

Licenciatura

Computação

Algoritmos e Programação II

Ivo Mario Mathias



Ministério da Educação



UNIVERSIDADE ESTADUAL DE PONTA GROSSA



Núcleo de Tecnologia e Educação Aberta e a Distância



EDUCAÇÃO A DISTÂNCIA



EDUCAÇÃO A DISTÂNCIA

LICENCIATURA EM

Computação

ALGORITMOS E PROGRAMAÇÃO II

Ivo Mario Mathias



PONTA GROSSA / PARANÁ
2017

CRÉDITOS

Universidade Estadual de Ponta Grossa

Carlos Luciano Sant'ana Vargas

Reitor

Gisele Alves de Sá Quimelli

Vice - Reitor

Pró-Reitoria de Assuntos Administrativos

Amaury dos Martyres - Pró-Reitor

Pró-Reitoria de Graduação

Miguel Archanjo de Freitas Junior - Pró-Reitor

Núcleo de Tecnologia e Educação Aberta e a Distância

Eliane de Fátima Rauski - Coordenadora Geral

Marli de Fátima Rodrigues - Coordenadora Pedagógica

Sistema Universidade Aberta do Brasil

Eliane de Fátima Rauski - Coordenadora Geral

Marli de Fátima Rodrigues - Coordenadora Adjunta

Marcelo Ferrasa - Coordenador de Curso

Colaboradores em EAD

Dênia Falcão de Bittencourt

Cláudia Cristina Muller

Projeto Gráfico

Eloise Guenther

Colaboradores de Publicação

Denise Galdino - Revisão

Eloise Guenther - Diagramação

Todos direitos reservados ao Ministério da Educação

Sistema Universidade Aberta do Brasil

Ficha catalográfica elaborada pelo Setor Tratamento da Informação BICEN/UEPG

M431a Mathias, Ivo Mario
Algoritmos e programação II/ Ivo Mario Mathias. Ponta Grossa :
UEPG/ NUTEAD, 2017.
160 p. ; il.

Curso de Licenciatura em Computação. Universidade Estadual
de Ponta Grossa.

1. Variáveis compostas homogêneas. 2. Vetores - aplicações. 3.
Matrizes - aplicações. 4. Subalgoritmos. 5. Variáveis compostas
heterogêneas. I. T.

CDD : 005.113

UNIVERSIDADE ESTADUAL DE PONTA GROSSA
Núcleo de Tecnologia e Educação Aberta e a Distância - NUTEAD
Av. Gal. Carlos Cavalcanti, 4748 - CEP 84030-900 - Ponta Grossa - PR
Tel.: (42) 3220-3163
www.nutead.org
2017

APRESENTAÇÃO INSTITUCIONAL



A Universidade Estadual de Ponta Grossa é uma instituição de ensino superior estadual, democrática, pública e gratuita, que tem por missão responder aos desafios contemporâneos, articulando o global com o local, a qualidade científica e tecnológica com a qualidade social e cumprindo, assim, o seu compromisso com a produção e difusão do conhecimento, com a educação dos cidadãos e com o progresso da coletividade.

No contexto do ensino superior brasileiro, a UEPG se destaca tanto nas atividades de ensino, como na pesquisa e na extensão. Seus cursos de graduação presenciais primam pela qualidade, como comprovam os resultados do ENADE, exame nacional que avalia o desempenho dos acadêmicos e a situa entre as melhores instituições do país.

A trajetória de sucesso, iniciada há mais de 40 anos, permitiu que a UEPG se aventurasse também na educação a distância, modalidade implantada na instituição no ano de 2000 e que, crescendo rapidamente, vem conquistando uma posição de destaque no cenário nacional.

Atualmente, a UEPG é parceira do MEC/CAPES/FNDE na execução dos programas de Pró-Licenciatura e do Sistema Universidade Aberta do Brasil e atua em 40 polos de apoio presencial, ofertando, diversos cursos de graduação, extensão e pós-graduação a distância nos estados do Paraná, Santa Catarina e São Paulo.

Desse modo, a UEPG se coloca numa posição de vanguarda, assumindo uma proposta educacional democratizante e qualitativamente diferenciada e se afirmando definitivamente no domínio e disseminação das tecnologias da informação e da comunicação.

Os nossos cursos e programas a distância apresentam a mesma carga horária e o mesmo currículo dos cursos presenciais, mas se utilizam de metodologias, mídias e materiais próprios da EaD que, além de serem mais flexíveis e facilitarem o aprendizado, permitem constante interação entre alunos, tutores, professores e coordenação.

Esperamos que você aproveite todos os recursos que oferecemos para promover a sua aprendizagem e que tenha muito sucesso no curso que está realizando.

A Coordenação



SUMÁRIO

- PALAVRAS DO PROFESSOR 7
- OBJETIVOS E EMENTA 9

VARIÁVEIS COMPOSTAS HOMOGÊNEAS 11

- SEÇÃO 1- VARIÁVEL INDEXADA 12
- SEÇÃO 2- DECLARAÇÕES DE VARIÁVEIS INDEXADAS - VETORES - MATRIZES 13
- SEÇÃO 3- OPERAÇÕES BÁSICAS COM VARIÁVEIS INDEXADAS - VETORES - MATRIZES 16

APLICAÇÕES COM VETORES E MATRIZES 27

- SEÇÃO 1- PESQUISA SEQUENCIAL OU LINEAR 28
- SEÇÃO 2- PESQUISA BINÁRIA 34
- SEÇÃO 3- MÉTODO DA BOLHA DE CLASSIFICAÇÃO/ORDENAÇÃO (BUBBLE SORT) 41
- SEÇÃO 4- APLICAÇÕES COM MATRIZES 48

SUBALGORITMOS 69

- SEÇÃO 1- ESTRUTURA E FUNCIONAMENTO 71
- SEÇÃO 2- FUNÇÕES 74
- SEÇÃO 3- PROCEDIMENTOS 83
- SEÇÃO 4- PASSAGEM DE PARÂMETROS 88

VARIÁVEIS COMPOSTAS HETEROGÊNEAS 97

- SEÇÃO 1- INTRODUÇÃO 98
- SEÇÃO 2- REGISTROS 100
- SEÇÃO 3- TIPO DEFINIDO PELO PROGRAMADOR 102

RECURSIVIDADE 113

- PALAVRAS FINAIS 121
 - REFERÊNCIAS 123
 - ANEXOS - Respostas das Atividades 125
 - NOTAS SOBRE O AUTOR 149
-

PALAVRAS DO PROFESSOR



Parabéns! Você está iniciando a disciplina de Algoritmos e Programação II do curso de Licenciatura em Computação, significa que você já concluiu a disciplina Algoritmos e Programação I, já conhece os mecanismos e os conceitos para o desenvolvimento de algoritmos com as estruturas de programação sequencial, decisão e repetição.

O objetivo principal desta disciplina é um aprofundamento sobre algoritmos e programação de computadores, possibilitando ampliar o desenvolvimento do raciocínio lógico aplicado na solução de problemas em nível computacional.

Do mesmo modo que Algoritmos e Programação I, essa disciplina possui uma abordagem teórica e prática. Sendo assim, o acompanhamento integral deste livro é de fundamental importância para assimilar todas as definições, acompanhar os exemplos e fazer os exercícios propostos.

Continue os estudos com seriedade e responsabilidade, pois a sua dedicação será recompensada com conhecimento.

Bons Estudos!



OBJETIVOS E EMENTA



OBJETIVOS

Objetivo geral:

- O objetivo geral da disciplina de Algoritmos e Programação II é proporcionar um aprofundamento nos estudos sobre programação de computadores, possibilitando que o aluno amplie o raciocínio lógico aplicado à solução de problemas a nível computacional.

Objetivos específicos:

- Introduzir os conceitos básicos e definições sobre variáveis compostas homogêneas e heterogêneas.
- Aprender os métodos de ordenação básicos.
- Conhecer as pesquisas sequencial e binária.
- Entender o uso da recursividade.
- Saber escrever algoritmos e programas em Pascal com variáveis compostas homogêneas e heterogêneas, método de ordenação, pesquisas sequencial e binária, e recursividade.
- Praticar o processo de desenvolvimento de algoritmos e a implementação deles na linguagem de programação Pascal.

EMENTA

Variáveis compostas homogêneas. Pesquisa sequencial e binária. Método de ordenação. Subalgoritmos. Variáveis compostas heterogêneas. Recursividade. Aplicações de algoritmos em uma linguagem de programação.



Variáveis Compostas Homogêneas

OBJETIVOS DE APRENDIZAGEM

- Introduzir os conceitos básicos e definições sobre Variáveis Compostas Homogêneas.

ROTEIRO DE ESTUDOS

- SEÇÃO 1 – Variável Indexada
- SEÇÃO 2 – Declarações de Variáveis Indexadas - Vetores - Matrizes
- SEÇÃO 3 – Operações Básicas com Variáveis Indexadas - Vetores - Matrizes

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Nesta unidade, você tomará conhecimento das definições e conceitos básicos sobre Variáveis Compostas Homogêneas e exemplos de aplicações em algoritmos e programas de computador.

Quando você estudou sobre variáveis aprendeu que cada variável possui dois vínculos principais, o *dado* ao qual ela está relacionada e o *endereço* na memória do computador, onde efetivamente a informação está armazenada. Diante disto, nota-se que uma variável é uma entidade que armazena *apenas uma informação por vez*, de determinado tipo de dado.

Contudo, na prática é comum a necessidade de referenciar uma coleção de variáveis, dados, situação que remete a necessidade de criar uma variável para cada instância. Dependendo da quantidade isso pode ser inviável. Diante desse contexto, para resolver essa situação foi criado o conceito de *Variável Indexada*.

SEÇÃO 1 VARIÁVEL INDEXADA

Variável Indexada, também denominada de *Variável Composta Homogênea*, *Agregado Homogêneo* ou *Arranjos*, corresponde a um conjunto de variáveis do mesmo tipo, referenciáveis pelo mesmo nome e diferenciáveis entre si por meio de sua posição, índice, dentro do conjunto. Dimensão é o nome dado ao número, quantidade, de índices necessários à localização de um componente, elemento, dentro da variável indexada e que corresponde também ao número máximo de índices da variável.

Resumidamente, o que foi dito anteriormente é que em uma variável indexada é possível armazenar vários dados de um mesmo tipo básico

(inteiro, real, literal ou lógico) e a quantidade de dados é proporcional ao valor expresso no(s) índice(s). O valor máximo de índices de uma variável indexada é proporcional ao valor máximo de números inteiros que a linguagem de programação comporta. Nessa disciplina e do ponto de vista algorítmico, os números inteiros serão sempre armazenados em 2 bytes, o que equivale ao limite de 65535.

Quando uma variável indexada possui um único índice é chamada de **vetor** (unidimensional), quando possui dois ou mais índices é chamada de **matriz** (multidimensional – bidimensional, tridimensional ...).

Possivelmente, os termos **vetor** e **matriz** são mais comuns a você e, possivelmente, facilitarão o entendimento dessa modalidade de estrutura de dados, no contexto da programação de computadores. Veja a seguir, como essas variáveis são declaradas.

SEÇÃO 2

DECLARAÇÕES DE VARIÁVEIS INDEXADAS - VETORES - MATRIZES



Como regra geral, por ser o que ocorre na maioria das linguagens de programação, as variáveis devem ser declaradas no início do programa ou das rotinas que as utilizarão, ou seja, antes de serem utilizadas. Com relação à sintaxe, seguindo os mesmos procedimentos já adotados em Algoritmos e Programação I, em que, embora as sintaxes para declaração das variáveis possam diferir entre as diferentes linguagens de programação e entre os algoritmos, neste livro será adotada a sintaxe similar à linguagem de programação Pascal.

Em pseudocódigo a sintaxe é a seguinte:

Var

```
<nome_da_variável> : conjunto[dim1..dim2] de <tipo_de_dado>;
```

Onde:

- <tipo_de_dado> – inteiro, real, literal ou lógico.
- <nome_da_variável> – é o nome simbólico pelo qual o conjunto é identificado;
- dim1.. dim2 – correspondem aos valores mínimos e máximos dos índices da variável, dimensões.

ATENÇÃO



Identificação de variáveis – de forma geral, as regras para criação de nomes de variáveis são as seguintes: deve começar necessariamente com uma letra; não deve conter nenhum símbolo especial exceto a sublinha (_); não pode usar acentuação e nem cedilha; não pode haver espaços em branco entre as palavras que compõem o nome da variável; números podem ser utilizados e o ideal é que a identificação expresse o conteúdo da variável.

A seguir, alguns exemplos de declarações de variáveis indexadas em pseudocódigo.

EXEMPLO



```
Var NOMES : conjunto[1..100] de literal[30];
    IDADES : conjunto[1..100] de inteiro;
    SALARIOS : conjunto[1..100] de real;
    TABELA : conjunto[1..20 , 1..30] de inteiro;
```



Do ponto de vista algorítmico e por padronização, nessa disciplina, assim como já foi considerado em Algoritmos e Programação I, os tipos de dados serão armazenados conforme segue:

- números inteiros - 2 bytes;
- números reais - 4 bytes;
- literal - cada caractere em 1 byte;
- lógicos - 1 byte.

Detalhamento de cada uma das quatro variáveis exemplificadas anteriormente:

- NOMES: **conjunto[1..100] de literal[30]** – corresponde a uma variável indexada que pode armazenar até 100 nomes, com no máximo 30 caracteres (bytes) cada um.
- IDADES: **conjunto[1..100] de inteiro** – variável do tipo inteiro, para a qual o compilador reservará 200 bytes na memória do computador, que correspondem a 2 bytes para cada dado do tipo inteiro, $2 * 100 = 200$, sendo assim, nessa variável é possível armazenar até 100 números do tipo inteiro.
- SALARIOS: **conjunto[1..100] de real** – variável do tipo real, que possuirá 400 bytes reservados na memória. Neste exemplo, seguindo a mesma analogia da variável anterior, é possível armazenar até 100 números do tipo real, conforme expressa a dimensão, 100.
- TABELA: **conjunto[1..20, 1..30] de inteiro** – corresponde a uma *matriz bidimensional*, capaz de armazenar 600 números do tipo inteiro, ou seja, $20 * 30 = 600$ elementos.

Resumidamente, pode-se dizer que as variáveis: NOMES, IDADES e SALARIOS, são *vetores*, pois possuem apenas uma dimensão e a variável TABELA por ter duas dimensões trata-se de uma *matriz*.

Agora, observe as mesmas variáveis anteriormente exemplificadas, codificadas em Linguagem de Programação Pascal.



```
Var NOMES: array[1..100] of string[30];  
  
IDADES: array[1..100] of integer;  
  
SALARIOS: array[1..100] of real;  
  
TABELA: array[1..20 , 1..30] of integer;
```

Assim como já ocorreu em outros exemplos é fácil perceber que as palavras-reservadas, que fazem parte das instruções, apenas foram traduzidas para o inglês, reforçando a ideia de que a sintaxe adotada em pseudocódigo é similar à do Pascal.

O próximo passo é saber como as operações básicas podem ser executadas, de modo tal que os valores armazenados possam ser manuseados.

SEÇÃO 3

OPERAÇÕES BÁSICAS COM VARIÁVEIS INDEXADAS - VETORES - MATRIZES

.....

O acesso aos dados é similar ao de variáveis simples, a diferença é o fato de que não é possível operar de forma direta o conjunto todo, mas o acesso deve ser individual a cada componente pela especificação da posição, de cada elemento, por meio dos índices.

Analise alguns exemplos a seguir:



- NOMES [3] - especifica o terceiro elemento do conjunto NOMES;
- IDADES [2] - especifica o segundo elemento do conjunto IDADES;
- SALARIOS [20] - especifica o vigésimo elemento do conjunto SALARIOS;
- TABELA [5, 11] - especifica o elemento da quinta linha e décima primeira coluna do conjunto TABELA.

Um bom modo de entender como as estruturas dos vetores e matrizes são representados, na memória do computador, é por meio de imagens. A seguir duas ilustrações, figuras 1 e 2, que ajudam a abstrair a ideia, uma vez que as variáveis indexadas, vetores e matriz são armazenados na memória do computador em endereços contíguos, um elemento após o outro, de acordo com a ordem crescente dos índices, posições.

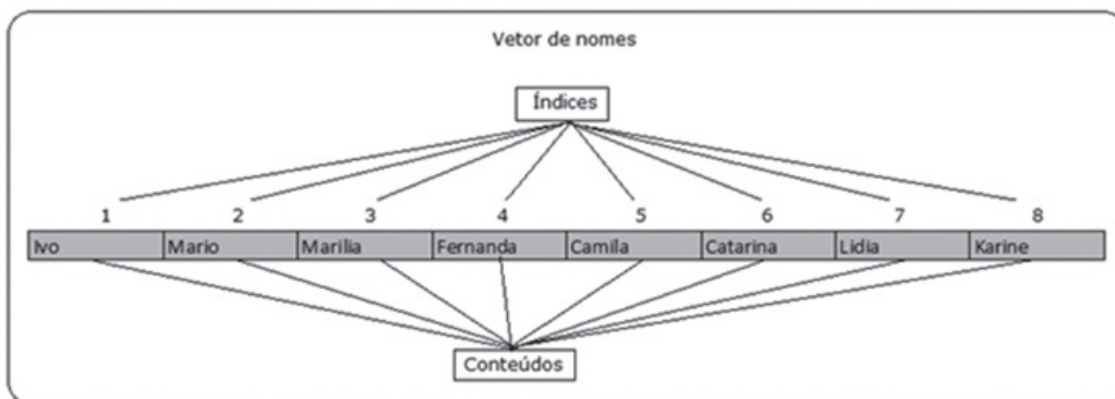


Figura 1: Estrutura genérica de um Vetor de nomes.

