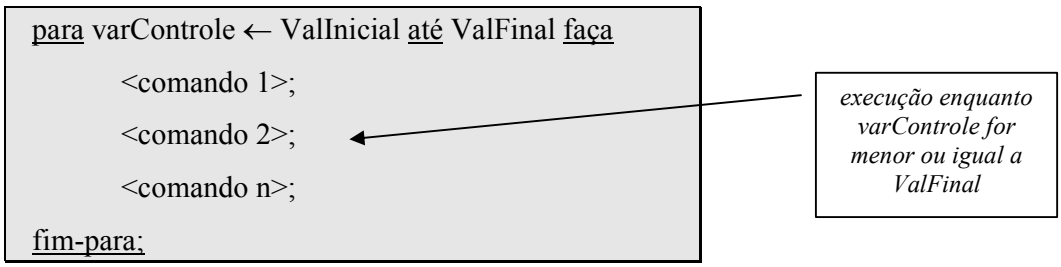
```
se media >= 7 então
        situacao ← 'Aprovado';
senão
        situacao ← 'Reprovado';
fim-se
```

Estruturas de Repetição

Quando um conjunto de ações é executado repetidamente enquanto uma determinada condição permanece válida.

Comando Para

Usamos a estrutura Para, quando precisamos repetir um conjunto de comandos um número pré-definido de vezes. Utiliza uma variável de controle, que é incrementada em 1 unidade de um valor inicial até um valor final.



Quando o programa encontra a instrução fim-para, incrementa a variável varControle em 1 unidade. Cada vez que o programa passa pela linha de instrução para ..., ele testa se varControle é menor ou igual a ValFinal. Se não for, o comando é abandonado.

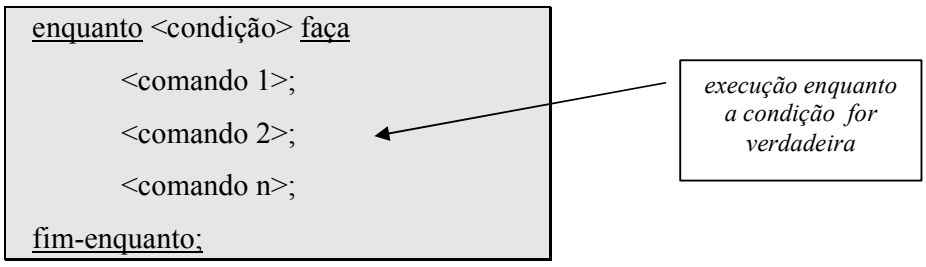
Obs: O valor da variável varControle não pode ser alterado no interior da estrutura para...fim-para.

Exemplo:

```
para aux ← 1 até 10 faça
        resultado ← 5 * aux;
fim-para
```

Enquanto

Utilizada quando não sabemos o número de repetições e quando possuímos uma expressão que deve ser avaliada para que os comandos da estrutura sejam executados. Assim, enquanto o valor da <condição> for verdadeiro, as ações dos comandos são executadas. Quando for falso, a estrutura é abandonada, passando a execução para a próxima linha após o comando. Se já da primeira vez o resultado for falso, os comandos não são executados nenhuma vez.

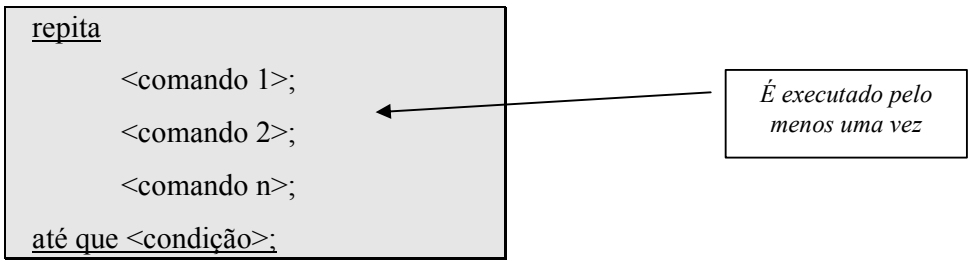


Exemplo:

```
        aux ← 1;
       enquanto (aux <= 10) faça
                sultado ← 5 * aux;
                x ← aux + 1;
        fim-para
```

Repita ... Até que

Utilizada quando não sabemos o número de repetições e quando os comandos devem ser executados pelo menos uma vez, antes da expressão ser avaliada. Assim, o programa entra na estrutura Repita...Até que e executa seus comandos pelo menos uma vez. Ao chegar no fim da estrutura, a expressão será avaliada. Se o resultado da expressão for verdadeiro, então o comando é abandonado.