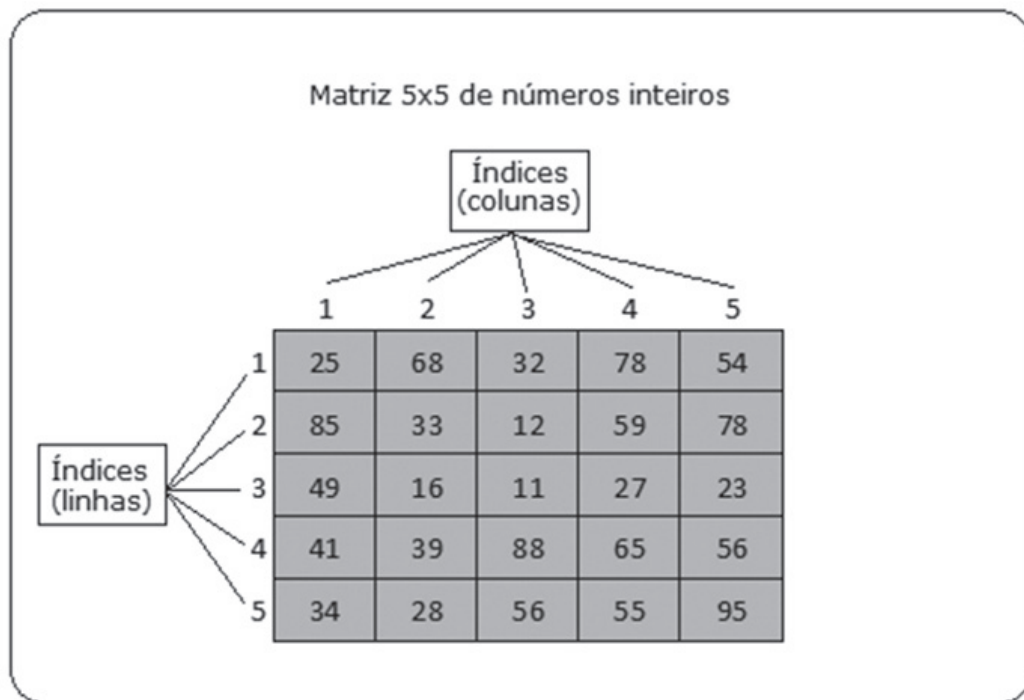


Figura 2: Estrutura genérica de uma Matriz de números inteiros.

Diante do fato de que o acesso deve ser feito individualmente a cada elemento de uma variável indexada, a seguir, você vai aprender como realizar as operações de atribuição, leitura e escrita, por meio dos índices.

ATRIBUIÇÃO

Uma vez que, em variáveis indexadas o acesso a cada elemento deve ser individual, especificando seus índices, observe nos exemplos a seguir, que esta é a única diferença em comparação a atribuição de variáveis simples.

EXEMPLO



- NOMES[3] := 'Marilia';
- IDADES[3] := 45;
- SALARIOS[3] := 1000,00;
- TABELA[i,j] := 88;

Analisando os exemplos anteriormente ilustrados, percebe-se que no caso das três primeiras variáveis indexadas, *NOMES*, *IDADES* e *SALARIOS*, foram atribuídos valores conforme cada tipo de dado correspondente, em que os valores foram alocados na posição 3 de cada conjunto.

No caso do exemplo da variável *TABELA*, como é uma matriz bidimensional, foram usadas duas variáveis, $\langle i, j \rangle$, para especificar os índices. Diante dessa situação, é importante salientar que os **índices**, de qualquer variável indexada, podem ser representados por **variáveis** ou **constantes**, mas sempre do tipo **inteiro**.

É oportuno lembrar que, assim como ocorre com variáveis simples, as variáveis indexadas também podem ser operadores de expressões, desde que seja especificado, individualmente, o elemento do conjunto que será utilizado.

LEITURA

A leitura também é feita passo a passo, um componente por vez, usando a mesma sintaxe da instrução primitiva de entrada de dados, $\langle \text{Leia } \langle \text{nome_da_variável} \rangle; \rangle$, explicitando a posição (índice) do componente lido.



Leia (NOMES[i]);

ESCRITA

Com sintaxe similar à da leitura, a escrita é feita passo a passo, um componente por vez, com a mesma sintaxe da instrução primitiva de saída de dados, $\langle \text{Escreva } \langle \text{nome_da_variável} \rangle; \rangle$, explicitando a posição (índice) do componente lido.



Escreva (NOMES[i]);

Um bom modo de consolidar os assuntos dessa seção é por meio de exemplos práticos, a seguir estão ilustrados dois exemplos, um em pseudocódigo e o equivalente em Pascal, envolvendo preliminarmente o que você precisa saber para realizar as operações básicas com um vetor. A figura 3 exhibe o algoritmo com a leitura e escrita de um vetor de números inteiros.



```
Algoritmo Leitura_escrita_vetor_inteiros;  
  
  Var  
    Numeros : conjunto [1..10] de inteiro;  
    i       : inteiro;  
  
  Início  
  
    Para i de 1 até 10 incr de 1 faça  
      leia(Numeros[i]);  
    Fim_para;  
  
    Para i de 1 até 10 incr de 1 faça  
      escreva(Numeros[i]);  
    Fim_para;  
  
  Fim.
```

Figura 3: Algoritmo para leitura e escrita de um vetor de números inteiros.

Antes de prosseguir, observe atentamente o algoritmo exibido anteriormente, procurando entender o funcionamento e mentalizar qual seria a saída dele se fosse executado. Caso julgue necessário, elabore um teste de mesa, que foi ensinado em Algoritmos e Programação I.

Uma vez que tenha entendido o funcionamento do algoritmo em questão, continue com a leitura e analise agora o programa da figura 4, que é o mesmo algoritmo da figura 3, codificado em Pascal. Do mesmo modo que fez com o algoritmo, analise atentamente os detalhes do programa comparando com o pseudocódigo.

ATENÇÃO



A partir de agora, tanto nos algoritmos como nos programas em Pascal, estarão sendo utilizadas as três estruturas de programação que você já aprendeu: *sequenciais*, *decisão* e *repetição*. Portanto, parte-se do princípio que você já domina esses assuntos e consegue analisar os respectivos códigos. Contudo, caso tenha dúvidas recorra ao livro da disciplina Algoritmos e Programação I.

```

Program Leitura_escrita_vetor_inteiros;

Var
  Numeros : array [1..10] of integer;
  i       : integer;

Begin

  For i:= 1 to 10 do
    Begin
      write ('Entre com o ',i,'o. número:');
      read(Numeros[i]);
    End;

  writeln ('Listagem completa do vetor:');
  For i:= 1 to 10 do
    writeln(Numeros[i]);

  readkey;
End.

```

Figura 4: Programa em Pascal para leitura e escrita de um vetor de números inteiros.

Para facilitar o entendimento dos exemplos das figuras 3 e 4, a seguir estão destacados alguns detalhes importantes:

- Um detalhe que se destaca é o fato de ser utilizada uma estrutura de repetição, *Para-passo*, para acessar, individualmente, todos os componentes do vetor. Qualquer uma das estruturas de repetição (*Para-passo*, *Enquanto* ou *Repita*) podem ser utilizadas, apesar da *Para-passo* ser a mais comum.
- Ainda na questão do uso de estruturas de repetição para realizar operações com vetores e matrizes, essa prática é a mais usual devido ao fato que geralmente as operações são com o conjunto todo ou parte dele e, é notório que facilita o acesso a todos os componentes do conjunto e com poucas linhas de código.
- Outro destaque é com relação ao uso da variável *i*, uma vez que ela teve uma dupla finalidade nos exemplos em questão, ser a variável de controle do *Para-passo* e também ser o índice de acesso aos componentes do vetor. Na prática, sempre que possível deve-se otimizar o uso de variáveis, reutilizando-as para diversas operações quando possível.
- É oportuno salientar que quando se escreve algoritmos, procura-se ser o mais sucinto possível, ou seja, não preocupar-se em demasia com a interface do programa para o usuário. Porém, quando um algoritmo é codificado em uma linguagem de programação, o ideal é incluir mensagens na tela do computador, para que durante a execução do programa o usuário saiba o que o programa requer em situações de entrada de dados e também quando exhibe os respectivos resultados, situação que ocorreu em relação ao exemplo da figura 4.



É fundamental destacar que é imprescindível que todos os exemplos em Pascal que forem apresentados, neste livro, sejam implementados em Pascalzim e testados. Caso contrário, pode comprometer o aprendizado do que está sendo ensinado, uma vez que, praticar é primordial.

.....

Para ampliar os conhecimentos com operações de vetores, estude e analise os dois próximos exemplos, figuras 5 e 6, seguindo os mesmos procedimentos em relação aos exemplos anteriores, ou seja, analise individualmente cada código e compare as diferenças, não se esquecendo de implementar e executar o programa em Pascalzim.

EXEMPLO



```

Algoritmo Leitura_soma_vetor_reais;

Var
  Valores : conjunto[1..10] de real;
  Soma : real;
  i      : inteiro;

Início

  Para i de 1 até 10 incr de 1 faça
    leia(Valores[i]);
  Fim_para;

  Soma := 0;

  Para i de 1 até 10 incr de 1 faça
    Soma := Soma + Valores[i];
  Fim_para;

  escreva ('Somatório dos valores do vetor: ', Soma);

Fim.
  
```

Figura 5: Algoritmo para leitura e soma de um vetor de números reais.



```
Program Leitura_soma_vetor_reais;  
  
Var  
  Valores : array [1..10] of real;  
  Soma : real;  
  i      : integer;  
  
Begin  
  
  For i:= 1 to 10 do  
    Begin  
      write ('Entre com o ',i,'o. número:');  
      read(Valores[i]);  
    End;  
  
  Soma := 0;  
  
  For i:= 1 to 10 do  
    Soma := Soma + Valores[i];  
  
  writeln ('Somatório dos valores do vetor: ', Soma);  
  
  readkey;  
End.
```

Figura 6: Programa em Pascal para leitura e soma de um vetor de números reais.

Para reforçar o entendimento a respeito dos dois códigos anteriormente ilustrados, a seguir estão descritos os principais detalhes:

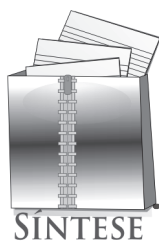
- Trata-se de exemplos de vetor com números reais, em que além da leitura também ocorre o somatório de todos os valores do vetor;
- Em razão do somatório, foi declarada a variável *Soma* do mesmo tipo do vetor, *real*, a qual teve que ser inicializada com zero (*Soma:=0*) pelo fato de que ela recebe valores de si mesma.

É oportuno relembrar que um algoritmo pode ser implementado em uma linguagem de programação, com códigos-fonte diferentes, mas que ao serem executados geralmente obtêm-se resultados iguais. Partindo dessa premissa, desenvolva a atividade a seguir.



ATIVIDADES

Reescreva e implemente em Pascalzim o programa da figura 6, de modo tal que seja utilizada apenas uma vez a estrutura de repetição, *Para-passo*, e se obtenha o mesmo resultado como saída do programa, ou seja, o somatório do conteúdo todo do vetor.



SÍNTESE

Nesta unidade, você teve a oportunidade de aprender o que são variáveis compostas homogêneas e a importância delas no contexto da programação de computadores. Visto que, com variáveis simples é possível armazenar apenas um valor de cada vez e, com variáveis indexadas pode-se armazenar um conjunto de valores de qualquer um dos tipos de dados.

Apreendeu também que as variáveis indexadas podem ser chamadas de vetores e matrizes e, embora elas possuam apenas uma identificação, nome, o acesso aos dados é feito por meio dos índices, que possibilitam o acesso individual aos dados.

Na questão de vetores e matrizes foi destacado que o diferencial é com relação à quantidade de índices, ou seja, quando uma variável indexada possui um único índice é chamada de vetor (unidimensional), quando possui dois ou mais índices é chamada de matriz (multidimensional – bidimensional, tridimensional ...).

Com relação às operações básicas, foi visto que seguem as mesmas sintaxes das instruções primitivas de atribuição, leitura e escrita, apenas com o detalhe de incluir o índice correspondente ao dado que se quer operar.

Foram ilustrados exemplos práticos, que auxiliam no entendimento de como os conjuntos podem ser operados e, neste quesito o principal destaque foi com relação ao uso de estruturas de repetição.

A próxima unidade vai abordar outros exemplos práticos de aplicações utilizando vetores e matrizes.



Aplicações com Vetores e Matrizes

OBJETIVOS DE APRENDIZAGEM

- Apresentar aplicações comuns com vetores: pesquisa e classificação ou ordenação.

ROTEIRO DE ESTUDOS

- SEÇÃO 1 – Pesquisa Sequencial ou Linear
- SEÇÃO 2 – Pesquisa Binária
- SEÇÃO 3 – Método da Bolha de Classificação/Ordenação (*Bubble Sort*)

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Na prática, as possibilidades de aplicações de vetores em programas de computador são extensas. Por exemplo, quando você utiliza um caixa eletrônico, que nada mais é que um computador com programas que são executados para atendê-lo é notório o acesso à tabelas, listas, extratos, em que não fica difícil perceber uma possível aplicação de conjuntos de dados, que podem estar armazenados em vetores e/ou matrizes.

Relacionado a situações similares como as citadas anteriormente, nesta unidade serão apresentadas duas aplicações clássicas com vetores: *pesquisa* e a *classificação* ou *ordenação*.

Classificação ou *Ordenação* consiste em arranjar os elementos de um conjunto em uma determinada ordem, segundo um critério específico, por exemplo, ordem crescente ou decrescente de um conjunto de dados numéricos ou de um conjunto de dados literais.

O problema da pesquisa ou busca pode ser definido por um conjunto de elementos, em que cada um pode ser identificado por uma chave. O objetivo da pesquisa é localizar, no conjunto o elemento que corresponde a uma chave específica. Os métodos mais comuns são a pesquisa sequencial ou linear e a pesquisa binária.

SEÇÃO 1 PESQUISA SEQUENCIAL OU LINEAR

É considerado o método mais objetivo para se encontrar um elemento particular num conjunto, por exemplo, em um vetor. Consiste na verificação, comparação, de cada componente do conjunto sequencialmente (um após o outro) com uma chave de busca, até que o elemento desejado seja ou não encontrado.

Como você já conhece as operações básicas com vetores, pode-se de imediato apresentar um algoritmo e o respectivo programa em Pascal, desenvolvidos com uma pesquisa sequencial para ler e pesquisar nomes em um vetor, figuras 7 e 8 respectivamente.



EXEMPLO

```

Algoritmo Pesquisa_Sequencial;

Var
  Nomes : conjunto [1..1000] de literal[30];
  Chave : literal[30];
  i, Ultimo : inteiro;
  Achou : lógico;

Início
  Para i de 1 até 1000 incr de 1 faça
    escreva ('Entre com o ', i, 'o. nome:');
    leia(Nomes[i]);
    Se (Nomes[i] = 'fim') então
      Ultimo := i - 1;
      saia;
    senão
      Ultimo := i;
    Fim_se;

    escreva ('Entre com o nome a ser pesquisado:');
    leia(Chave);
    i := 1;
    Achou := FALSO;

    Enquanto((i <= Ultimo) .E. (Achou = FALSO)) faça
      Se (Nomes[i] = Chave) então
        Achou := VERDADEIRO
      senão
        i := i + 1;
      Fim_se;
    Fim_enquanto;

    Se (ACHOU = VERDADEIRO) então
      escreva('Nome: ', Chave, ' - encontrado')
    senão
      escreva('Nome: ', Chave, ' - NÃO encontrado');
    Fim_se;
Fim.
  
```

Figura 7: Algoritmo – Pesquisa Sequencial – vetor de nomes.



```
Program Pesquisa_Sequencial;

Var
  Nomes : array [1..1000] of string[30];
  Chave : string[30];
  i, Ultimo : integer;
  Achou : boolean;

Begin
  For i := 1 to 1000 do
    Begin
      write ('Entre com o ', i, 'o. nome:');
      read(Nomes[i]);
      If (Nomes[i] = 'fim') then
        Begin
          Ultimo := i - 1;
          break;
        End
      else
        Begin
          Ultimo := i;
        End;
      End;
    End;

    write ('Entre com o nome a ser pesquisado:');
    readln(Chave);
    i := 1;
    Achou := FALSE;

    While((i <= Ultimo) and (Achou = FALSE)) do
      Begin
        If (Nomes[i] = Chave) then
          Achou := TRUE
        else
          i := i + 1;
        End;
      End;

      If (Achou = TRUE) then
        writeln('Nome: ', Chave, ' - encontrado')
      else
        writeln('Nome: ', Chave, ' - NÃO encontrado');

      readkey;

    End.
```

Figura 8: Programa em Pascal – Pesquisa Sequencial – vetor de nomes.

Antes de prosseguir com as leituras, analise linha a linha de cada um dos exemplos apresentados, implemente o programa no Pascalzim, execute o programa e faça algumas simulações com nomes fictícios. Caso julgue necessário, elabore um teste de mesa para auxiliar no entendimento das rotinas destes códigos.



Ao analisar os códigos em referência, deve ter percebido algumas novidades, tanto em relação a comandos novos como detalhes de lógica. A seguir, uma análise dos principais aspectos que merecem destaque.

- Dois comandos novos foram apresentados, *saia* (algoritmo) e *break* (Pascal), a finalidade de ambos é forçar a saída de uma estrutura de repetição (*Para-passo, Enquanto, Repita / While, For, Repeat*). Uma vez que, a estrutura de decisão que controla os comandos *saia/break* seja verdadeira, o próximo comando a ser executado é o comando logo, após o final da estrutura de repetição, caso contrário a estrutura de repetição é executada mais uma vez.
- Nos exemplos anteriormente ilustrados, observe que a finalidade do uso dos comandos *saia/break* está relacionada ao fato que o vetor suporta até 1000 nomes, portanto, é ilógico digitar 1000 nomes para poder testar o programa. Sendo assim, ao digitar a palavra *fim* torna verdadeira a estrutura de decisão, que tem em seu comando composto as instruções *saia/break*.
- Outro detalhe importante em relação ao comando composto da estrutura de decisão, que controla os comandos *saia/break* é a expressão `< Ultimo := i - 1; >`. Você deve ter percebido que a variável *Ultimo* tem como finalidade armazenar o valor do índice correspondente ao do último nome da lista, que tenha sido digitado. O fato da variável ser decrementada em *-1* é pelo fato de que o índice em que está sendo digitada a palavra *fim* não corresponde ao último nome válido.
- Outro destaque é o uso de duas modalidades de estruturas de repetição nestes códigos, a primeira com *Para-passo/ For* e a segunda *Enquanto/While*, sendo que a segunda, `< Enquanto((i <= Ultimo) .E. (Achou = FALSO)) faça / While((i <= Ultimo) and (Achou = FALSE)) do >`, possui duas variáveis de controle: a variável *Ultimo* e a variável lógica *Achou* que se torna *Verdadeira* quando o nome armazenado na variável *Chave* corresponde a algum nome constante na lista. Por outro lado, a variável *Ultimo* juntamente com a variável *i* fazem com que a leitura da lista ocorra até o último nome da lista, caso o nome procurado não conste na lista. Por outro lado, lembrando de como o operador lógico *.E./and* funciona, se uma das subexpressões for *Falsa* a expressão toda se torna *Falsa* e a estrutura de repetição é interrompida.

É oportuno lembrar que um determinado problema a ser resolvido computacionalmente, pode ser resolvido com códigos diferentes, mas é possível obter o mesmo resultado, como saída. Diante dessa premissa, resolva a atividade proposta a seguir.



Reescreva o algoritmo/programa das figuras 7 e 8, de modo tal que a estrutura de repetição *Para-passo* seja substituída por *Repita* e quando houver nomes duplicados na lista o algoritmo/programa localize-os e conte quantas vezes ocorrem.

Inicialmente, procure resolver apenas utilizando o seu conhecimento e os códigos das figuras em referência. Após ter encontrado uma possível solução, recorra à proposta de solução e as observações apresentadas no final do livro.



SEÇÃO 2

PESQUISA BINÁRIA

Trata-se de um método que pode minimizar o esforço computacional na pesquisa de elementos de um vetor (lista), desde que os dados estejam previamente classificados, segundo algum critério (crescente ou decrescente).

Um bom modo de entender esse método é por meio de uma descrição narrativa, a saber:

- A partir de um vetor classificado - ordem crescente.
- Por meio dos índices, localizar o elemento que divide o vetor ao meio, aproximadamente.
- Se o elemento que divide o vetor ao meio for o procurado, a pesquisa é bem-sucedida e é interrompida.
- Se o elemento procurado for menor que o elemento divisor, repete-se o processo na primeira metade e se for maior, na segunda metade.
- O procedimento se repete até que se localize o valor procurado, ou até que não haja nenhum trecho do vetor a ser pesquisado.

Uma vez que tenha entendido a descrição narrativa, analise os exemplos de algoritmo e programa de pesquisa binária das figuras 9 e 10. Lembrando que é fundamental prosseguir com os próximos assuntos, desde que o que foi estudado anteriormente já esteja consolidado.

Siga em frente!

EXEMPLO



```

Algoritmo Pesquisa_Binaria_Nomes;

Var
  Nomes : conjunto [1..1000] de literal[30];
  Chave : literal[30];
  i, Primeiro, Medio, Ultimo : inteiro;
  Achou : lógico;

Início
  i := 1;

  Repita
    escreva ('Entre com o ', i, 'o. nome:');
    leia(Nomes[i]);
    Se (Nomes[i] = 'fim') então
      Ultimo := i - 1
    senão
      i := i + 1;
      Ultimo := i;
    Fim_se;
  Até_que (Nomes[i] = 'fim');

  escreva ('Entre com o nome a ser pesquisado:');
  leia(Chave);

  Primeiro := 1;
  Achou := FALSO;

  Enquanto((Primeiro <= Ultimo) .E. (ACHOU = FALSO)) faça
    Medio := (Primeiro + Ultimo)/2;
    Se (Chave = Nomes[Medio]) então
      Achou := VERDADEIRO
    senão
      Se (Chave < Nomes[Medio]) então
        Ultimo := Medio - 1
      senão
        Primeiro := Medio + 1;
      Fim_se;
    Fim_se;
  Fim_enquanto;

  Se (Achou = VERDADEIRO) então
    escreva('Nome: ', Chave, ' - encontrado ')
  senão
    escreva('Nome: ', Chave, ' - NÃO encontrado');
  Fim_se;

Fim.

```

Figura 9: Algoritmo – Pesquisa Binária – vetor de nomes.

EXEMPLO



```

Program Pesquisa_Binaria_Nomes;

Var
  Nomes : array [1..1000] of string[30];
  Chave : string[30];
  i, Primeiro, Medio, Ultimo : integer;
  Achou : boolean;

Begin
  i := 1;
  // Entrada dos nomes
  Repeat
    write ('Entre com o ', i, 'o. nome:');
    read(Nomes[i]);
    If (Nomes[i] = 'fim') then
      Ultimo := i - 1
    else
      Begin
        i := i + 1;
        Ultimo := i;
      End;
    Until (Nomes[i] = 'fim');

    write ('Entre com o nome a ser pesquisado:');
    readln(Chave);
    { Inicialização de variáveis }
    Primeiro := 1;
    Achou := FALSE;
  // Rotina da Pesquisa binária
  While((Primeiro <= Ultimo) and (ACHOU = FALSE)) do
    Begin
      Medio := trunc((Primeiro + Ultimo)/2);
      If (Chave = Nomes[Medio]) then
        Begin
          Achou := TRUE
        End
      else
        If (Chave < Nomes[Medio]) then
          Ultimo := Medio - 1
        else
          Primeiro := Medio + 1;
      End;
    (* Saída do programa *)
    If (Achou = TRUE) then
      writeln('Nome: ', Chave, ' - encontrado ')
    else
      writeln('Nome: ', Chave, ' - NÃO encontrado');

    readkey;

End.

```

Figura 9: Programa em Pascal – Pesquisa Binária – vetor de nomes.

Nos códigos anteriormente apresentados, você deve ter percebido que há novidades em termos de comandos, além da pesquisa binária em si.

- Inicialmente, focando uma análise no algoritmo da pesquisa binária, percebe-se que a essência dela encontra-se na estrutura de repetição *Enquanto/While*, em que se divide sucessivamente ao meio, aproximadamente, `< Medio := (Primeiro + Ultimo)/2;>`, o valor total dos índices e a cada novo passo, concentra-se a divisão no intervalo onde possivelmente o elemento correspondente à chave de pesquisa possa ser encontrado.
- Agora, ao observar no código em Pascal a expressão que faz a divisão, `< Medio := trunc((Primeiro + Ultimo)/2); >`, aparece um novo comando, *trunc*, o qual tem por finalidade obter a parte inteira de um valor do tipo real. Isso ocorre porque nas divisões da pesquisa binária, quando o valor dos índices é ímpar gera uma parte fracionária, que deve ser descartada, pois os índices das variáveis indexadas sempre devem ser inteiros, conforme já visto.
- O uso da função *trunc* pode ser utilizado em qualquer situação em que seja necessário obter apenas a parte inteira de um número do tipo real.
- Outra novidade são os trechos com *comentários*, por exemplo, `{ Inicialização de variáveis }`. Os comentários são usados como parte do texto do código-fonte, com a finalidade de documentar trechos de código e não afetam a execução do programa, ou seja, os comentários são ignorados pelo compilador. A definição de um comentário, na linguagem Pascal, pode ser feita de dois modos:
 - *Comentário de bloco* – é feito com o uso de pares de chaves, `{comentário}`, ou parêntesis com asterisco, `(*comentário*)`. Esse tipo de comentário pode ser escrito em mais de uma linha, mas o texto deve ser delimitado pelos símbolos em referência.
 - *Comentário de linha* – é aquele que tem apenas no início da linha um par de barras, `// Rotina da Pesquisa binária.`

- Os comentários em códigos-fonte são úteis, pois auxiliam a identificar/documentar trechos de códigos para futuras consultas e manutenções. É um modo de fixar lembretes no código-fonte. Neste livro, será minimizado o uso de comentários para não tornar muito extensos os códigos-fonte, mas na prática o ideal que você utilize sempre que julgar necessário.

Agora que você já estudou como funciona o algoritmo de pesquisa binária, resolva as atividades propostas a seguir.



Desenvolva um teste de mesa, tomando como referência o algoritmo da pesquisa binária, para a situação a seguir: na tabela denominada *Nomes[]*, simule a localização da existência do nome 'Zenildo' e do nome 'Amélia' (dois testes de mesa).

Nomes[]	
Dim	Conteúdo
1	Amélia
2	Camila
3	Fernanda
4	Ivo
5	Lauro
6	Marília
7	Tatiana
8	Xandra
9	Zenildo
10	Zilda





Reescreva o algoritmo/programa das figuras 9 e 10, de modo tal que o conjunto de dados sejam números inteiros. Faça o uso de comentários para documentar como você está resolvendo a atividade e para facilitar estudos futuros deste código.



Apesar do método da pesquisa binária ter a desvantagem de exigir que o vetor seja previamente classificado, o que não acontece com a pesquisa sequencial, a pesquisa binária em média é mais rápida que a sequencial.

Para suprir essa necessidade de ordenar um vetor, o próximo assunto a ser estudado é o método de classificação chamado de *Bolha*.

SEÇÃO 3

MÉTODO DA BOLHA DE CLASSIFICAÇÃO/ORDENAÇÃO (BUBBLE SORT)

.....

É intuitivo afirmar que o acesso aos elementos de um conjunto de dados classificados torna-se mais fácil e rápido. Em um processo de classificação ou ordenação, o objetivo é que o resultado obtido seja um conjunto com elementos organizados segundo algum critério, por exemplo, crescente, do menor para o maior.

São vários os métodos de classificação que podem ser encontrados na literatura, tais como: *Bubble Sort* (Método Bolha), *Insert Sort* (Método de Inserção), *Selection Sort* (Método de Seleção), *Quicksort*, *Heapsort*, *Bucket Sort* (Bin Sort), *Radix Sort*, *Merge Sort* (Ordenação por Intercalação). Neste livro, os estudos serão focados no *Bubble Sort* (Método Bolha), por ser um dos métodos mais populares e relativamente simples.

Assim como foi com relação à pesquisa, a descrição narrativa facilita o entendimento lógico do método, a saber:

- Percorrer o conjunto comparando os elementos vizinhos entre si.
- Caso estejam fora de ordem, trocar a posição entre eles.
- Repetir os processos anteriores até o final do vetor.
- Na primeira varredura verificar se o último elemento (maior de todos) do conjunto já está no seu devido lugar, para o caso de ordenação crescente.
- A segunda varredura é análoga a primeira e vai até o penúltimo elemento.
- O processo é repetido até que seja feito um número de varreduras igual ao número de elementos a serem ordenados menos um.
- Ao final do processo o conjunto estará classificado segundo o critério escolhido.

O método bolha (*Bubble Sort*) baseia-se no princípio de que, em um conjunto de dados há valores menores, mais leves, e maiores, mais pesados. Diante desse fato, como bolhas, os valores leves sobem no conjunto um por vez, ao passo que os mais pesados descem em direção ao final do conjunto.

Para concretizar a ideia do método de classificação *Bubble Sort*, nas figuras 11 e 12, estão ilustrados um algoritmo e respectivo programa em Pascal, em que os dados a serem classificados são números inteiros.



```

Algoritmo Metodo_Bolha_Bubble_Sort_Inteiros;

Var
  Numeros: conjunto [1..1000] de inteiro;
  i, j, Aux, Ultimo : inteiro;

Início
  i := 1;
  Repita
    escreva ('Entre com o ', i, 'o. número: ');
    leia(Numeros[i]);
    Se (Numeros[i] < 0) então
      Ultimo := i - 1
    senão
      i := i + 1;
      Ultimo := i;
    Fim_se;
  Até_que ((Numeros[i] < 0) .OU. (i = 1000));

  j := Ultimo;
  Enquanto(j > 1) faça
    Para i de 1 até j-1 faça
      Se Numeros[i] > Numeros[i+1] então
        Aux := Numeros[i];
        Numeros[i] := Numeros[i+1];
        Numeros[i+1] := Aux;
      Fim_se;
    Fim_para;
    j := j - 1;
  Fim_enquanto;

  Para i de 1 até Ultimo faça
    escreva(i, 'o. - ', Numeros[i]);
  Fim_para;

End.

```

Figura 11: Algoritmo – Bubble Sort – vetor de números inteiros.



```
Program Metodo_Bolha_Bubble_Sort_Inteiros;

Var
  Numeros: array [1..1000] of integer;
  i, j, Aux, Ultimo : integer;

Begin
  // Entrada dos números
  i := 1;
  Repeat
    write ('Entre com o ',i,'o. número: ');
    read(Numeros[i]);
    If (Numeros[i] < 0) then
      Ultimo := i - 1
    else
      Begin
        i := i + 1;
        Ultimo := i;
      End;
  Until ((Numeros[i] < 0) or (i = 1000));

  j := Ultimo;

  // Rotina do Método Bolha
  While(j > 1) do
    Begin
      For i := 1 to j-1 do
        Begin
          If Numeros[i] > Numeros[i+1] then
            Begin
              Aux := Numeros[i];
              Numeros[i] := Numeros[i+1];
              Numeros[i+1] := Aux;
            End;
          End;
        j := j - 1;
      End;

  // Saída do programa
  For i := 1 to Ultimo do
    writeln(i,'o. - ',Numeros[i]);

  readkey;

End.
```

Figura 12: Programa em Pascal – *Bubble Sort* – vetor de números inteiros.

Seguindo o mesmo método já adotado em relação aos algoritmos e programas anteriores, analise cuidadosamente cada um dos códigos, tire as suas conclusões, para então prosseguir com a leitura.

Os códigos das figuras 11 e 12, além de tratar-se de um assunto novo, cabem também alguns destaques em relação a assuntos já abordados.

- Tanto nos códigos em referência como nos anteriores, é visível a importância da indentação, pois sem ela dificulta o entendimento dos códigos. Principalmente na identificação das diferentes estruturas de um algoritmo ou programa. Outro fator crucial é a visualização dos *aninhamentos*, quando uma estrutura de programação está dentro do conjunto de comandos (comando composto) de uma outra construção.
- No quesito *aninhamento*, a rotina do método bolha possui um *aninhamento* triplo, formado por três estruturas, duas de repetição, que são a *Enquanto/While* e o *Para-passo/For*, e na estrutura mais interna, uma estrutura de decisão, *Se/If*.
- Outro item importante que a rotina do método bolha possui é a utilização de uma variável *Aux* (auxiliar), que tem por finalidade armazenar temporariamente o valor de um elemento do vetor e transferir esse conteúdo para outra posição. Observe e analise que esta técnica é útil em qualquer situação quando se quer trocar de posição os conteúdos de um conjunto, pois, sem o uso de uma variável auxiliar (intermediária), fatalmente o valor de uma das posições seria perdido.

As atividades, a seguir, vão ajudá-lo a consolidar os conhecimentos dessa seção. Resolva integralmente antes de comparar com as soluções no final do livro.



Desenvolva um teste de mesa, tomando como referência o Algoritmo de classificação *Bubble Sort* para a ordenação da tabela, a seguir, denominada *Numeros[]*:

Numeros[]	
Dim	Valores
1	223
2	789
3	768
4	001
5	987
6	345
7	006
8	026
9	121
10	003





ATIVIDADES

Reescreva o algoritmo/programa das figuras 11 e 12, de modo tal que o conjunto de dados a serem classificados sejam *nomes*.

Melhore o programa, para que ao finalizar a entrada dos nomes, o algoritmo/programa reconheça a palavra *fim*, tanto minúscula como maiúscula.



ANOTAÇÕES

A seguir você vai aprender os procedimentos básicos no desenvolvimento prático com matrizes, finalizando essa Unidade.

SEÇÃO 4

APLICAÇÕES COM MATRIZES

As matrizes aparecem em diversas aplicações práticas, seja na própria informática, em cálculos matemáticos, editores de imagem, estrutura de hardware, onde o próprio teclado dos computadores tem a configuração realizada por um sistema de matrizes, entre outras aplicações.

Não que isso seja regra, mas em geral a matriz bidimensional é a mais utilizada. Na matemática, uma matriz $m \times n$ corresponde a uma tabela de m linhas e n colunas. Geralmente representada por uma letra maiúscula, por exemplo, A , enquanto os seus termos e elementos são representados pela mesma letra, mas minúscula, acompanhada de dois índices, por exemplo, $a_{1,1} - a_{1,2} - a_{1,3} \dots a_{m,n}$, onde o primeiro número representa a linha e o segundo a coluna em que o elemento está localizado. A figura 13 representa graficamente a estrutura de uma matriz bidimensional.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & a_{m,3} & \dots & a_{m,n} \end{bmatrix}$$

Figura 13: Estrutura de uma matriz bidimensional $m \times n$.

Na informática, como já visto, uma matriz é uma variável indexada para a qual existem estruturas de programação adequadas ao tratamento delas. Na sequência, serão exibidos e detalhados alguns exemplos de algoritmos e programas em Pascal, que possibilitam o tratamento de

matrizes. Você vai perceber que as rotinas, as quais serão estudadas, podem ser aplicadas na resolução de problemas práticos com matrizes.

Os primeiros exemplos, figuras 14 e 15, ilustram um algoritmo e o respectivo programa em Pascal para leitura e escrita de uma matriz de números inteiros.

EXEMPLO



```

Algoritmo Leitura_Escrita_Matriz;

Var
  i,j: inteiro;
  Matriz: conjunto [1..3, 1..3] de inteiro;

Início
  Para i de 1 até 3 incr de 1 faça
    Para j de 1 até 3 incr de 1 faça
      escreva('Entre com o valor da posição [',i,',',j, ' ] : ');
      leia(Matriz[i,j]);
    Fim_para;
  Fim_para;

  escreva('Conteúdo da Matriz: ');

  Para i de 1 até 3 incr de 1 faça
    Para j de 1 até 3 faça
      escreva(Matriz[i,j]);
    Fim_para;
  Fim_para;

Fim.
  
```

Figura 14: Algoritmo – leitura e escrita – matriz bidimensional de números inteiros.

EXEMPLO



```
Program Leitura_Escrita_Matriz;  
  
Var  
  i,j: integer;  
  Matriz: array [1..3, 1..3] of integer;  
  
Begin  
  // Leitura dos dados da matriz  
  For i := 1 to 3 do  
    For j := 1 to 3 do  
      Begin  
        write('Entre com o valor da posição [' ,i, ', ',j, ' ] : ');  
        readln(Matriz[i,j]);  
      End;  
  
  // Exibição dos dados da matriz  
  writeln('Conteúdo da Matriz: ');  
  
  For i := 1 to 3 do  
    Begin  
      For j := 1 to 3 do  
        write(Matriz[i,j]:3);  
  
      writeln;  
    End;  
  
  readkey;  
  
End.
```

Figura 15: Programa em Pascal – leitura e escrita – matriz bidimensional de números inteiros.