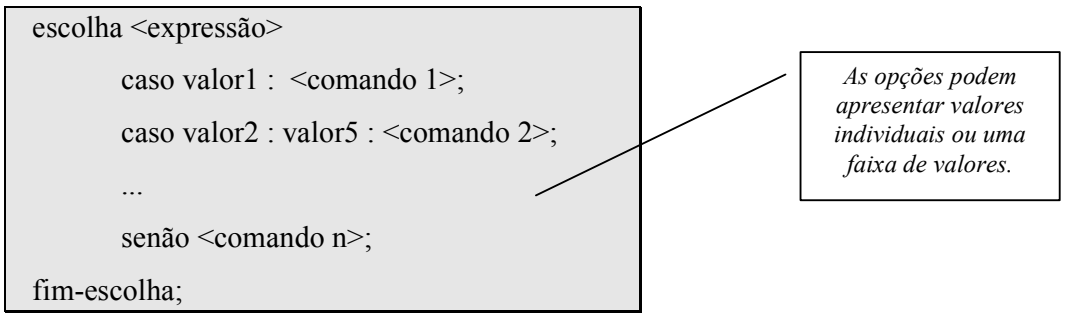
Exemplo:

```
aux ← 1;
repita
        resultado ← 5 * aux;
        escrever resultado;
        aux ← aux + 1;
até que (aux > 10);
```

Resultado do algoritmo: 5 10 15 20 25 30 35 40 45 50

Estrutura de Múltipla Escolha

Utilizada quando temos muitas possibilidades para uma determinada situação, onde a aplicação da estrutura se...então...senão...fim-se, tornaria o algoritmo muito complexo.



Exemplo:

```
ler(Numero);
escolha Numero
        caso 1: Extenso ← 'Um';
        caso 2: Extenso ← 'Dois';
        caso 3: Extenso ← 'Três';
        caso 4: Extenso ← 'Quatro';
        caso 5: Extenso ← 'Cinco';
        caso 6: Extenso ← 'Seis';
        caso 7: Extenso ← 'Sete';
        caso 8: Extenso ← 'Oito';
        caso 9: Extenso ← 'Nove';
        senão:  Extenso ← 'Erro';
 fim-escolha;
```

Capítulo 3 – Variáveis Multidimensionais

Vetores

Os vetores são estruturas de dados que permitem o armazenamento de um conjunto de dados de mesmo tipo. Por este motivo, são chamadas de estruturas homogêneas. Os vetores são unidimensionais, pois cada elemento do vetor é identificado por um índice.

Similarmente, podemos definir vetores como posições de memória, identificadas por um mesmo nome, individualizadas por índices e cujo conteúdo é de mesmo tipo.

Para acessarmos um elemento de um vetor, referimo-nos ao nome do vetor acompanhado pelo seu índice que virá entre colchetes (**[** e **]**). Pense num prédio com 120 apartamentos. Para enviar uma correspondência a um determinado apartamento, devemos colocar no endereço de destinatário, o número do prédio mais o número do apartamento. O vetor funciona de forma similar.

Veja a sintaxe da declaração de um vetor:

```
Nome do vetor : vetor [ nº de elementos ] de <tipo básico do vetor >
```

Para fazermos referência a um elemento do vetor, colocamos:

```
Nome do vetor [ elemento ]
```

Cada elemento de um vetor é tratado como se fosse uma variável simples.

Exemplo:

Supondo que pedíssemos para criar um algoritmo para ler o nome de 4 pessoas, e mostrasse esses nomes na ordem inversa de leitura. A princípio, vocês pensariam em cinco variáveis: nome1, nome2, nome3 e nome4.

Veja como ficaria a solução, nesse caso:

```
declare
        nome1, nome2, nome3, nome4 : caracter;
início
        escrever('Informe o nome de 4 pessoas: ');
        ler(nome1);
        ler(nome2);
        ler(nome3);
        ler(nome4);
        escrever('Ordem Inversa de Leitura ');
        escrever(nome4);
        escrever(nome3);
        escrever(nome2);
        escrever(nome1);
fim
```

Assim, na memória teríamos ...

Nome1	Nome2	Nome3	Nome4
ANA	PAULA	CRISTINA	GUSTAVO

Todavia, se alterássemos esse algoritmo para ler o nome de 100 pessoas, a solução anterior se tornaria inviável. Para casos como este, podemos fazer uso de vetores. Se tivéssemos criado 100 variáveis, teríamos que declarar e usar: nome1, nome2, nome3, ... , nome99, nome100. Com o vetor passamos a ter: nome[1], nome[2], nome[3], nome[99], nome[100], onde a declaração do vetor se limita à linha: nome : vetor[1..100] de caracter.

Veja que para todos os elementos nos referimos ao mesmo nome de vetor.

Assim, veja a solução do algoritmo anterior com o uso de vetores:

```
declare
        nome : vetor[4] de caracter;
        aux : inteiro;
início
        para aux ← 1 até 4 faça
                escrever (‘Informe o Nome ‘, aux);
                ler (nome[aux]);
        fim-para;
        escrever(‘Ordem Inversa de Leitura ‘);
        para aux ← 4 até 1 faça
                escrever (nome[aux]);
        fim-para
fim
```

Veja a representação da memória:

Nome[1]	Nome[2]	Nome[3]	Nome[4]
ANA	PAULA	CRISTINA	GUSTAVO

Matrizes

As matrizes são estruturas de dados que permitem o armazenamento de um conjunto de dados de mesmo tipo, mas em dimensões diferentes. Os vetores são unidimensionais, enquanto as matrizes podem ser bidimensionais (duas dimensões) ou multidimensionais.

Similarmente podemos conceituar matrizes como um conjunto de dados referenciado por um mesmo nome e que necessitam de mais de um índice para ter seus elementos individualizados.

Para fazer referência a um elemento da matriz serão necessários tantos índices quantas forem as dimensões da matriz.

Veja a sintaxe da declaração de uma matriz:

Nome da matriz : matriz [li_1: ls_1, li_2:ls_2, ... , li_n:ls_n] de <tipo básico da matriz >

onde:

- **li** – limite inferior
- **ls** – limite superior
- **li_1: ls_1, li_2:ls_2, ... , li_n:ls_n** – são os limites dos intervalos de variação dos índices da matriz, onde cada par de limites está associado a um índice.
- **tipo** – tipo a que pertencem todos os campos do conjunto.

Para fazermos referência a um elemento da matriz, colocamos:

Nome da matriz [linha, coluna]

O número de dimensões de uma matriz pode ser obtido pelo número de vírgulas (,) da declaração mais 1. O número de elementos pode ser obtido através do produto do número de elementos de cada dimensão.

Obs: Quando você desejar percorrer uma matriz, linha por linha, crie uma estrutura de repetição, fixando a linha e variando a coluna. Para percorrer uma matriz, coluna por coluna, fixe a coluna e varie a linha.

Vamos pensar numa estrutura onde as colunas representem os cinco dias úteis da semana, e as linhas representem as três vendedoras de uma loja. Na interseção de cada linha x coluna, colocaremos o faturamento diário de cada vendedora.

	(Segunda) COLUNA 1	(Terça) COLUNA 2	(Quarta) COLUNA 3	(Quinta) COLUNA 4	(Sexta) COLUNA 5
(SANDRA) LINHA 1	1050,00	950,00	1241,00	2145,00	1256,00
(VERA) LINHA 2	785,00	1540,00	1400,00	546,00	0,00
(MARIA) LINHA 3	1658,00	1245,00	1410,00	245,00	1546,00

A representação desta tabela em forma de matriz, seria:

VendasDiarias : matriz [3, 5] de real;

Indicando a declaração de uma matriz com 3 linhas e 5 colunas, cujos valores serão do tipo real.