À medida que os estudos avançam você deve ter percebido uma gradativa evolução nos detalhes lógicos de construção dos diversos códigos já apresentados. E assim como já fez com os anteriores, antes de prosseguir é fundamental que faça uma leitura minuciosa de cada uma das linhas, tanto do algoritmo como do programa, de modo que entenda o raciocínio lógico empregado, bem como a sintaxe de cada comando.

Os códigos das figuras 14 e 15 possuem alguns detalhes novos e relembrando, para um melhor entendimento, todos os programas em Pascal constantes deste livro devem ser implementados e executados no Pascalzim.

- Primeiro detalhe que merece destaque é o fato que para o tratamento de variáveis indexadas, geralmente é necessário uma quantidade de estruturas de repetições proporcional ao número de índices. Como o exemplo ilustrado anteriormente é uma matriz bidimensional foram usadas duas estruturas de repetição, *Para-passo/For*, aninhadas uma com a outra.
- Outro aspecto importante em relação ao aninhamento de estruturas de repetição é entender o funcionamento de ambas as repetições, ou seja, quantas vezes cada uma é executada. Para entender essa situação, basta abstrair o conceito de que a estrutura de repetição mais externa, que controla o índice i , faz o tratamento das linhas e a interna, do índice j , faz o tratamento das colunas. Sendo assim, para cada passo de i , a execução de j ocorre o número de vezes que está programado, ao encerrar, retorna ao próximo passo de i , e assim sucessivamente. Observe atentamente a execução do programa quando da entrada dos dados.

Para ajudar a fixar a leitura e escrita de matrizes, elabore as atividades a seguir.

Agora que você já sabe os procedimentos para a leitura e escrita de matrizes, a seguir serão apresentadas outras operações comuns em tratamento de matrizes.

Nas figuras 16 e 17, você encontra um algoritmo e o respectivo programa para tratamento de uma matriz de números inteiros, com as seguintes rotinas;

- uso de constantes predefinidas;
- inicialização de matriz.

EXEMPLO



```

Algoritmo Uso_constantes_e_Inicializacao_Matriz;

Var
    i,j,num: inteiro;
    Matriz: conjunto [1..100, 1..100] de inteiro;

Constante
    linhas = 10;
    colunas = 10;

Início
    num := 1;
    Para i de 1 até linhas incr de 1 faça
        Para j de 1 até colunas incr de 1 faça
            Matriz[i,j] := num;
            num := num + 1;
        Fim_para;
    Fim_para;

    escreva('Conteúdo da Matriz: ');

    Para i de 1 até linhas incr de 1 faça
        Para j de 1 até colunas incr de 1 faça
            escreva(Matriz[i,j]);
        Fim_para;
    Fim_para;

Fim.
    
```

Figura 16: Algoritmo – uso de constantes e inicialização de matriz de números inteiros.



EXEMPLO

```
Program Uso_constantes_e_Inicializacao_Matriz;

Var
  i,j,num: integer;
  Matriz: array [1..100, 1..100] of integer;

Const // Declaração de constantes
  linhas = 10;
  colunas = 10;

Begin
  // Inicialização da matriz com números sequenciais
  num := 1;
  For i := 1 to linhas do
    For j := 1 to colunas do
      Begin
        Matriz[i,j] := num;
        num := num + 1;
      End;
  // Exibição dos dados da matriz
  writeln('Conteúdo da Matriz: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz[i,j]:5);
      writeln;
    End;

  readkey;

End.
```

Figura 17: Programa em Pascal – uso de constantes e inicialização de matriz de números inteiros.

Novamente, antes de prosseguir, faça um estudo detalhado dos dois códigos, procure entender o funcionamento do programa. Identifique as novidades, aí então prossiga com a leitura dos destaques a seguir:

- Primeiro destaque é o uso de *constantes*, que correspondem a um valor fixo, atribuído a um identificador, que no decorrer do algoritmo ou execução de um programa sempre terá o mesmo valor. Nos exemplos em questão, foram usadas duas constantes para delimitar a quantidade de *linhas* e *colunas* da matriz, respectivamente. Com relação à sintaxe e em comparação ao pseudocódigo e o código em Pascal, apenas uma pequena diferença nas palavras, *constante* e *const*.
- O uso de constantes é uma prática útil na programação de computadores, pois como pode ser observado nos códigos em referência, as constantes foram utilizadas em dois trechos, sem a necessidade de usar constantes numéricas.
- Outra vantagem no uso de constantes é o fato de que por meio delas podem ser alterados valores nas linhas de código dos algoritmos/programa sem reescrevê-los. Imagine que você tem um código extenso para diversos tratamentos de matrizes e quer alterar o tamanho das matrizes. Com o uso de constantes basta alterar o valor no local da declaração, que automaticamente esse valor será assimilado em todos os locais do código onde as constantes aparecem. Por outro lado, se estiver utilizando constantes numéricas, teria que editar todas as linhas onde fosse necessário, ainda correndo o risco de esquecer a alteração de alguma linha de código.
- Outro destaque importante é a rotina que faz a inicialização da matriz, atribuindo valores a todos os elementos. Esse procedimento é comum, principalmente nos casos em que os elementos da matriz são utilizados para cálculos, ou ainda, quando se quer padronizar com valores iniciais a matriz. Situação que, na prática, facilita saber quais elementos já foram ou não utilizados, por exemplo, inicializar uma matriz de números com zeros, ou valores negativos.

Uma vez que tenha assimilado os destaques anteriormente citados, estude os códigos das figuras 18 e 19, em que são exemplificados manipulações de dados entre as linhas e colunas de uma matriz.

EXEMPLO



```
Algoritmo Tratamento_de_Matriz_linhas_colunas;  
  
Var  
  i, j, num, soma_lin_5, soma_col_10 : inteiro;  
  Matriz: conjunto [1..100, 1..100] de inteiro;  
  
Constante  
  linhas = 10;  
  colunas = 10;  
  
Início  
  num := 1;  
  Para i de 1 até linhas incr de 1 faça  
    Para j de 1 até colunas incr de 1 faça  
      Matriz[i,j] := num;  
      num := num + 1;  
    Fim_para;  
  Fim_para;  
  
  escreva('Conteúdo da Matriz: ');  
  
  Para i de 1 até linhas incr de 1 faça  
    Para j de 1 até colunas incr de 1 faça  
      escreva(Matriz[i,j]);  
    Fim_para;  
  Fim_para;  
  
  soma_lin_5 := 0;  
  Para j de 1 até colunas incr de 1 faça  
    soma_lin_5 := soma_lin_5 + Matriz[5,j];  
  Fim_para;  
  
  soma_col_10 := 0;  
  Para i de 1 até linhas incr de 1 faça  
    soma_col_10 := soma_col_10 + Matriz[i,10];  
  Fim_para;  
  
  escreva('Saídas do programa: ');  
  escreva('Somatório dos elementos da linha 5:',soma_lin_5);  
  escreva('Somatório dos elementos da coluna 10:',soma_col_10);  
  
Fim.
```

Figura 18: Algoritmo – tratamento de matriz – linhas e colunas.



EXEMPLO

```
Program Tratamento_de_Matriz_linhas_colunas;

Var
  i,j,num,soma_lin_5,soma_col_10 : integer;
  Matriz: array [1..100, 1..100] of integer;

Const // Declaração de constantes
  linhas = 10;
  colunas = 10;

Begin
  // Inicialização da matriz com números sequenciais
  num := 1;
  For i := 1 to linhas do
    For j := 1 to colunas do
      Begin
        Matriz[i,j] := num;
        num := num + 1;
      End;

  // Exibição dos dados da matriz
  writeln('Conteúdo da Matriz: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz[i,j]:5);
      writeln;
    End;

  // Somatório dos elementos da linha 5
  soma_lin_5 := 0;
  For j := 1 to colunas do
    soma_lin_5 := soma_lin_5 + Matriz[5,j];

  // Somatório dos elementos da coluna 10
  soma_col_10 := 0;
  For i := 1 to linhas do
    soma_col_10 := soma_col_10 + Matriz[i,10];

  // Saídas do programa
  writeln;writeln('Saídas do programa: ');
  writeln;
  writeln('Somatório dos elementos da linha 5: ',soma_lin_5);
  writeln('Somatório dos elementos da coluna 10: ',soma_col_10);

  readkey;

End.
```

Figura 19: Programa em Pascal – tratamento de matriz – linhas e colunas.

Considerando que você já tenha feito uma análise dos códigos, a seguir o principal destaque a respeito deles:

- Em uma matriz, quando se manipula apenas uma linha ou coluna, observa-se nos exemplos, que apenas uma estrutura de repetição foi utilizada, para cada caso e, o índice da linha ou coluna em tratamento é fixo, por meio de uma constante, neste caso, numérica, podendo ser por identificador.

- Exemplo: `soma_lin_5 := soma_lin_5 + Matriz[5, j];`

Continuando com exemplos de tratamento de matrizes, estude os códigos das figuras 20 e 21, que ilustram manipulações de dados entre as diagonais, principal e secundária, de uma matriz quadrada, quando o número de linhas e colunas é o mesmo.



```
Algoritmo Tratamento_de_Matriz_diagonais;

Var
  i,j,num: inteiro;
  soma_diag_P,soma_diag_S: inteiro;
  Matriz: conjunto [1..100, 1..100] de inteiro;

Constante
  linhas = 10;
  colunas = 10;

Início
  num := 1;
  Para i de 1 até linhas incr de 1 faça
    Para j de 1 até colunas incr de 1 faça
      Matriz[i,j] := num;
      num := num + 1;
    Fim_para;
  Fim_para;

  escreva('Conteúdo da Matriz: ');
  Para i de 1 até linhas incr de 1 faça
    Para j de 1 até colunas incr de 1 faça
      escreva(Matriz[i,j]);
    Fim_para;
  Fim_para;

  soma_diag_P := 0;
  j := 1;
  Para i de 1 até colunas incr de 1 faça
    soma_diag_P := soma_diag_P + Matriz[i,j];
    j := j + 1;
  Fim_para;

  soma_diag_S := 0;
  j := colunas;
  Para i de 1 até colunas incr de 1 faça
    soma_diag_S := soma_diag_S + Matriz[i,j];
    j := j - 1;
  Fim_para;

  escreva('Saídas do programa: ');
  escreva('Somatório dos elementos da diagonal principal : ',soma_diag_P);
  escreva('Somatório dos elementos da diagonal secundária: ',soma_diag_S);

Fim.
```

Figura 20: Algoritmo – tratamento de matriz – diagonais.



```

Program Tratamento_de_Matriz_diagonais;

Var
  i,j,num: integer;
  soma_diag_P,soma_diag_S: integer;
  Matriz: array [1..100, 1..100] of integer;

Const // Declaração de constantes
  linhas = 10;
  colunas = 10;

Begin
  // Inicialização da matriz com números sequenciais
  num := 1;
  For i := 1 to linhas do
    For j := 1 to colunas do
      Begin
        Matriz[i,j] := num;
        num := num + 1;
      End;

  // Exibição dos dados da matriz
  writeln('Conteúdo da Matriz: ');
  For i := 1 to linhas do
    Begin
      for j := 1 to colunas do
        write(Matriz[i,j]:5);

      writeln;
    End;

  // Somatório dos elementos da diagonal principal
  soma_diag_P := 0;
  j := 1;
  For i := 1 to colunas do
    Begin
      soma_diag_P := soma_diag_P + Matriz[i,j];
      j := j + 1;
    End;

  // Somatório dos elementos da diagonal secundaria
  soma_diag_S := 0;
  j := colunas;
  For i := 1 to colunas do
    Begin
      soma_diag_S := soma_diag_S + Matriz[i,j];
      j := j - 1;
    End;

  // Saídas do programa
  writeln; writeln('Saídas do programa: ');
  writeln;
  writeln('Somatório dos elementos da diagonal principal : ',soma_diag_P);
  writeln('Somatório dos elementos da diagonal secundária: ',soma_diag_S);

  readkey;

End.

```

Figura 21: Programa em Pascal – tratamento de matriz – diagonais.

Para facilitar o entendimento dos exemplos de códigos exibidos nas figuras 20 e 21, a figura 22 ilustra onde são representadas as diagonais principal e secundária de uma matriz.

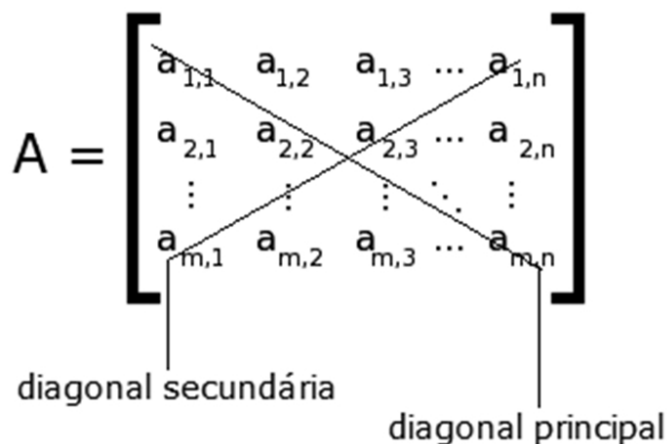


Figura 22: Diagonais de uma matriz $m \times n$.

Nesse contexto, a seguir os principais aspectos a serem observados em relação aos códigos em referência.

- Em ambos os casos, diagonal principal e secundária, nas soluções apresentadas percebe-se que foram resolvidas com uma estrutura de repetição apenas.
- Na solução computacional, o tratamento da diagonal principal é resolvido pela variação simultânea, incremento, das variáveis i e j , iniciando-se em 1 até o valor máximo da dimensão da matriz inicializada, representada pelas constantes *linhas* e *colunas*.
- Na diagonal secundária, observe que a variável j , que representa as colunas é inicializada pelo valor máximo da dimensão, *colunas*, e durante a execução a variável j é decrementada, enquanto que i , linhas, é incrementada.

Ampliando as possibilidades de operações com matrizes estude atentamente os exemplos das figuras 23 e 24, em que é realizado o somatório entre duas matrizes e o resultado é atribuído em uma terceira matriz.



```
Algoritmo Somatrio_entre_Matrizes;

Var
  i,j,num: inteiro;
  Matriz_1: conjunto [1..100, 1..100] de inteiro;
  Matriz_2: conjunto [1..100, 1..100] de inteiro;
  Matriz_3: conjunto [1..100, 1..100] de inteiro;

Constante
  linhas = 10;
  colunas = 10;

Início
  num := 1;

  Para j de 1 até colunas incr de 1 faça
    Matriz_1[i,j] := num;
    Matriz_2[i,j] := num*2;
    num := num + 1;
  Fim_para;
Fim_para;

// Exibição dos dados das matrizes-----
escreva('Conteúdo da Matriz 1: ');
Para i de 1 até linhas incr de 1 faça
  Para j de 1 até colunas incr de 1 faça
    escreva(Matriz_1[i,j]);
  Fim_para;
Fim_para;

escreva('Conteúdo da Matriz 2: ');
Para i de 1 até linhas incr de 1 faça
  Para j de 1 até colunas incr de 1 faça
    escreva(Matriz_2[i,j]);
  Fim_para;
Fim_para;

// Atribuindo o somatório das matrizes 1 e 2 na 3-----
Para i de 1 até linhas incr de 1 faça
  Para j de 1 até colunas incr de 1 faça
    Matriz_3[i,j] := Matriz_1[i,j] + Matriz_2[i,j];
  Fim_para;
Fim_para;

// Saída do algoritmo-----
escreva('Conteúdo da Matriz 3: ');
Para i de 1 até linhas incr de 1 faça
  Para j de 1 até colunas incr de 1 faça
    escreva(Matriz_3[i,j]);
  Fim_para;
Fim_para;

Fim.
```

Figura 23: Algoritmo – somatório entre matrizes.



```

Program Somatorio_entre_Matrizes;

Var
  i,j,num: integer;
  Matriz_1: array [1..100, 1..100] of integer;
  Matriz_2: array [1..100, 1..100] of integer;
  Matriz_3: array [1..100, 1..100] of integer;

Const // Declaração de constantes
  linhas = 10;
  colunas = 10;

Begin
  // Inicialização das matrizes
  // Matriz 1 com números sequenciais
  // Matriz 2 com dobro dos valores da Matriz 1
  num := 1;
  For i := 1 to linhas do
    For j := 1 to colunas do
      Begin
        Matriz_1[i,j] := num;
        Matriz_2[i,j] := num*2;
        num := num + 1;
      End;

  // Exibição dos dados das matrizes-----
  writeln('Conteúdo da Matriz 1: ');
  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz_1[i,j]:5);
        writeln;
      End;

  writeln('Conteúdo da Matriz 2: ');
  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz_2[i,j]:5);
        writeln;
      End;

  // Atribuindo o somatório das matrizes 1 e 2 na 3-----
  For i := 1 to linhas do
    For j := 1 to colunas do
      Matriz_3[i,j] := Matriz_1[i,j] + Matriz_2[i,j];

  // Saída do programa-----
  writeln('Conteúdo da Matriz 3: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz_3[i,j]:5);
        writeln;
      End;

  readkey;

End.

```

Figura 24: Programa em Pascal – somatório entre matrizes.

A seguir os três principais detalhes sobre os exemplos das figuras 23 e 24.

- O primeiro aspecto em relação aos exemplos anteriores é o fato de terem sido declaradas três matrizes. Evidentemente, significa que podem ser declaradas quantas matrizes forem necessárias em um programa, mas também com dimensões diferentes. Neste exemplo, são iguais por força das operações propostas.
- Um segundo item é o fato de ter sido utilizada uma mesma rotina com duas estruturas de repetição aninhadas para inicializar duas matrizes simultaneamente. Esse fato indica que dependendo das necessidades de um programa, deve-se procurar otimizar códigos, de modo que determinadas operações possam ser realizadas por uma mesma estrutura de programação.
- O terceiro detalhe é apenas um destaque sobre a situação que envolve operações com as matrizes, ou seja, tanto neste exemplo como nos anteriores, percebe-se que as operações podem ser realizadas livremente e que o principal cuidado é a indicação dos índices corretos para cada operação que se deseja realizar.