Veja como ficaria o algoritmo para ler esses valores:

```
Algoritmo LeVendasDiarias;
declare
        VendasDiarias : matriz[3,5] de real;
        indLinha, indColuna : inteiro;
início
        { Variando o número de linhas - Vendedoras}
        para indLinha ← 1 até 3 faça
                escrever ('Vendedora :', indLinha);
                { Variando o número de colunas – Dias da Semana}
                para indColuna ← 1 até 5 faça
                        escrever ('Faturamento do Dia : ', indColuna);
                        ler (VendasDiarias[indLinha, indColuna]);
                fim-para;
        fim-para;
fim
```

Poderíamos melhorar o algoritmo acima, trabalhando com um vetor que contivesse os nomes dos dias da semana e das vendedoras. Assim, a comunicação do programa com o usuário ficaria mais clara. Veja:

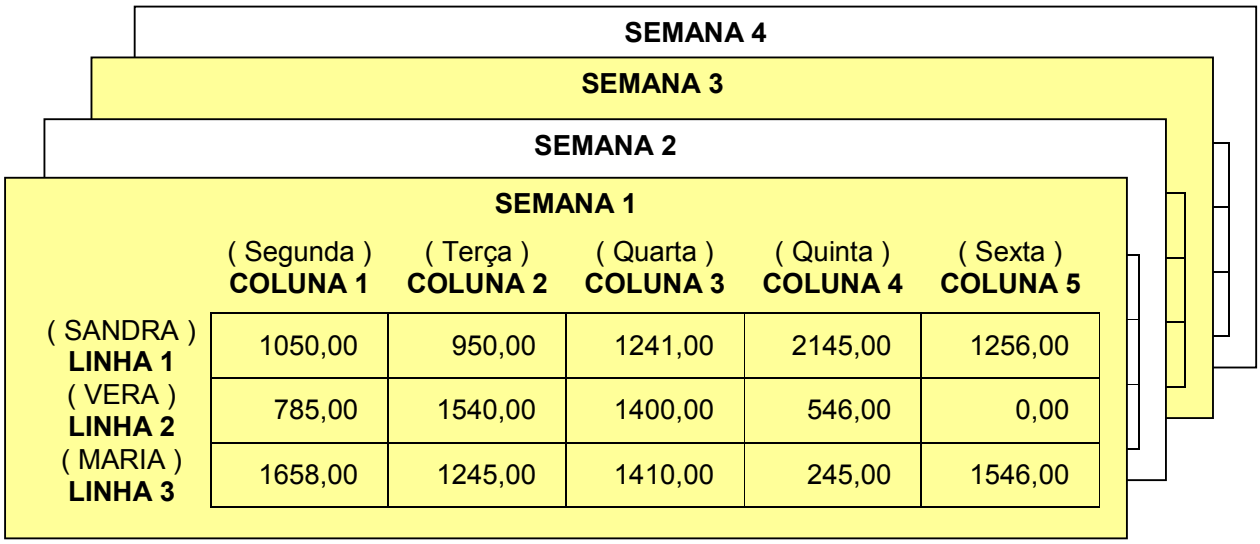
```
Algoritmo LeVendasDiariasVersao2;
declare
        VendasDiarias : matriz[3,5] de real;
        Vendedoras : vetor[3] de caracter;
        DiasSemana : vetor[5] de caracter;
        indLinha, indColuna : inteiro;
início
        Vendedoras[1] ← 'Sandra';
        Vendedoras[2] ← 'Vera';
        Vendedoras[3] ← 'Maria';
        DiasSemana[1] ← 'Segunda';
        DiasSemana[2] ← 'Terça';
        DiasSemana[3] ← 'Quarta';
        DiasSemana[4] ← 'Quinta';
        DiasSemana[5] ← 'Sexta';
        { Variando o número de linhas - Vendedoras}
        para indLinha ← 1 até 3 faça
                escrever ('Vendedora : ', Vendedoras[indLinha]);
                { Variando o número de colunas – Dias da Semana}
                para indColuna ← 1 até 5 faça
                        escrever ('Faturamento do Dia : ', DiasSemana[indColuna]);
                        ler (VendasDiarias[indLinha, indColuna]);
                fim-para;
        fim-para;
fim
```

Um algoritmo que apenas lê e nada faz com esses resultados, não serve para grande coisa, certo ?! Por isso, vamos melhorar esse algoritmo e apresentar como resultado o faturamento diário de todas as vendedoras.

```
Algoritmo LeVendasDiariasVersao3;
declare
        VendasDiarias : matriz[3,5] de real;
        Vendedoras : vetor[3] de caracter;
        DiasSemana : vetor[5] de caracter;
        indLinha, indColuna : inteiro;
        FaturaDia : real;
início
        Vendedoras[1] ← 'Sandra';
        Vendedoras[2] ← 'Vera';
        Vendedoras[3] ← 'Maria';
        DiasSemana[1] ← 'Segunda';
        DiasSemana[2] ← 'Terça';
        DiasSemana[3] ← 'Quarta';
        DiasSemana[4] ← 'Quinta';
        DiasSemana[5] ← 'Sexta';
        { Variando o número de linhas – Vendedoras }
        para indLinha ← 1 até 3 faça
                escrever ('Vendedora : ', Vendedoras[indLinha]);
                { Variando o número de colunas – Dias da Semana}
                para indColuna ← 1 até 5 faça
                        escrever ('Faturamento do Dia : ', DiasSemana[indColuna]);
                        ler (VendasDiarias[indLinha, indColuna]);
                fim-para;
        fim-para;
        { Vamos começar variando a coluna, para poder obter o faturamento
        de cada dia da semana }
        para indColuna ← 1 até 5 faça
                { A cada novo dia, a variável que recebe o faturamento é
                zerada }
                FaturaDia ← 0;
                { Vamos variar a linha, para obter os valores faturados de
                cada vendedora }
                para indLinha ← 1 até 3 faça
                        FaturaDia ← FaturaDia + VendasDiarias[indLinha, indColuna];
                fim-para
                escrever(Faturamento de : ', DiasSemana[indColuna]);
                escrever(FaturaDia);
        fim-para;
fim
```

Até agora, está fácil, certo ?! Então vamos complicar um pouquinho. Na matriz anterior, estamos controlando o faturamento de apenas uma semana. Todavia, as vendedoras trabalham o mês todo. E o correto seria termos uma planilha para cada semana do mês, que vamos considerar que sejam quatro semanas.

Exemplo:



A representação deste conjunto de planilhas em forma de matriz, seria:

VendasDiarias : matriz [3, 5, 4] de real;

Indicando a declaração de uma matriz com 3 linhas e 5 colunas, repetidas em 4 faces, cujos valores serão do tipo real.

Assim, vejamos como fica o algoritmo para lermos os faturamentos diários por semana.

```
Algoritmo LeVendasDiariasVersao4;
declare
        VendasDiarias : matriz[3,5,4] de real;
        Vendedoras : vetor[3] de caracter;
        DiasSemana : vetor[5] de caracter;
        indLinha, indColuna, indFace : inteiro;
início
        Vendedoras[1] ← 'Sandra';
        Vendedoras[2] ← 'Vera';
        Vendedoras[3] ← 'Maria';
        DiasSemana[1] ← 'Segunda';
        DiasSemana[2] ← 'Terça';
        DiasSemana[3] ← 'Quarta';
        DiasSemana[4] ← 'Quinta';
        DiasSemana[5] ← 'Sexta';



        { Variando o número de faces – Semanas do mês }
        para indFaces ← 1 até 4 faça
                escrever ('Semana do Mês : ',indFaces);
                { Variando o número de linhas – Vendedoras }
```

```
                para indLinha ← 1 até 3 faça
                        escrever (‘Vendedora : ‘, Vendedoras[indLinha]);
                        { Variando o número de colunas – Dias da Semana }
                        para indColuna ← 1 até 5 faça
                                escrever (‘Faturamento do Dia : ‘,
                                DiasSemana[indColuna]);
                                ler (VendasDiarias[indLinha, indColuna, indFaces]);
                        fim-para;
                fim-para;
        fim-para;
fim
```

Capítulo 4 - Registros

O conceito de registro visa facilitar o agrupamento de variáveis que não são do mesmo tipo, mas que guardam estreita relação lógica. Assim, registros correspondem a uma estrutura de dados heterogênea, ou seja, permite o armazenamento de informações de tipos diferentes. São localizados em posições de memória, conhecidos por um mesmo nome e individualizados por identificadores associados a cada conjunto de posições. Vamos pensar que precisamos fazer um algoritmo que leia os dados cadastrais dos funcionários de uma empresa. Somente com os campos nome e salário já temos uma divergência de tipos – caracter e real. Assim, precisamos de uma estrutura de dados que permita armazenar valores de tipos de diferentes – estamos diante dos registros.