Sem dúvida, você já deve ter percebido que além da análise dos exemplos ilustrados, outro modo de efetivamente fixar os conteúdos é por meio de exercícios. Sendo assim, desenvolva as atividades propostas a seguir, sempre reforçando, inicialmente faça com os conhecimentos que você adquiriu até onde consta a atividade e com o auxílio dos exemplos. Após ter resolvido, compare com as soluções apresentadas no final do livro.



Escreva um algoritmo e o implemente em linguagem Pascal com as seguintes rotinas:

- que inicialize uma matriz 5 x 5 com valores inteiros ímpares, iniciando em 1;
- exiba a matriz lida;
- efetue a troca do conteúdo, elementos, da 2ª linha com o da 5ª linha;
- efetue a troca do conteúdo, elementos, da 3ª coluna com o da 4ª coluna;
- exiba a matriz modificada.



Escreva um algoritmo e o implemente em linguagem Pascal com as seguintes rotinas:

- que inicialize uma matriz 10 x 10 com valores inteiros pares, iniciando em 20;
- exiba a matriz inicializada;
- efetue a troca do conteúdo, elementos, da diagonal principal com os da secundária;
- exiba a matriz modificada.



Escreva um algoritmo e o implemente em linguagem Pascal com as seguintes rotinas:

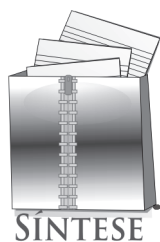
- que inicialize uma matriz 10 x 10 com valores inteiros pares, iniciando em 20;
- exiba a matriz inicializada;
- efetue a troca do conteúdo, elementos, da 1ª linha com os da 10ª coluna;
- exiba a matriz modificada.



ANOTAÇÕES



Como você deve ter resolvido as atividades propostas e aprendeu todos os conteúdos desta unidade, está apto a prosseguir para um assunto novo, subalgoritmos.



Nesta unidade, *Aplicações com Vetores e Matrizes*, como o próprio título expressa, você aprendeu a resolver várias operações práticas envolvendo variáveis indexadas. Com vetores estudou de modo prático como efetuar pesquisas sequenciais e binárias.

Nas sequenciais, sendo o método mais objetivo para encontrar um elemento particular num conjunto, estudou que consiste na verificação de cada componente do conjunto sequencialmente (um após o outro) até que o elemento desejado seja ou não encontrado.

Com relação à pesquisa binária, foi visto que corresponde a um método que divide sucessivamente um conjunto ao meio e, se o elemento procurado for menor que o elemento divisor, repete-se o processo na primeira metade e se for maior, na segunda metade. O procedimento se repete até que se localize o valor procurado ou até que não haja nenhum trecho do conjunto a ser pesquisado.

Um requisito da pesquisa binária é que os dados estejam previamente classificados, segundo algum critério (crescente ou decrescente). Uma das vantagens da pesquisa binária é o fato de minimizar o esforço computacional na pesquisa de elementos de um conjunto, conforme a posição do elemento procurado.

Outro assunto estudado nesta unidade foi Método da Bolha de Classificação (*Bubble Sort*), que se baseia no princípio de que, em um conjunto de dados há valores menores, mais leves, e maiores, mais pesados. Diante desse fato, como bolhas, os valores leves sobem no conjunto um por vez, ao passo que os mais pesados descem em direção ao final do conjunto. Algoritmicamente falando, o objetivo é percorrer o conjunto comparando os elementos vizinhos entre si; caso estejam fora de ordem, os mesmos trocam de posição entre si; repetir os processos anteriores até o final do vetor; na primeira varredura verifica se o último elemento (maior de todos) do conjunto já está no seu devido lugar, para o caso de ordenação crescente; a segunda varredura é análoga a primeira e vai até o penúltimo elemento; o processo é repetido até que seja feito um número de varreduras igual ao número de elementos a serem ordenados menos um; ao final do processo o conjunto estará classificado segundo o critério escolhido.

Para finalizar a unidade, o foco da abordagem foi aplicações com matrizes bidimensionais. Diversos exemplos de algoritmos e programas foram apresentados para o tratamento de matrizes, envolvendo operações de cálculos entre os elementos da matriz, manuseio de dados entre linhas, colunas e diagonais da matriz.

Possivelmente, nos exemplos de códigos que você estudou até agora e, principalmente na última seção dos tratamentos de matrizes, você deve ter percebido várias rotinas, similares e que se repetiam num mesmo algoritmo/programa, por exemplo, leitura e escrita de matrizes. O próximo assunto, subalgoritmos, tem como finalidade minimizar as redundâncias de códigos, buscando gerar rotinas genéricas que podem ser reutilizadas.

.....

Subalgoritmos

OBJETIVOS DE APRENDIZAGEM

- Saber subdividir algoritmos e programas, facilitando o seu entendimento.
- Aprender a estruturar algoritmos e programas, facilitando a detecção de erros e a documentação de sistemas.
- Entender como os sistemas podem ser modularizados.
- Aprender a escrever algoritmos e programas com Funções e Procedimentos.

ROTEIRO DE ESTUDOS

- SEÇÃO 1 – Estrutura e Funcionamento
- SEÇÃO 2 – Funções
- SEÇÃO 3 – Procedimentos
- SEÇÃO 4 – Passagem de parâmetros

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Algo que você deve estar percebendo no desenvolvimento da disciplina, e isso inclui Algoritmos e Programação I também, é que os códigos gradativamente vão se tornando mais complexos. Pode-se afirmar que a complexidade dos códigos está diretamente relacionada a aplicação que se destinam. Por sua vez, aplicações mais complexas geram códigos mais extensos o que geralmente dificulta a escrita e manutenção desses códigos.

Por outro lado, os códigos de sistemas computacionais extensos e complexos podem ser divididos em soluções menores, módulos, em que são gerados algoritmos reduzidos, subalgoritmos, que formam o sistema como um todo. Subalgoritmos podem ser definidos também como subrotinas, pois representam uma parcela de código, que tem por finalidade uma tarefa bem definida e pode ser executado tantas vezes quanto for necessário num mesmo sistema computacional.

A partir da próxima seção, você vai aprender como modularizar os algoritmos e programas.

Siga em frente!

SEÇÃO 1

ESTRUTURA E FUNCIONAMENTO

.....

Decompor um problema que será resolvido computacionalmente é um fator determinante para a redução da complexidade. A complexidade é algo que geralmente remete a algo de difícil compreensão ou entendimento, que um problema pode apresentar. É natural afirmar que ao decompor um problema em subproblemas menores, divide-se também a complexidade e por consequência pode-se simplificar a resolução do problema. Situação que permite ao desenvolvedor focar a atenção em menores trechos dos códigos, bem como, auxilia na compreensão do sistema como um todo.

Diante desse contexto, os subalgoritmos correspondem a porções menores de um sistema computacional complexo, que possuem em si uma porção da solução do problema como um todo. Como regra, o algoritmo completo é dividido num algoritmo principal e diversos subalgoritmos, conforme a necessidade. O algoritmo principal, como o próprio nome expressa é responsável pela execução inicial de um sistema, meio pelo qual os demais subalgoritmos são acessados para que o sistema funcione integralmente.

No que se refere à execução de um sistema computacional, implementado com subalgoritmos, a partir da execução do algoritmo principal, à medida que nas linhas de comandos surgirem chamadas a subalgoritmos, a execução do algoritmo principal é interrompida, a seguir a execução ocorre com os comandos do subalgoritmo, ao término, a execução retorna a linha seguinte no algoritmo principal. Também é possível que um subalgoritmo chame outro por meio do mesmo mecanismo.

Além das vantagens já descritas com relação aos subalgoritmos, podem-se elencar mais algumas, tais como:

- facilita a detecção de erros nos códigos;
- auxilia na manutenção dos sistemas;
- favorece a organização e documentação dos sistemas;
- possibilita a reutilização de códigos.

Após as considerações iniciais, ficou evidente que fazer uso de subalgoritmos no desenvolvimento de sistemas computacionais, traz benefícios. A partir de agora, você vai aprender como os subalgoritmos podem ser definidos em um algoritmo e também em Pascal.

Inicialmente, antes de exemplificar, estude a seguir os itens que definem um subalgoritmo:

- **Nome** – nome simbólico pelo qual o subalgoritmo será acessado, chamado.
- **Parâmetros** – são canais pelos quais dados são transferidos pelo algoritmo chamador a um subalgoritmo e vice-versa (bidirecional).
- **Variáveis Globais** – são aquelas declaradas no início de um algoritmo, elas são visíveis, isto é, podem ser usadas tanto no algoritmo principal e por todos os demais subalgoritmos existentes no sistema.
- **Variáveis locais** – são aquelas definidas dentro do próprio subalgoritmo e só podem ser utilizadas pelo mesmo.
- **Corpo** – onde se encontram os comandos (instruções), que serão executados cada vez que o subalgoritmo for chamado.



A composição dos nomes simbólicos de subalgoritmos seguem as mesmas regras de nomes de variáveis e algoritmos/programas. Palavras-reservadas e demais quesitos sintáticos, já estudados na construção de algoritmos/programas, serão os mesmos adotados para subalgoritmos.

Nos itens anteriormente exibidos, há vários termos novos que serão detalhadamente estudados na sequência. Porém, ainda é necessário que você tenha uma visão macro de um algoritmo, em que há subalgoritmos definidos e declarados.

A seguir, na figura 25, está ilustrada uma forma geral, sintaxe, de representação de um algoritmo em pseudocódigo com subalgoritmos.

```
Algoritmo <nome_do_algoritmo>;  
  
  Var  
    <declaração_de_variáveis_globais>;  
  
  <definição_dos_subalgoritmos>;  
  
  Início  
    <corpo_do_algoritmo_principal>;  
  Fim.
```

Figura 25: Forma geral de representação de um algoritmo com subalgoritmos.

Ao analisar a estrutura ilustrada na figura 25, perceber-se que se trata de uma complementação em relação à forma de representar algoritmos/programas que você já aprendeu. A princípio, as diferenças estão na questão da definição de subalgoritmos e as variáveis globais.

Tomando por base a sintaxe exibida anteriormente, você deve estar percebendo que até agora quando desenvolveu os seus algoritmos e programas, as variáveis declaradas eram globais. No entanto, se há variáveis globais existem também as locais, que são declaradas no corpo do subalgoritmo. Variáveis locais são aquelas declaradas dentro de um subalgoritmo e, portanto, somente são visíveis, acessíveis, dentro do mesmo. Isto significa, que outros subalgoritmos, nem mesmo o algoritmo principal, podem utilizá-las. Para criar variáveis locais é necessário então definir o subalgoritmo, que por sua vez pode ser de dois tipos, *Funções* ou *Procedimentos* e cada um tem as suas particularidades.

SEÇÃO 2

FUNÇÕES

A função tem como característica principal retornar um único valor ao algoritmo principal ou ao subalgoritmo que a chamou. As funções são similares ao conceito da função da matemática, em que um valor é calculado a partir de outro(s) fornecido(s) à função, parâmetros. Por exemplo: seno, cosseno, tangente, raiz quadrada, etc.

Para entender o funcionamento de funções no contexto computacional, o melhor modo é exemplificando. Inicialmente, a figura 26 ilustra a forma geral, sintaxe, de como uma função pode ser definida.

EXEMPLO



```
Função <nome> (parâmetros : tipo_de_dado) : tipo_de_dado;  
  
  Var  
    <declaração_de_variáveis_locais>;  
  
  Início  
  
    <corpo_do_subalgoritmo>;  
  
  Fim.
```

Figura 26: Algoritmo – forma geral – definição de funções.

Observam-se no exemplo da figura 26, alguns detalhes fundamentais que devem ser analisados detalhadamente:

- Na linha que identifica a função,
< **Função** <nome> (parâmetros : tipo_de_dado) : tipo_de_dado; >
há referência a parâmetros e duas vezes tipo de dado.

- *Parâmetros* – são opcionais, mas geralmente as funções necessitam receber dados para serem executadas, conforme forem projetadas. Esses dados são transmitidos no momento da chamada ao subalgoritmo, que corresponde a uma comunicação bidirecional chamada de *passagem de parâmetros*.
- Parâmetros são similares a variáveis, logo estão vinculados aos tipos de dados básicos, situação que deve ser declarada na definição do subalgoritmo: inteiro, real, literal ou lógico.
- Como uma função retorna um valor ao encerrar o processamento, a ela também é necessário que seja especificado que tipo de dado retornará.

Nas figuras 27 e 28, estão ilustrados um algoritmo completo e o respectivo programa em Pascal, em que há uma função que calcula o quadrado de um número. Seguindo a metodologia de estudos proposta anteriormente, inicialmente estude cada um dos exemplos, faça uma abstração de como o programa seria executado, para daí então seguir com as leituras.



```

Algoritmo Funcao_eleva_quadrado;

Var
    num, w : real;
//-----
// Definição da função quadrado
//-----
Função quadrado(x : real) : real;
    Var
        y : real;

    Início
        y := x * x;
        retorne y;
    Fim;
//-----
// Algoritmo principal
//-----
Início
    escreva('Entre com um número: ');
    leia(num);
    w := quadrado(num);
    escreva('O quadrado de ', num, ' é = ', w);
Fim.
    
```

Figura 27: Algoritmo – função eleva ao quadrado.



```

Program Funcao_eleva_quadrado;

Var
    num, w : real; // variáveis globais

//-----
// Definição da função quadrado
//-----
Function quadrado(x : real) : real;
Var
    y : real; // variável local
Begin
    y := x * x;
    quadrado := y;
End;

//-----
// Programa principal
//-----
Begin
    write('Entre com um número: ');
    read(num);
    w := quadrado(num); // chamada a função
    quadrado
    write('O quadrado de ', num, ' é = ', w);
    readkey;
End.

```

Figura 28: Programa em Pascal – função eleva ao quadrado.

Os dois exemplos anteriores possuem características essenciais para o entendimento de como implementar funções.

- O primeiro quesito é a diferença do algoritmo relação ao Pascal, assim como em outros códigos, basicamente é apenas a tradução dos termos para o inglês.
- No algoritmo a linha com a instrução < **retorne** *y*; >, corresponde a interrupção da execução da função e, como o próprio termo expressa, retorna um valor ao trecho de algoritmo/ programa que a chamou. Em Pascal, a instrução equivalente é < quadrado := *y*; >, ou seja, a atribuição do valor de retorno ao

nome da função. Essa característica do Pascal não é universal, o padrão da maioria das linguagens, tais como Java e C, é o uso do comando *return*.

- Com relação à chamada a função, `< w := quadrado (num) ; >`, é igual em ambos, algoritmo/Pascal. O destaque nesse item é o fato de que a uma variável está sendo atribuída o nome da função.
- Ainda em relação à chamada a função, deve-se observar o uso dos parâmetros, nesse caso, a variável *num* foi utilizada para a passagem de parâmetros. O uso de parâmetros é amplo, podendo ser utilizados além de variáveis, constantes e até mesmo outras funções. Quando forem necessários mais de um parâmetro, devem ser separados por vírgulas. Com relação ao tipo de dados dos parâmetros, devem ser compatíveis com os que forem declarados na definição da função.
- Na questão das declarações de variáveis, do ponto de vista sintático, é fundamental fixar que tanto as globais como as locais, além de um local específico, conforme os exemplos necessitam da palavra-reservada **Var**.

Para exercitar a construção de funções, realize as atividades propostas a seguir.



Escreva um algoritmo e o respectivo programa em Pascal que calcule, por meio de funções, o *perímetro* e a *área* de um triângulo retângulo. A fórmula geral de cálculo da área de um triângulo é `< Área = (B * H) / 2 >`, onde B, H respectivamente correspondem à base e altura do triângulo. Para o cálculo do perímetro, devem ser somados os comprimentos dos três lados do triângulo.

.....



O universo de aplicações práticas envolvendo funções é extenso, evidentemente muitas surgem conforme o que o sistema exige para ser implementado e também de acordo com a experiência do desenvolvedor. Neste livro, os exemplos têm características didáticas, para que você possa explorar ao máximo as técnicas de como os comandos podem ser utilizados e com isso ampliar os seus conhecimentos.

O próximo passo é o tratamento de vetores utilizando funções. Nas figuras 29 e 30, estão ilustrados um algoritmo e o programa em Pascal, que tem por finalidade responder qual o maior elemento em um vetor.

Estude detalhadamente os códigos antes de prosseguir.



```

Algoritmo Funcao_leitura_maior_valor_vetor_inteiros;

Var
  Numeros: conjunto [1..1000] de inteiro;
  Ultimo: inteiro;
// -----
Função Maior_valor(x : inteiro): inteiro;
  Var
    i, maior : inteiro;

  Início
    maior := Numeros[1];

    Para i de 2 até x incr de 1 faça
      Se Numeros[i] > maior então
        maior := Numeros[i];
      Fim_se;
    Fim_para;

    retorne maior;
  Fim;
// -----
Função Leitura_Vetor(): inteiro;
  Var
    i, Ult : inteiro;

  Início
    i := 1;
  Repita
    escreva ('Entre com o ', i, 'o. número: ');
    leia(Numeros[i]);
    Se (Numeros[i] < 0) então
      Ult := i - 1;
      retorne Ult;
    senão
      i := i + 1;
      Ult := i;
    Fim_se;
  Até_que ((Numeros[i] < 0) .ou. (Ult = 1000));
  Fim;

// Algoritmo Principal-----

  Início

    Ultimo := Leitura_Vetor();
    escreva('Maior valor do vetor é ', Maior_valor(Ultimo));

  Fim.

```

Figura 29: Algoritmo – tratamento de vetores com funções.



```

Program Funcao_leitura_maior_valor_vetor_inteiros;

Var
  Numeros: array [1..1000] of integer;
  Ultimo: integer;

// -----
Function Maior_Valor(x : integer): integer;
  Var
    i, maior : integer;

  Begin
    maior := Numeros[1];

    For i := 2 to x do
      If Numeros[i] > maior then
        maior := Numeros[i];

    Maior_Valor := maior;
  End;

// -----
Function Leitura_Vetor(): integer;
  Var
    i, Ult : integer;

  Begin
    i := 1;
    Repeat
      write ('Entre com o ',i,'o. número: ');
      read(Numeros[i]);
      If (Numeros[i] < 0) then
        Begin
          Ult := i - 1;
          Leitura_Vetor := Ult;
        End
      else
        Begin
          i := i + 1;
          Ult := i;
        End;
    Until ((Numeros[i] < 0) or (Ult = 1000));
  End;

// Programa Principal-----
Begin

  Ultimo := Leitura_Vetor();
  writeln('Maior valor do vetor é ',Maior_Valor(Ultimo));

  readkey;

End.