Nas matrizes, a individualização de um elemento é feita através de índices, já no registro cada campo é individualizado pela referência do nome do identificador.

Veja a sintaxe da declaração de um registro:

```
tipo nome_registro = registro
                identificador-1 : tipo-1;
                identificador-2 : tipo-2;
                ...
                identificador-n : tipo-n;
fim-registro;
```

Para trabalharmos com um registro, devemos primeiramente criar um tipo registro. Depois, declaramos uma variável cujo tipo será este tipo registro.

Exemplo:

```
declare
        tipo DataExtenso = registro
                dia : inteiro;
                mes : string;
                ano : inteiro;
        fim-registro
        DataCarta : DataExtenso;
```

A sintaxe que representa o acesso ao conteúdo de um campo do registro é:

```
nome_registro.nome_campo;
```

A atribuição de valores aos registros é feita da seguinte forma:

```
nome_registro.nome_campo ← valor;
```

Suponha uma aplicação onde devemos controlar os dados de funcionários da empresa. Imagine que tenhamos fichas onde os dados estão logicamente relacionados entre si, pois constituem as informações cadastrais do mesmo indivíduo.

NOME DO FUNCIONÁRIO	ENDEREÇO		
JOÃO DA SILVA	RUA DA SAUDADE, 100 CASA 1		
CPF	ESTADO CIVIL	DATA NASC	ESCOLARIDADE
000.001.002-	CASADO	01/01/1960	SUPERIOR
CARGO		SALÁRIO	DATA DE ADMISSÃO
GERENTE DE CONTAS		1000,00	10/05/1997

NOME DO FUNCIONÁRIO	ENDEREÇO		
MARIA SANTOS	AVENIDA DESPERTAR, 1000 CASA 101-A		
CPF	ESTADO CIVIL	DATA NASC	ESCOLARIDADE
100.201.202-	SOLTEIRA	01/01/1980	2º GRAU COMPLETO
CARGO		SALÁRIO	DATA DE ADMISSÃO
DIGITADORA		650,00	01/08/1999

Cada conjunto de informações do funcionário pode ser referenciável por um mesmo nome, como por exemplo, FUNCIONARIO. Tais estruturas são conhecidas como registros e aos elementos do registro dá-se o nome de campos.

Assim, definiríamos um registro da seguinte forma:

```
tipo    Funcionario = registro
            nome : caracter;
            endereco : caracter;
            cpf : caracter;
            estado_civil : caracter;
            data_nascimento : caracter;
            escolaridade : caracter;
            cargo : caracter;
            salario : real;
            data_admissao : caracter;
 fim-registro;
```

Veja um exemplo de como podemos criar conjuntos de registros, utilizando vetores e/ou matrizes:

Exemplo:

```
declare
        tipo    Funcionario = registro
                nome : caracter;
                endereco : caracter;
                data_nascimento : caracter;
                data_admissao : caracter;
                cargo : caracter;
                salario : real;
        fim-registro;
        Funcionarios_Empresa : vetor[1:100] de Funcionario;
```

Capítulo 5 - Arquivos

Na maioria das vezes, desejaremos desenvolver um algoritmo de forma que os dados manipulados sejam armazenados por um período longo de tempo, e não somente durante o tempo de execução do algoritmo. Como a memória principal do computador é volátil, ou seja, ao ser desligado o computador, todos os dados da memória são perdidos, necessitamos de uma memória auxiliar que seja permanente, como por exemplo, um disquete ou o disco rígido (HD).

Assim, passamos a ter um novo conceito no mundo computacional – o de arquivos. Arquivo é um conjunto de registros armazenados em um dispositivo de memória auxiliar (secundária). Por sua vez, um registro consiste de um conjunto de unidades de informação logicamente relacionadas – os campos. Assim, podemos definir que um registro corresponde a um conjunto de campos de tipos heterogêneos. Veja que neste momento estamos tratando de registros físicos, ao contrário, do que vimos no item anterior, que são os registros lógicos.