```

Figura 30: Programa em Pascal – tratamento de vetores com funções.

Os exemplos apresentados anteriormente trazem detalhes relevantes que devem ser observados com atenção:

- Inicialmente o fato de haver duas funções com propósitos diferentes, uma com o uso de parâmetros e a outra não.
- Observa-se que a função *Leitura_Vetor*, ao encerrar a execução, retorna o índice do último elemento inserido válido. Significa que, tem a vantagem de ser usada para qualquer tamanho de vetor.
- A função *Maior_valor* lê o vetor e detecta qual o maior valor que consta no vetor e retorna esse valor. É uma função que também pode ser usada em qualquer tamanho do vetor, pois o valor do último índice válido é controlado pelo parâmetro, *x*.
- A variável indexada *Numeros* e a variável *Ultimo* são globais e são visíveis no sistema todo.
- Como variáveis locais aparecem, *i*, *maior* e *Ult*, cujos valores são acessíveis apenas dentro da função onde foram declaradas. Observa-se que a variável *i* consta nas duas funções, situação que não gera conflito, pelo fato de que é local e pode ter valores diferentes em cada função.
- Observa-se que o algoritmo/programa principal ficou reduzido em comparação a outros exemplos já estudados. Essa é uma das vantagens do uso de funções, pois modulariza-se o sistema e ainda os módulos podem ser genéricos. Nos exemplos em referência, servem para qualquer tamanho de vetor.

Antes de prosseguir com outro assunto, teste seus conhecimentos sobre funções, resolvendo a atividade a seguir.



Aprimore o programa da figura 30, de modo que tenha mais uma função para determinar qual é o menor valor e também informe quais as posições, índices, dos respectivos valores, menor e maior.

.....


SEÇÃO 3

PROCEDIMENTOS

Um procedimento é um subalgoritmo que tem como característica retornar nenhum valor ao algoritmo chamador de forma explícita, ou seja, por meio da própria identificação. O modo de retorno de valores dos procedimentos é por meio de variáveis globais.

Com relação à sintaxe são similares as funções com exceção do fato de que não é indicado um tipo, observe o exemplo na figura 31.

EXEMPLO



```

Procedimento <nome> (parâmetros : tipo_de_dado);

    Var
        <declaração_de_variáveis_locais>;

    Início

        <corpo_do_subalgoritmo>;

    Fim.
    
```

Figura 31: Algoritmo – forma geral – definição de procedimentos.

Nas figuras 32 e 33, estão ilustrados um algoritmo completo e o respectivo programa em Pascal com a Pesquisa Sequencial (figuras 7 e 8), implementada com procedimentos. Estude os códigos antes de continuar.



```

Algoritmo Pesquisa_Sequencial_Procedimentos;

Var
  Nomes : conjunto [1..1000] de literal[30];
  Chave : literal[30];
  Ultimo, Qtde_nomes : inteiro;
//-----
Procedimento Entrada_Nomes();
  Var
    i : inteiro;

  Início
    i := 1;
    Qtde_nomes := 0;

  Repita
    escreva ('Entre com o ', i, 'o. nome:');
    leia (Nomes[i]);
    Se (Nomes[i] = 'fim') então
      Ultimo := i - 1
    senão
      i := i + 1;
      Ultimo := i;
    Fim_se;
  Até_que (Nomes[i] = 'fim');
Fim;
//-----
Procedimento Pesquisa_Sequencial(nome : literal[30]);
  Var
    i : inteiro;

  Início
    i := 1;
    Enquanto (i <= Ultimo) faça
      Se (Nomes[i] = Chave) então
        Qtde_nomes := Qtde_nomes + 1;
      Fim_se;
      i := i + 1;
    Fim_enquanto;

  Se (Qtde_nomes > 0) então
    escreva ('Nome: ', Chave, ' - encontrado ',
      Qtde_nomes, ' Vez(s)');
  senão
    escreva ('Nome: ', Chave, ' - NÃO encontrado');
  Fim_se;
Fim;
//Algoritmo Principal-----
Início
  Entrada_Nomes();
  escreva ('Entre com o nome a ser pesquisado:');
  leia (Chave);
  Pesquisa_Sequencial(Chave);
Fim.

```

Figura 32: Algoritmo – Pesquisa Sequencial – Procedimentos.



```

Program Pesquisa_Sequencial_Procedimentos;

Var
  Nomes : array [1..1000] of string[30];
  Chave : string[30];
  Ultimo, Qtde_nomes : integer;
//-----
Procedure Entrada_Nomes();
  Var
    i : integer;

  Begin
    i := 1;
    Qtde_nomes := 0;

    Repeat
      write ('Entre com o ', i, 'o. nome:');
      read(Nomes[i]);
      if (Nomes[i] = 'fim') then
        Ultimo := i - 1
      else
        Begin
          i := i + 1;
          Ultimo := i;
        End;
      Until (Nomes[i] = 'fim');
    End;
//-----
Procedure Pesquisa_Sequencial(nome : string[30]);
  Var
    i : integer;

  Begin
    i := 1;
    While(i <= Ultimo) do
      Begin
        if (Nomes[i] = Chave) then
          Qtde_nomes := Qtde_nomes + 1;
          i := i + 1;
        End;

      if (Qtde_nomes > 0) then
        writeln('Nome: ', Chave, ' - encontrado ', Qtde_nomes, ' Ve(z)s')
      else
        writeln('Nome: ', Chave, ' - NÃO encontrado');
      End;
//Programa Principal-----
  Begin
    Entrada_Nomes();
    write ('Entre com o nome a ser pesquisado:');
    readln(Chave);
    Pesquisa_Sequencial(Chave);
    readkey;
  End.

```

Figura 33: Programa em Pascal – Pesquisa Sequencial – Procedimentos.

Com relação aos exemplos ilustrados, pode-se destacar alguns detalhes, tais como:

- um procedimento com parâmetros e outro sem;
- um literal como parâmetro;
- ambos os procedimentos servem para qualquer tamanho de vetor de nomes.

Agora, para testar os seus conhecimentos sobre funções, procedimentos, variáveis globais e locais resolva a atividade proposta a seguir.

Nesta etapa dos estudos é essencial uma autoavaliação. Portanto, resolva a atividade a seguir com o que você já aprendeu e, se necessário, consulte os exemplos anteriormente ilustrados. Somente após ter uma solução de sua autoria, consulte a resposta no final do livro e leia atentamente as observações.



Escreva um algoritmo e o respectivo programa em Pascal, de modo que tenha uma função para a leitura de números inteiros, um procedimento que retorne o maior e o menor valor e quantas vezes cada um desses valores se repetem no vetor.


.....

SEÇÃO 4

PASSAGEM DE PARÂMETROS

Até agora, tanto nos exemplos como nas atividades, os parâmetros foram tratados de um modo direto. Mas existem algumas particularidades importantes neste tema, inclusive pelo fato de que a passagem de parâmetros pode ser de dois modos distintos: *por valor* (cópia) ou *por referência*. Nesse contexto é necessário saber diferenciar quem são os parâmetros *formais* e os *reais*. O exemplo da figura 34 ilustra esta situação.

EXEMPLO



```

Algoritmo Funcao_eleva_quadrado;

  Var
    num, w : real;
  //-----
  // Definição da função quadrado
  //-----
  Função quadrado(x : real) : real;
    Var
      y : real;

    Início
      y := x * x;
      retorne y;
    Fim;
  //-----
  // Algoritmo principal
  //-----
  Início
    escreva('Entre com um número: ');
    leia(num);
    w := quadrado(num);
    escreva('O quadrado de ', num, ' é = ', w);
  Fim.
  
```

Figura 34: Algoritmo – função eleva ao quadrado.

Os parâmetros *reais* são aqueles que substituem os parâmetros *formais* quando da chamada a uma função ou procedimento. Observando o algoritmo da figura 34, pode-se dizer que quando a função é chamada para ser executada pelo algoritmo principal, na linha `< w := quadrado(num); >`, a variável *num*, naquele momento é o parâmetro real, que substitui o parâmetro formal *x*, variável que consta da definição da função `< Função quadrado(x : real) : real; >`.

Um bom modo de fixar essa situação é lembrar que, as variáveis que são utilizadas na definição, *formalização*, da função ou procedimento, são as *formais* e os valores que são usados para chamar e executar a função ou procedimento são os *reais*.



O uso de parâmetros é amplo, podendo ser utilizados além de variáveis, constantes e até mesmo outras funções. Quando for necessário mais de um parâmetro, deve ser separado por vírgulas. Com relação ao tipo de dados dos parâmetros, devem ser compatíveis com os que forem declarados na definição da função.

Uma vez que tenha entendido a diferença entre parâmetros formais e reais, estude, a seguir, como os mecanismos de passagem de parâmetros por valor e referência funcionam.

Passagem de parâmetros por valor

Ocorre quando uma cópia do valor do parâmetro é passado ao parâmetro formal no ato da chamada à função ou procedimento. A execução da sub-rotina prossegue normalmente e todas as modificações, que por ventura ocorram no parâmetro formal, não afetam o parâmetro real. Neste tipo de passagem de parâmetros é feita, pelo compilador, uma reserva de espaço na memória do computador para os parâmetros formais, para que neles sejam armazenadas cópias dos valores dos parâmetros reais. Lembrando que, os parâmetros reais naturalmente já possuem espaço de memória alocado, pois são variáveis, constantes ou outras funções.

Apesar de você já ter outros exemplos, a seguir nas figuras 35 e 36 serão exibidos um algoritmo e o respectivo programa em Pascal que demonstram como os valores dos parâmetros reais e formais se comportam. Implemente o código da figura 36 em Pascal e execute o programa.

EXEMPLO



```

Algoritmo Passagem_parametros_valor ;

Var
  X : inteiro;
//-----
Procedimento passa_valor(y : inteiro);
  Início
    Y := Y + 1;
    escreva('Valor de Y no procedimento: ',Y);
  Fim;
//-----
Início
  X := 1;
  escreva('Valor de X ANTES da chamada ao procedimento: ',X);
  passa_valor(X);
  escreva('Valor de X DEPOIS da chamada ao procedimento: ',X);
Fim.

```

Figura 35: Algoritmo – passagem de parâmetro por valor.

EXEMPLO



```

Program Passagem_parametros_valor ;

Var
  X : integer;
//-----
Procedure passa_valor(Y : integer);
  Begin
    Y := Y + 1;
    writeln('Valor de Y no procedimento: ',Y);
  End;
//-----
Begin
  X := 1;
  writeln('Valor de X ANTES da chamada ao procedimento: ',X);
  passa_valor(X);
  writeln('Valor de X DEPOIS da chamada ao procedimento: ',X);
  readkey;
End.

```

Figura 36: Programa em Pascal – passagem de parâmetro por valor.

Ao executar o programa da figura 36, você deve ter tido a seguinte saída:

Valor de X ANTES da chamada ao procedimento: 1

Valor de Y no procedimento: 2

Valor de X DEPOIS da chamada ao procedimento: 1

A princípio parecem óbvias as saídas, mas, mesmo assim, é necessário registrar as razões, isso vai facilitar o entendimento da passagem de parâmetros por referência.

1. Como X foi inicializado com 1 no início do programa principal, por essa razão na primeira saída, antes de ser usado como parâmetro real, o valor de X permanece 1.
2. Na segunda saída em que é exibido o valor do parâmetro formal Y, o valor é 2, pois ele recebeu 1 de X e em seguida foi incrementado em mais 1.
3. Na última linha de saída novamente o valor de X é exibido, após a execução do procedimento, permanecendo em 1.

Entendido como funciona a passagem de parâmetros por valor, agora por referência.

Passagem de parâmetros por referência

A característica principal dessa modalidade de passagem de parâmetros é que não é feita uma reserva de espaço de memória para os parâmetros formais. Quando ocorre a execução de uma função ou procedimento com parâmetros passados por referência, o espaço de memória ocupado pelos parâmetros reais é compartilhado com os parâmetros formais correspondentes. Assim, as modificações que por ventura ocorram nos parâmetros formais também afetam os parâmetros reais correspondentes e vice-versa.

Para diferenciar qual modalidade de passagem de parâmetros que está sendo utilizada em uma função ou procedimento, coloca-se a palavra-reservada *Var*, antes da declaração dos parâmetros formais, quando for por *referência* e quando for por *valor* nada é colocado. É possível inclusive,

que em uma função ou procedimento ocorram as duas modalidades de passagem de parâmetros, observe no exemplo a seguir, uma definição de procedimento em pseudocódigo com as duas modalidades.

Procedimento Exemplo (Var X,Y : inteiro : Var Z : real; J : real);

Onde:

- X e Y são parâmetros formais do tipo inteiro, passagem por *referência*;
- Z é um parâmetro formal do tipo real, passagem por *referência*;
- J é um parâmetro formal do tipo real, passagem por *valor*.

Para concretizar o entendimento sobre as duas modalidades de passagem de parâmetros, nas figuras 37 e 38, o algoritmo e o respectivo programa de uma passagem por referência, adaptados dos mesmos exemplos ilustrados nas figuras 35 e 36, passagem por valor.

Implemente o código em Pascal e execute o programa.

EXEMPLO



```

Algoritmo Passagem_parametros_referencia ;

Var
  X : integer;
  //-----
Procedimento passa_referencia(Var Y : inteiro);
  Início
    Y := Y + 1;
    escreva('Valor de Y no procedimento: ',Y);
  Fim;
  //-----
Início
  X := 1;
  escreva('Valor de X ANTES da chamada ao procedimento: ',X);
  passa_referencia(X);
  escreva('Valor de X DEPOIS da chamada ao procedimento: ',X);
Fim.
  
```

Figura 37: Algoritmo – passagem de parâmetro por referência.



```

EXEMPLO

Program Passagem_parametros_referencia ;

Var
  X : integer;
//-----
Procedure passa_referencia(Var Y : integer);
  Begin
    Y := Y + 1;
    writeln('Valor de Y no procedimento: ',Y);
  End;
//-----
Begin
  X := 1;
  writeln('Valor de X ANTES da chamada ao procedimento: ',X);
  passa_referencia(X);
  writeln('Valor de X DEPOIS da chamada ao procedimento: ',X);
  readkey;
End.

```

Figura 38: Programa em Pascal – passagem de parâmetro por referência.

Se você executou o programa da figura 38, deve ter tido como saída a seguinte situação:

```

Valor de X ANTES da chamada ao procedimento: 1
Valor de Y no procedimento: 2
Valor de X DEPOIS da chamada ao procedimento: 2

```

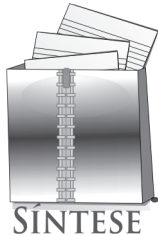
O interessante dessa saída do programa é que mesmo sem que a variável X tenha sofrido alguma alteração explícita em código, passou a ter o valor 2. O fato se resume em que, tanto X como Y, durante a execução do procedimento estavam compartilhando a mesma posição de memória e, como Y além de receber o valor inicial de X, 1, a ele foi atribuído mais 1, resultando em 2, que foi colocado no mesmo endereço de memória que ambos compartilhavam.

Pode-se afirmar que a passagem de parâmetros por referência é o mesmo que declarar um parâmetro formal para ser um endereço de memória (ponteiro) e, então, quando da execução da função ou procedimento, passar o endereço do parâmetro real.

O uso de referências na passagem de parâmetros, nitidamente traz uma economia de memória na construção de um sistema, pois são comuns situações em que os valores das variáveis têm finalidades momentâneas na execução de um sistema. Trata-se também de uma prática que pode tornar os sistemas mais rápidos na execução.

Nesta unidade, você aprendeu os detalhes essenciais para desenvolver algoritmos e programas com funções e procedimentos, o aprofundamento vai ocorrer à medida que os sistemas forem sendo desenvolvidos e as necessidades surgindo.

Na próxima unidade, a abordagem será focada em variáveis compostas heterogêneas, que além de trazer maior flexibilidade na construção de algoritmos e programas, também ampliará as possibilidades de desenvolvimento de funções e procedimentos.



Nesta unidade, você deve ter percebido o quanto evoluiu em termos de conhecimentos técnicos para o desenvolvimento de algoritmos e programas. À medida que a disciplina vai avançando, torna-se notório que é necessário modularizar os códigos para facilitar a compreensão lógica, assim como, à manutenção dos códigos.

Neste contexto, você aprendeu a criar subalgoritmos e por sua vez os respectivos programas em Pascal, o que além de estruturar melhor os algoritmos e programas, facilita a detecção de erros e a documentação de sistemas.

Nas seções dos estudos sobre funções e procedimentos além de aprender a construir essas estruturas, teve a oportunidade de implementar em Pascal e a desenvolver diversas atividades que o ajudaram a consolidar os conhecimentos.

Na questão da passagem de parâmetros por valor e referência, aprendeu como em termos de código a diferença é sutil, bastando apenas colocar a palavra-reserva *Var* antes da declaração dos parâmetros formais, que serão utilizados por referência. Por outro lado, na execução a diferença é significativa, pois as mudanças nos parâmetros formais também afetam os valores dos parâmetros reais.

Enfim, foi uma unidade de estudos que mostrou a você a flexibilidade que existe na construção de algoritmos e programas.

.....

Variáveis Compostas Heterogêneas

OBJETIVOS DE APRENDIZAGEM

- Entender como as estruturas de dados heterogêneas funcionam.
- Saber definir variáveis compostas heterogêneas.
- Aprender a manipular dados com registros.
- Aprender a criar tipos definidos pelo programador.
- Aprender a escrever algoritmos e programas com variáveis compostas heterogêneas.