O fato do arquivo ser armazenado em uma memória secundária, o torna independente de qualquer algoritmo, isto é, um arquivo pode ser criado, consultado, processado e eventualmente removido por algoritmos distintos.

Sendo o arquivo uma estrutura fora do ambiente do algoritmo, para que este tenha acesso aos dados do arquivo é necessária a operação de leitura do registro no arquivo. As operações básicas que podem ser feitas em um arquivo através de um algoritmo são: obtenção de um registro, inserção de um novo registro, modificação ou exclusão de um registro.

A disposição dos registros no arquivo – organização – oferece ao programador formas mais eficientes e eficazes de acesso aos dados. Vamos considerar, aqui, as duas principais formas de organização de arquivos: a seqüencial e a direta.

Organização Seqüencial: A principal característica da organização seqüencial é a de que os registros são armazenados um após o outro. Assim, tanto a leitura quanto a escrita, são feitas seqüencialmente, ou seja, a leitura de um determinado registro só é possível após a leitura de todos os registros anteriores e a escrita de um registro só é feita após o último registro.

Organização Direta: A principal característica da organização direta é a facilidade de acesso. Para se ter acesso a um registro de um arquivo direto, não é necessário pesquisar registro a registro, pois este pode ser obtido diretamente – acesso aleatório. Isto é possível porque a posição do registro no espaço físico do arquivo é univocamente determinada a partir de um dos campos do registro (chave), escolhido no momento de criação do arquivo.

O acesso a um arquivo dentro do algoritmo é feito através da leitura e escrita de registros. No algoritmo, o arquivo deve ser declarado e aberto, antes que tal acesso possa ser feito. No final do algoritmo, ou quando houver necessidade, o arquivo deve ser fechado.

A sintaxe da declaração de arquivos é :

```
nome_arquivo : arquivo organização de nome_registro;
```

onde:

organização – indica o tipo de organização do arquivo, que pode ser seqüencial ou direta.

nome_registro – nome do registro lógico que será usado para se ter acesso aos registros físicos do arquivo.

Exemplo:

```
tipo    Registro_Endereco = registro
        rua : caracter;
        numero : inteiro;
        bairro, cidade, uf, cep : caracter;
fim-registro;
Agenda : arquivo sequencial de Registro_Endereco;
```

Abertura de Arquivos

A declaração do arquivo é a definição, para o algoritmo, do modelo e dos nomes que estarão associados à estrutura de dados, isto é, ao arquivo.

A sintaxe da declaração de arquivos é :

```
abrir nome_ arq1, nome_arq2, ... , nome_arqn tipo_utilização;
```

onde:

tipo_utilização – especifica se o arquivo será usado somente para leitura, somente para escrita ou ambos, simultaneamente.

Exemplo:

abrir Agenda leitura;

abrir Agenda escrita;

abrir Agenda;

abrir Agenda, ContasPagar leitura;

Fechamento de Arquivos

Para se desfazer a associação entre o modelo e o arquivo físico, usa-se o comando de fechamento de arquivos, cuja sintaxe está a seguir:

```
fechar nome_ arq1, nome_arq2, ... , nome_arqn;
```

Comandos de Entrada (Leitura) e Saída (Escrita)

```
ler ( nome_arquivo , nome_registro );
```

```
escrever ( nome_arquivo , nome_registro );
```

Pesquisa de registro num arquivo seqüencial

```
Algoritmo Pesquisa1

declare

tipo    Dados_Aluno = registro
        matricula : integer;
        nome : caracter;
        data_nascimento : caracter;
        endereco : caracter;
fim-registro;

Alunos : arquivo sequencial de Dados_Aluno;
Encontrou : lógico;

início
        Abrir Alunos leitura;
        Encontrou ← Falso;

        enquanto (não Alunos.EOF) e (não Encontrou) faça
                Ler ( Alunos , Dados_Aluno )
                se Dados_Aluno.matricula = 159 então
                        Escrever('Aluno encontrado');
                        Encontrou ← Verdadeiro;
                fim-se;
        fim-enquanto

        Fechar Alunos;
fim
```

Obs: Na leitura de registros, fazemos uso de uma informação lógica associada ao arquivo, que indica se o ponteiro ultrapassou o último registro, ou seja, chegou ao fim do arquivo. No algoritmo, representamos essa informação pela função EOF.