ROTEIRO DE ESTUDOS

- SEÇÃO 1 – Introdução
- SEÇÃO 2 – Registros
- SEÇÃO 3 – Tipo definido pelo programador

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Com relação à representação dos dados, tanto nos exemplos já ilustrados, como nas atividades que você desenvolveu, foram utilizados os quatro tipos de dados básicos: numéricos (inteiros e reais), literais e lógicos. Porém, na prática à medida que os sistemas tornam-se mais complexos, podem ocorrer situações em que os dados básicos podem ser insuficientes. Diante deste contexto, as variáveis compostas heterogêneas e os tipos definidos pelo programador, buscam suprir a necessidade de flexibilizar a representação de dados de modo mais amplo.

SEÇÃO 1 INTRODUÇÃO

Variáveis compostas heterogêneas correspondem a um tipo de estrutura, que possui a capacidade de armazenar diversos dados de tipos básicos diferentes, denominados campos, mas que são logicamente relacionados entre si.

Um modo de entender a estruturação das variáveis compostas heterogêneas é por meio de um exemplo. Hipoteticamente, imagine uma situação em que você vai desenvolver um programa para uma universidade, em que uma parte dos dados a serem manipulados no sistema são dados cadastrais dos alunos, com as seguintes informações: RA, Nome, Rua, Número, CEP, Cidade, Estado, CPF, Idade e Curso. Tomando por base o que você aprendeu até agora, para armazenar essas informações em variáveis simples, seria necessário uma variável para cada informação. Porém na prática, geralmente trabalha-se com listas de

alunos e, com variáveis isoladas torna-se impraticável manipular tantas informações. Trabalhar com vetores ou matrizes seria uma opção, mas não tão interessante, pois como vetores e matrizes são variáveis homogêneas, de um tipo de dado básico apenas, precisaria um vetor ou matriz para cada informação especificamente.

Diante desse contexto, sem dúvida, a melhor opção é trabalhar com variáveis compostas heterogêneas do tipo *registro*, o qual permite a manipulação de um conjunto de informações de tipos primitivos diferentes, campos. Por exemplo, na figura 39 está ilustrado um formulário hipotético, de informações cadastrais de alunos, em que cada campo é composto por dados de tipos e tamanhos diferentes, mas que são inter-relacionados, pois em cada formulário preenchido os dados são específicos a uma pessoa apenas.



Registro Acadêmico:		
Nome:		
Rua	Número:	CEP:
Cidade:	Estado:	
CPF:	Idade:	
Curso:		

Figura 39: Formulário hipotético de dados cadastrais de um aluno.

Então, na próxima seção você vai aprender como os registros podem ser implementados a partir de informações reais.

SEÇÃO 2

REGÍSTROS

O registro é a principal estrutura heterogênea, sendo formado por um agrupamento de um ou mais campos (geralmente de tipos diferentes) declarado com uma única identificação, nome. Os campos podem ser formados por qualquer estrutura de dados, tais como: variáveis simples, variáveis compostas homogêneas (vetores, matrizes) e outros registros.

Em pseudocódigo a sintaxe para definir um tipo registro é a seguinte:


```
Var
  <Nome_do_registro> : Registro
                        <campo_1>:<tipo_de_dado>;
                        <campo_2>:<tipo_de_dado>;
                        : :
                        <campo_n>:<tipo_de_dado>;
Fim_registro;
```

Onde:

- <Nome_do_registro> – nome simbólico pelo qual o registro será identificado.
- <campo_n> – nome simbólico pelo qual os campos serão identificados.
- <tipo_de_dado> – inteiro, real, literal ou lógico, ou outro tipo definido pelo programador, desde que tenha sido declarado anteriormente.

Tomando por base o exemplo do formulário da figura 39, a seguir nas figuras 40 e 41, como os dados poderiam ser declarados em um registro, em algoritmo e Pascal respectivamente.

EXEMPLO




```

Var
  Cadastro_alunos : Registro
    RA : literal[10];
    Nome, Rua : literal[30];
    Numero : literal[5];
    CEP : literal[8];
    Cidade : literal[20];
    Estado : literal[2];
    CPF : literal[11];
    Idade : inteiro;
    Curso : literal[20];
  Fim_registro;
  
```

Figura 40: Algoritmo – declaração de registro – cadastro de aluno.

EXEMPLO



```

Var
  Cadastro_alunos : Record
    RA : string[10];
    Nome, Rua : string[30];
    Numero : string[5];
    CEP : string[8];
    Cidade : string[20];
    Estado : string[2];
    CPF : string[11];
    Idade : integer;
    Curso : string[20];
  End;
  
```

Figura 41: Pascal – declaração de registro – cadastro de aluno.

Observa-se nos exemplos exibidos que a transição entre pseudocódigo e Pascal é sutil, ou seja, assim como em outras instruções, basicamente é a tradução do português para o inglês. Com relação aos campos, foram usadas variáveis com os tipos básicos e o dimensionamento dos tamanhos dos campos, similarmente a dados corriqueiros de um cadastro de pessoas.


O uso de registros pode ser ampliado pelo programador, por meio da criação de tipos de dados específicos.

SEÇÃO 3

TIPO DEFINIDO PELO PROGRAMADOR

Outra forma de utilizar os registros é definir um tipo próprio, criado pelo programador. A definição, criação do tipo registro sempre deve ser feita antes da declaração das variáveis, para que o compilador possa identificar as variáveis que serão utilizadas com o novo tipo definido. Nas figuras 42 e 43, estão exemplificadas as criações do tipo registro e a declaração de variáveis do tipo criado, em algoritmo e também em Pascal, usando como referência os dados das figuras 40 e 41.

EXEMPLO



```
Tipo  
Cadastro_alunos = Registro  
    RA : literal[10];  
    Nome, Rua : literal[30];  
    Numero : literal[5];  
    CEP : literal[8];  
    Cidade : literal[20];  
    Estado : literal[2];  
    CPF : literal[11];  
    Idade : inteiro;  
    Curso : literal[20];  
    Fim_registro;  
  
Var  
Aluno_graduacao : Cadastro_alunos;  
Aluno_pos_graduacao : Cadastro_alunos;
```

Figura 42: Algoritmo – definição do tipo registro – cadastro de aluno.

EXEMPLO



```

Type
  Cadastro_alunos = Record
    RA : string[10];
    Nome, Rua : string[30];
    Numero : string[5];
    CEP : string[8];
    Cidade : string[20];
    Estado : string[2];
    CPF : string[11];
    Idade : integer;
    Curso : string[20];
  End;

Var
  Aluno_graduacao : Cadastro_alunos;
  Aluno_pos_graduacao : Cadastro_alunos;

```

Figura 43: Pascal – definição do tipo registro – cadastro de aluno.

Comparando os exemplos anteriores com os das figuras 40 e 41, nota-se que para a criação de um *tipo registro* foi apenas substituída a palavra-reservada *Var* por *Tipo* e *Type* e a notação de dois pontos, $< : >$, entre a identificação do registro e as palavras-reservadas *Registro/Record* foi substituída pelo sinal de igualdade, $< = >$, respectivamente em algoritmo e Pascal. A partir daí, criou-se um tipo de dado registro intitulado *Cadastro_alunos*, que por sua vez passou a ser uma palavra-reservada.

Observa-se também, que a partir da criação do tipo, pode-se livremente declarar as variáveis correspondentes conforme os exemplos. Neste contexto, o tipo *Cadastro_alunos* pode ser considerado como um modelo de formulário do qual foi criado. Por sua vez, as variáveis declaradas, *Aluno_graduacao* e *Aluno_pos_graduacao* são do tipo *Cadastro_alunos*, logo, possuem os mesmos campos, definidos na criação do tipo.

A partir da definição do tipo registro e declarar as respectivas variáveis é necessário saber como manipular os dados. Nas figuras 44 e 45, um algoritmo e o respectivo programa em Pascal, de rotinas para a leitura, atribuição e escrita de dados de registros.



```

Algoritmo Definicao_Manipulacao_Registros;

Tipo
  Cadastro_alunos = Registro
    RA : literal[10];
    Nome, Rua : literal[30];
    Numero : literal[5];
    CEP : literal[8];
    Cidade : literal[20];
    Estado : literal[2];
    CPF : literal[11];
    Idade : integer;
    Curso : literal[20];
    Fim_registro;

//-----
Var
  Aluno_graduacao : Cadastro_alunos;
  Aluno_pos_graduacao : Cadastro_alunos;
//-----

Função Le_formulario() : Cadastro_alunos;
Var
  Aluno : Cadastro_alunos;
Início
  escreva('Cadastro de aluno:');
  escreva('Registro Acadêmico: '); leia(Aluno.RA);
  escreva('Nome: '); leia(Aluno.Nome);
  escreva('Rua: '); leia(Aluno.Rua);
  escreva('Número: '); leia(Aluno.Numero);
  escreva('CEP: '); leia(Aluno.CEP);
  escreva('Cidade: '); leia(Aluno.Cidade);
  escreva('Estado: '); leia(Aluno.Estado);
  escreva('CPF: '); leia(Aluno.CPF);
  escreva('Idade: '); leia(Aluno.Idade);
  escreva('Curso: '); leia(Aluno.Curso);
  retorne Aluno;
Fim;

//-----
Procedimento Exibe_formulario(Aluno : Cadastro_alunos);
Início
  escreva('Dados do aluno:');
  escreva('RA: ', Aluno.RA, ' - Nome: ', Aluno.Nome);
  escreva('Rua ', Aluno.Rua, ', ', Aluno.Numero);
  escreva('Aluno.CEP, ', Aluno.Cidade, ' - ', Aluno.Estado);
  escreva('CPF: ', Aluno.CPF);
  escreva('Idade: ', Aluno.Idade);
  escreva('Curso: ', Aluno.Curso);
Fim;

//-----
Início
  Aluno_graduacao := Le_formulario();
  Exibe_formulario(Aluno_graduacao);
  escreva('Pressione qualquer tecla para continuar...');
  Aluno_pos_graduacao := Le_formulario();
  Exibe_formulario(Aluno_pos_graduacao);
  escreva('Pressione qualquer tecla para encerrar...');
Fim.

```

Figura 44: Algoritmo – Definição e manipulação de registros.



```

Program Definicao_Manipulacao_Registros;

Type
  Cadastro_alunos = Record
    RA : string[10];
    Nome, Rua : string[30];
    Numero : string[5];
    CEP : string[8];
    Cidade : string[20];
    Estado : string[2];
    CPF : string[11];
    Idade : integer;
    Curso : string[20];
  End;

//-----
Var
  Aluno_graduacao : Cadastro_alunos;
  Aluno_pos_graduacao : Cadastro_alunos;
//-----

Function Le_formulario() : Cadastro_alunos;
  Var
    Aluno : Cadastro_alunos;
  Begin
    writeln('Cadastro de aluno:');
    writeln;
    write('Registro Acadêmico: '); readln(Aluno.RA);
    write('Nome: '); readln(Aluno.Nome);
    write('Rua: '); readln(Aluno.Rua);
    write('Número: '); readln(Aluno.Numero);
    write('CEP: '); readln(Aluno.CEP);
    write('Cidade: '); readln(Aluno.Cidade);
    write('Estado: '); readln(Aluno.Estado);
    write('CPF: '); readln(Aluno.CPF);
    write('Idade: '); readln(Aluno.Idade);
    write('Curso: '); readln(Aluno.Curso);
    writeln;
    Le_formulario := Aluno;
  End;

//-----
Procedure Exibe_formulario(Aluno : Cadastro_alunos);
  Begin
    writeln('Dados do aluno:');
    writeln;
    writeln('RA: ', Aluno.RA, ' - Nome: ', Aluno.Nome);
    writeln('Rua ', Aluno.Rua, ', ', Aluno.Numero);
    writeln( Aluno.CEP, ' - ', Aluno.Cidade, ' - ', Aluno.Estado);
    writeln('CPF: ', Aluno.CPF);
    writeln('Idade: ', Aluno.Idade);
    writeln('Curso: ', Aluno.Curso);
    writeln;
  End;

//-----
Begin
  Aluno_graduacao := Le_formulario();
  Exibe_formulario(Aluno_graduacao);
  write('Pressione qualquer tecla para continuar...');
  readkey; clrscr;
  Aluno_pos_graduacao := Le_formulario();
  Exibe_formulario(Aluno_pos_graduacao);
  write('Pressione qualquer tecla para encerrar...');
  readkey;
End.

```

Figura 45: Programa em Pascal – Definição e manipulação de registros.

Os exemplos anteriormente ilustrados possuem detalhes relevantes para a compreensão do uso de algumas técnicas de programação estudadas até o momento:


- Com relação às variáveis declaradas a partir do tipo *Cadastro_alunos*, nota-se que *Aluno_graduacao* e *Aluno_pos_graduacao* são globais.
- Na função *Le_formulario*, há uma variável local *Aluno*, do tipo *Cadastro_alunos*, que tem por finalidade a leitura, atribuição de dados, via teclado de todos os campos do registro. Em seguida, o retorno de todos os dados digitados é realizado pela função. Essa técnica demonstra a vantagem no uso da função em si, pois nesse caso é genérica, serve para qualquer variável do tipo *Cadastro_alunos*.
- Outra vantagem é o uso do registro em si, pois em uma única identificação é possível manipular um conjunto heterogêneo de dados, sem a necessidade de especificar individualmente cada campo.
- No procedimento *Exibe_formulario*, o parâmetro *Aluno* é do tipo *Cadastro_alunos*. Assim como, foram destacadas as vantagens com relação ao uso da função, com o procedimento as vantagens são similares. Apenas o sentido do fluxo dos dados é do principal para o procedimento. Lembrando que o fluxo de parâmetros é bidirecional.
- Sobre a manipulação dos campos de um registro, nota-se que para acessar individualmente um campo, usa-se a notação de ponto com a seguinte sintaxe: *Variável_registro.Nome_do_campo*; – A *Variável_registro* corresponde a identificação da variável que foi declarada a partir de um tipo de dado registro. Por exemplo: *Aluno.Nome := 'Ivo Mario Mathias'*; – nesse caso *Aluno* é uma variável do tipo *Cadastro_alunos*.

As considerações realizadas anteriormente demonstraram as vantagens tanto do uso de funções e procedimentos e dos registros, recursos que tornaram os códigos mais concisos e mais legíveis. Imagine, se no caso dos exemplos das figuras 44 e 45, você simplesmente não usasse o recurso de funções e procedimentos. Quantas linhas de código teria que escrever! Quantas redundâncias de código haveria!

Outra flexibilização do uso de registros ocorre no fato de poder utilizar registros como campos de registros, o que torna alguns registros mais objetivos e também podem ser criados registros genéricos, que podem servir para diversas situações. Um caso são os endereços postais, que são comuns nos mais diversos tipos de cadastros e possuem um padrão de informações.

Na figura 46, o tipo *Cadastro_aluno*, das figuras 44 e 45, com uma alternativa de usar o endereço como um outro registro, com adaptações.

EXEMPLO



```

Tipo

    Endereco = Registro
        Logradouro : literal[30];
        Numero : literal[5];
        CEP : literal[8];
        Cidade : literal[20];
        Estado : literal[2];
        Fim_registro;

Tipo

    Cadastro_alunos = Registro
        RA : literal[10];
        Nome : literal[30];
        Ender : Endereco;
        CPF : literal[11];
        Idade : integer;
        Curso : literal[20];
        Fim_registro;

Var

    Aluno_graduacao : Cadastro_alunos;
    Aluno_pos_graduacao : Cadastro_alunos;

```

Figura 46: Algoritmo – definição do tipo registro com campos de registro.

O uso do exemplo anterior evita redundância de código, pois se o sistema tiver diversos cadastros não é necessário reproduzir em todos os cadastros os mesmos campos que compõem um endereço.

Com relação ao acesso a um campo do tipo registro, usa-se a notação de ponto, acrescentando a variável do tipo registro que foi usada para definir o campo, com o campo correspondente, por exemplo:

```
leia (Aluno_graduacao.Ender.Logradouro) ;
escreva (Aluno_graduacao.Ender.Logradouro) ;
Aluno_graduacao.Ender.Logradouro := 'Avenida Gal. Carlos Cavalcanti' ;
```

Outro modo de usar registros, além de campos de registros, é com campos compostos por outros tipos de dados definidos pelo programador. Imagine um cenário, hipotético, em que no cadastro do aluno é necessário registrar a carga horária diária semanal do aluno, conforme formulário ilustrado da figura 47.




Registro Acadêmico:						
Nome:						
Rua	Número:		CEP:			
Cidade:		Estado:				
CPF:		Idade:				
Curso:						
Carga horária diária semanal	seg	ter	qua	qui	sex	sab

Figura 47: Formulário hipotético de um aluno com carga horária semanal.

Como geralmente as soluções computacionais podem ter alternativas diferentes de códigos, neste contexto, para exemplificar as possibilidades para o formulário da figura 47, a ideia é focar em duas abordagens. A primeira seria criar um campo para cada um dos dias da semana, porém isso deixaria o registro extenso. A segunda alternativa é criar um registro genérico que represente as semanas por meio de vetores. Situação que pode ser considerada uma solução mais elegante para o problema e você aprenderá como criar um tipo definido baseado em uma variável indexada.

Na figura 48, estão trechos das definições do tipo vetor, baseados no exemplo *Cadastro_alunos*.

EXEMPLO



```


Tipo
    Dias_semana = conjunto[1..6] de inteiro ;

Tipo
    Cadastro_alunos = Registro
        RA : literal[10];
        Nome : literal[30];
        ..
        ..
        ..
        Carga_horaria : Dias_semana;
Fim_registro;
    
```

Figura 48: Algoritmo – definição do tipo vetor como campos de registro.

Basicamente, no momento da definição do tipo, é como se fosse dito ao compilador que *Dias_semana* é um tipo de dado formado por um vetor de inteiros, de dimensão 6. Como se trata de um vetor, os elementos são indexados, portanto quando da manipulação o melhor modo é por meio de uma estrutura de repetição. Na figura 49, estão ilustrados alguns exemplos de como manipular os dados, tomando por base a figura 48.

EXEMPLO



```

Para i de 1 até 6 incr de 1 faça
    leia(Aluno.Carga_horaria[i]);
    ..
    ..
    ..
Para i de 1 até 6 incr de 1 faça
    escreva(Aluno.Carga_horaria[i]);
    ..
    ..
    ..
    Aluno.Carga_horaria[4] := 68;
    
```

Figura 49: Algoritmo – manipulação do tipo vetor como campos de registro.