Pesquisa de registro num arquivo direto

Para se acessar diretamente um registro, usa-se um comando para pesquisa da chave.

Veja sintaxe:

```
pesquisar ( nome_ arquivo , chave );
```

```
Algoritmo Pesquisa2
declare
tipo    Dados_Aluno = registro
        matricula : integer;
        nome : caracter;
        data_nascimento : caracter;
        end     ereco : caracter;
fim-registro;

Alunos : arquivo direto de Dados_Aluno;

início
        Abrir Alunos leitura
        se Pesquisar ( Alunos , '159' ) então
                Ler (Alunos, Dados_Aluno )
                Escrever('Aluno encontrado');
        fim-se;
        Fechar Alunos;
fim
```

Capítulo 6 - Procedimentos e Funções

Para construirmos grandes programas, necessitamos fazer uso da técnica de modularização. Esta técnica faz com que dividamos um grande programa em pequenos trechos de código, onde cada qual tem uma função bem definida. Assim, além da facilidade em lidar com trechos menores, ainda podemos fazer uso da reutilização de código, já que estes trechos devem ser bem independentes.

Assim, definimos módulo como um grupo de comandos, constituindo um trecho de algoritmo, com uma função bem definida e o mais independente possível em relação ao resto do algoritmo.

A maneira mais intuitiva de trabalharmos com a modularização de problemas é definir-se um módulo principal de controle e módulos específicos para as funções do algoritmo. Módulos de um programa devem ter um tamanho limitado, já que módulos muito grandes são difíceis de serem compreendidos.

Os módulos são implementados através de procedimentos ou funções.

Sintaxe de definição de um procedimento:

```
Procedimento Nome_Procedimento [ (parâmetros) ];
declare
        < variáveis locais >
início
        comando 1;
        comando 2;
        comando n;
fim
```

Os parâmetros podem ser passados por valor ou por referência. Um parâmetro passado por valor, não pode ser alterado pelo procedimento. Os parâmetros por referência são identificados usando-se a palavra VAR antes de sua declaração.

Sintaxe da chamada do procedimento:

```
Nome_Procedimento (<lista de parâmetros>);
```

Os valores passados como parâmetros na chamada de um procedimento, devem corresponder sequencialmente à ordem declarada.

A função é semelhante ao procedimento, todavia sua diferença consiste no fato de que um procedimento não retorna valor quando é chamado para execução, enquanto que a função retorna um valor.

Definição de uma função

```
Função Nome_Função [ (parâmetros) ] : valor_retorno;
declare
    < variáveis locais >
início
        comando 1;
        comando 2;
        comando n;
fim
```

Chamada da função

```
Nome_Função (<lista de parâmetros>);
```

Ou

```
Variável ← Nome_Função (<lista de parâmetros>);
```

Referências Bibliográficas

1) FARRER, Harry, BECKER, Christiano G., FARIA, Eduardo C., MATOS, Helton Fábio de, SANTOS, Marcos Augusto dos, MAIA, Miriam Lourenço. Algoritmos Estruturados. Rio de Janeiro: Editora Guanabara, 1989.

2) GUIMARÃES, Angelo de Moura, LAGES, Newton A de Castilho. Algoritmos e estruturas de dados. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora, 1985.

3) MECLER, Ian, MAIA, Luiz Paulo. Programação e lógica com Turbo Pascal. Rio de Janeiro: Campus, 1989.

4) SALVETTI, Dirceu Douglas, BARBOSA, Lisbete Madsen. Algoritmos. São Paulo: Makron Books, 1998.

5) SILVA, Joselias Santos da. Concursos Públicos – Raciocínio Lógico. São Paulo: R&A Editora Cursos e Materiais Didáticos, 1999.

6) WIRTH, Niklaus. Algoritmos e Estruturas de Dados. Rio de Janeiro: Editora Prentice-Hall do Brasil, 1986.