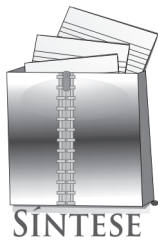
Observa-se nos exemplos da figura 49, que o tratamento de um tipo vetor definido pelo programador é similar ao de um vetor já estudado, com o diferencial que neste caso o vetor faz parte de uma estrutura do tipo registro, respeitando a sintaxe dessa estrutura.

Para finalizar esta unidade e consolidar os assuntos até agora estudados, desenvolva a atividade a seguir.



Tomando como referência os exemplos estudados, reescreva e implemente em Pascalzim o programa da figura 45, com a estrutura do formulário a seguir. Para os campos relativos a endereços e telefones, criar campos de registros separadamente, um para endereço e outro para telefones.

Registro Acadêmico:						
Nome:						
Rua	Número:	CEP:				
Cidade:	Estado:					
CPF:	Idade:					
Curso:						
Telefone residencial:			Telefone celular:			
e-mail:						
Carga horária diária semanal	seg	ter	qua	qui	sex	sab



Nesta unidade, você teve a oportunidade de aprender os detalhes teóricos, mas principalmente os práticos sobre variáveis compostas heterogêneas. Elas correspondem a um tipo de dado que possui a capacidade de armazenar diversos dados de tipos básicos diferentes, denominados campos que são logicamente relacionados entre si.

Foi visto que o registro é a principal estrutura composta heterogênea, sendo formado por um agrupamento de um ou mais campos, declarado com uma única identificação, nome. Os campos podem ser formados por qualquer estrutura de dados, tais como: variáveis simples, variáveis compostas homogêneas, variáveis compostas heterogêneas e outros tipos definidos pelo programador.

Você aprendeu que uma das formas de utilizar os registros é definir um tipo próprio, criado pelo programador. A definição, criação, do tipo registro sempre deve ser feita antes da declaração das variáveis, para que o compilador possa identificar as variáveis que serão utilizadas com o novo tipo definido.

De modo geral, você deve ter percebido que na programação de computadores existe flexibilidade e está diretamente relacionada ao conhecimento do programador em relação às técnicas de programação. Por essa razão, os exemplos ilustrados não correspondem a única solução, mas uma solução sucinta visando facilitar a compreensão do raciocínio lógico empregado.

No que tange a técnicas de programação e códigos concisos, além das variáveis compostas heterogêneas foi a utilização de funções e procedimentos, que também possibilitaram exemplos sem redundâncias de códigos e rotinas genéricas que podem ser reutilizadas para diversos propósitos.

Na próxima unidade, o assunto será Recursividade, que complementa os estudos em técnicas de programação quando da utilização de funções e procedimentos.

.....

Recursividade

OBJETIVOS DE APRENDIZAGEM

- Entender como funciona a recursividade.
- Saber identificar quando há recursão em códigos de algoritmos e programas.
- Aprender a escrever algoritmos e programas usando recursividade.

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

O termo recursividade, segundo o dicionário Aurélio, corresponde a possibilidade de aplicar uma regra repetidamente na construção de enunciado. Com relação à recursividade do ponto de vista da programação de computadores, há certa semelhança conceitual, em que uma rotina é recursiva quando ela chama a si mesma repetidamente para a resolução de um problema.

Diversos problemas que são resolvidos computacionalmente têm a propriedade de em cada instância do problema possuir instâncias menores do mesmo problema. Quando essa situação é identificada, diz-se que o problema possui uma estrutura recursiva. O modo de aplicar a recursividade é por meio de funções ou procedimentos.

A seguir estão descritos os detalhes para você entender esse mecanismo.

Siga em frente!

Partindo do princípio que um algoritmo cuja solução pode ser dividida em subproblemas mais simples e, estas soluções requerem a aplicação dele mesmo é chamado recursivo. Nesse contexto, isso significa que quando uma rotina, função ou procedimento, chama a si mesmo, de forma direta ou indireta, é recursiva.

Uma chamada recursiva direta é quando a rotina chama a si mesma e, indireta se a rotina possui uma chamada a outra rotina, que por sua vez contém uma chamada direta ou indireta à rotina que iniciou o processo de recursão. Em chamadas indiretas, nem sempre a recursão é explícita e às vezes pode ser difícil percebê-la por meio de uma simples leitura da rotina.

Conforme visto, o processo de recursão ocorre em chamadas que uma rotina faz a si mesma, porém todo programa de computador deve ser finito. Diante disto, para garantir que uma chamada recursiva não criará um *looping* infinito é necessário que ela esteja condicionada a uma expressão lógica que, em algum instante tornar-se-á falsa ou verdadeira (conforme o caso) e forçará que o processo de recursão encerre.

Quando se planeja um processo recursivo, deve-se dividir o problema do seguinte modo:

- *Solução trivial* – dada por definição, não necessita de recursão para ser obtida. Esta parte do problema é resolvida pelo conjunto de comandos da rotina.
- *Solução geral* – parte do problema que em essência é igual ao problema original, sendo, porém menor. A solução pode ser obtida por uma chamada recursiva.



Exemplo: fatorial;

- solução trivial : $0! = 1 \rightarrow$ implica na condição de parada da recursão.
- solução geral : $n! = n * (n-1)!$

Para facilitar o entendimento, na figura 50 estão ilustradas duas funções para o cálculo do fatorial de um número, uma no modo não recursivo e a outra recursiva, respectivamente.

<u>Função não Recursiva</u>	<u>Função Recursiva</u>
<pre> Função Fatorial(n : inteiro) : inteiro; Var i, fat : inteiro; Início fat := 1; Para i de 1 até n incr 1 faça fat := fat * i; Fim_para; Retorne fat; Fim. </pre>	<pre> Função Fatorial (n : inteiro) : inteiro; Início Se n = 0 então Retorne 1; senão Retorne (n * Fatorial(n-1)); Fim_se; Fim. </pre>

Figura 50: Algoritmo – Fatorial – não recursivo e recursivo.

Detalhes importantes que se deve observar em relação aos exemplos anteriores:

- Uma função recursiva, com chamada direta, pode ser identificada pela ocorrência da chamada a própria função em alguma de suas linhas. No exemplo em referência, esse detalhe pode ser observado na linha < **Retorne** (n * Fatorial(n-1)); >. Nessa linha que ocorre a chamada recursiva em < Fatorial(n-1)>.
- Geralmente o código de uma rotina recursiva é menor em relação a não recursiva.

A decisão de quando aplicar a recursão nem sempre é um procedimento simples. Enquanto alguns problemas têm solução imediata com o uso de recursão, outros são praticamente impossíveis de se resolver de forma recursiva. É preciso analisar o problema e verificar se realmente vale a pena tentar encontrar uma solução recursiva.

Quando o problema tem soluções recursivas, pode tornar o algoritmo conciso e relativamente simples, por outro lado a solução não recursiva (iterativa) geralmente é mais eficiente do ponto de vista da execução. Pois, cada chamada recursiva implica em um custo computacional, tanto de tempo de execução, quanto de espaço ocupado na memória. Cada

vez que uma rotina recursiva é chamada todas as variáveis locais são recriadas. Isso significa que tantas vezes quantas chamadas ocorrerem é o número de vezes que aquela função ou procedimento, com toda a estrutura de dados nela existente, ocuparão espaço na memória do computador.

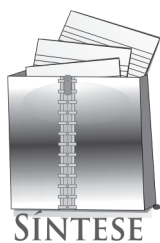
Diante do que foi exposto, pode-se generalizar que o processo recursivo é um processo de iteração, em que uma função é executada tantas vezes quanto for necessário, até que uma estrutura condicional interrompa o processo. Sendo assim, para tornar não recursiva uma rotina que é recursiva, em seu código deve estar implementada alguma estrutura de repetição.

Para testar e consolidar os seus conhecimentos sobre recursividade, elabore as atividades a seguir.



Implemente em Pascal os dois algoritmos da figura 50. Elabore dois programas separadamente, um para cada caso, função não recursiva e outro para a recursiva. O programa deve solicitar ao usuário que digite o valor do qual quer saber o fatorial.

.....



SÍNTESE

Nesta unidade, você aprendeu que um processo recursivo é um processo de iteração, em que uma função chama a si mesma e é executada tantas vezes quanto for necessário até que uma estrutura condicional interrompa o processo. Diante disso, pode-se afirmar que para tornar recursiva uma rotina que não é, em seu código deve estar implementada alguma estrutura de repetição.

Foi visto também que a decisão de quando aplicar a recursão nem sempre é um procedimento simples. Há problemas que têm solução imediata com o uso de recursão, por outro lado, há casos que praticamente é impossível a solução de forma recursiva, mesmo que seja um processo de iteração. É fundamental analisar o problema e verificar se realmente vale a pena tentar encontrar uma solução recursiva.



PALAVRAS FINAIS

Caro (a) aluno (a),

Parabéns!

Você concluiu os estudos da disciplina Algoritmos e Programação II.

Com este livro e com os demais recursos e materiais disponibilizados, você aprendeu os conceitos básicos e definições sobre Variáveis Compostas Homogêneas e também estudou de modo prático como desenvolver as aplicações mais comuns com pesquisa e classificação. Neste contexto, mais especificamente, você aprendeu como funcionam a Pesquisa Sequencial ou Linear e também a Pesquisa Binária. No que tange a classificação, os estudos foram direcionados no método da Bolha de Classificação (*Bubble Sort*).

Com relação às Matrizes, foi demonstrado que elas figuram em diversas situações de aplicações práticas, seja na própria informática, em cálculos matemáticos, editores de imagem, entre outras aplicações.

Outro tópico importante estudado foram os Subalgoritmos, em que você aprendeu que os códigos de sistemas computacionais extensos e complexos podem ser divididos em soluções menores, módulos. Além das questões práticas, você aprendeu que os Subalgoritmos podem ser definidos também como sub-rotinas, mais precisamente Funções e Procedimentos.

Na Unidade IV o foco dos estudos foram as Variáveis Compostas Heterogêneas, que correspondem a um tipo de dado que possui a capacidade de armazenar diversos dados de tipos básicos diferentes, denominados campos, que são logicamente relacionados entre si. Foi detalhado que o registro é a principal estrutura heterogênea, sendo formado por um agrupamento de um ou mais campos, declarado com uma única identificação, nome. Já os campos podem ser formados por qualquer estrutura de dados, tais como: variáveis simples, variáveis compostas homogêneas, variáveis compostas heterogêneas e outros tipos definidos pelo programador.

Na última Unidade, a abordagem foi sobre Recursividade, onde o destaque é o fato de que um processo recursivo baseia-se em um método de programação que tem como característica as rotinas que chamam a si mesmas.

Em todas as Unidades, você teve a disposição diversas atividades propostas com as respectivas soluções, que possibilitaram a você consolidar e aprimorar os assuntos abordados neste livro.

Como este livro é seu, é importante guardá-lo com cuidado. Mas consultá-lo sempre que tiver dúvidas a respeito de algoritmos e programação de computadores, pois diversas respostas podem ser encontradas aqui.

Continue estudando e sendo perseverante!

A recompensa, tanto minha como sua, será vê-lo (a) formado (a).

Ivo Mario Mathias.



REFERÊNCIAS

- Araújo, E.C. de; Hoffmann, A.B.G. **C++ BUILDER: IMPLEMENTAÇÃO DE ALGORITMOS E TÉCNICAS PARA AMBIENTES VISUAIS**. Florianópolis: Visual Books, 2006.
- Berg, A.C.; Figueiró, J.P.; **LÓGICA DE PROGRAMAÇÃO**. Canoas: Ed. ULBRA, 1998.
- Farrer, H. et al. **ALGORITMOS ESTRUTURADOS**. Rio de Janeiro: Ed. Guanabara, 1985.
- Ferreira, Aurélio B. H.; **DICIONÁRIO AURÉLIO**. Positivo Livros, 2010.
- Forbellone, A.L.V.; Eberspächer, H.F. **LÓGICA DE PROGRAMAÇÃO**. São Paulo: Makron Books, 2000.
- Guimarães, A.M.; Lages, N.A.C. **ALGORITMOS E ESTRUTURAS DE DADOS**. Rio de Janeiro: LTC, 1985.
- InfoEscola, web: www.infoescola.com, acesso em fevereiro de 2017.
- Manzano, J.A.; Oliveira, J. F. **ALGORITMOS, LÓGICA PARA DESENVOLVIMENTO DE PROGRAMAÇÃO**. São Paulo: Ed. Érica, 1996.
- Manzano, J.A; Oliveira, J. F. **ALGORITMOS: ESTUDO DIRIGIDO**. São Paulo: Ed. Érica, 1997.
- Pintacuda, N. **ALGORITMOS ELEMENTARES: PROCEDIMENTOS BÁSICOS DE PROGRAMAÇÃO**. Lisboa: Ed. Presença, 1988.
- Pascalzim, Apostila versão 6.0.3, 2016.
- Pinto, W.L. **INTRODUÇÃO AO DESENVOLVIMENTO DE ALGORITMOS E ESTRUTURAS DE DADOS**. São Paulo: Érica, 1990.
- Saliba, W.L.C. **TÉCNICAS DE PROGRAMAÇÃO: UMA ABORDAGEM ESTRUTURADA**. São Paulo: Makron-Books, 1992.
- Salveti, D.D.; Barbosa, L.M. **ALGORITMOS**, Makron-Books, São Paulo, 1998.
- Tremblay, J-P; Bunt, R.B. **CIÊNCIA DOS COMPUTADORES: UMA ABORDAGEM ALGORÍTMICA**. São Paulo: McGraw-Hill do Brasil, 1983.
- Venâncio C.F. **DESENVOLVIMENTO DE ALGORITMOS, UMA NOVA ABORDAGEM**. São Paulo: Ed. Érica, 1998.
- Wirth, N. **ALGORITMOS E ESTRUTURAS DE DADOS**. Rio de Janeiro: Prentice-Hall do Brasil, 1989.

ANEXOS

Respostas das atividades:



- Nas propostas de soluções das atividades apresentadas na sequência, principalmente nos casos de algoritmos e programas, não se tem como pretensão afirmar que são as melhores alternativas de resolução. Mas indicar uma possível alternativa de solução que contemple o assunto abordado.
 - A intenção é apresentar soluções simples e objetivas, visando códigos mais curtos para facilitar o entendimento pelo leitor.
 - Caso você sinta-se seguro, elabore soluções mais completas, principalmente nas questões que envolvem a interface com o usuário e consistências na entrada de dados.
 - Caso você tenha resolvido de modo diferente, o que é bem provável, não descarte o seu trabalho, compare com as soluções propostas, com isso aprimorará os seus conhecimentos.
-

UNIDADE I

Seção 3

```
Program Leitura_soma_vetor_reais;  
  
Var  
  VALORES : array [1..10] of real;  
  SOMA : real;  
  i      : integer;  
  
Begin  
  
  SOMA := 0;  
  
  For i:= 1 to 10 do  
    Begin  
      write ('Entre com o ',i,'o. número:');  
      read(VALORES[i]);  
      SOMA := SOMA + VALORES[I];  
    End;  
  
  writeln ('Somatório dos valores do vetor: ', SOMA);  
  
  readkey;  
End.
```

UNIDADE II

Seção 1

```
Program Pesquisa_Sequencial_nomes;

Var
  Nomes : array [1..1000] of string[30];
  Chave : string[30];
  i, Ultimo, Qtde_nomes : integer;

Begin
  i := 1;
  Qtde_nomes := 0;

  Repeat
    write ('Entre com o ', i, 'o. nome:');
    read(Nomes[i]);
    If (Nomes[i] = 'fim') then
      Ultimo := i - 1
    else
      Begin
        i := i + 1;
        Ultimo := i;
      End;

  Until (Nomes[i] = 'fim');

  write ('Entre com o nome a ser pesquisado:');
  readln(Chave);

  i := 1;

  While(i <= Ultimo) do
    Begin
      If (Nomes[i] = Chave) then
        Begin
          Qtde_nomes := Qtde_nomes + 1;
        End;

        i := i + 1;
      End;

  If (Qtde_nomes > 0) then
    writeln('Nome: ', Chave, ' - encontrado ', Qtde_nomes, ' Vez(s)')
  else
    writeln('Nome: ', Chave, ' - NÃO encontrado');

  readkey;

End.
```



```
Algoritmo Pesquisa_Sequencial_nomes;

Var
  Nomes : conjunto [1..1000] de literal[30];
  Chave : literal[30];
  i, Ultimo, Qtde_nomes : inteiro;

Inicio

  i := 1;
  Qtde_nomes := 0;

  Repita
    escreva ('Entre com o ', i, 'o. nome:');
    leia(Nomes[i]);
    Se (Nomes[i] = 'fim') então
      Ultimo := i - 1
    senão
      i := i + 1;
      Ultimo := i;
    Fim_se;
  Até que (Nomes[i] = 'fim');

  escreva ('Entre com o nome a ser pesquisado:');
  leia(Chave);

  i := 1;

  Enquanto (i <= Ultimo) faça
    Se (Nomes[i] = Chave) então
      Qtde_nomes := Qtde_nomes + 1;
    Fim_se;
    i := i + 1;
  Fim_enquanto;

  Se (Qtde_nomes > 0) então
    escreva('Nome: ', Chave, ' - encontrado ', Qtde_nomes, ' vez(es)')
  senão
    escreva('Nome: ', Chave, ' - NÃO encontrado');
  Fim_se;

Fim.
```

Observações relativas a solução apresentada:

- Ao substituir a estrutura de repetição *Para-passo/For* pela *Repita/Repeat* pode-se dizer que essa prática trouxe benefícios na construção da rotina de entrada dos nomes, a saber:
 - dispensou o uso de uma estrutura de decisão, que controla a interrupção da digitação de nomes, quando digitado *fim*;
 - diante disso, não é mais necessário o uso do comando *saia/break*, pois o *Repita/Repeat* tem o próprio controle de interrupção *Até que/Until*.

- Sobre a possibilidade de ocorrer nomes duplicados na lista e, se houver contar quantos são. Nesse quesito, cabem algumas analogias:
 - a pesquisa deve ocorrer na lista como um todo, pois os nomes podem ser encontrados em qualquer posição da lista, diferentemente do algoritmo/programa das figuras 7 e 8, em que após uma ocorrência ser localizada a pesquisa é interrompida;
 - sendo assim, dispensa o uso da variável lógica *Achou* e, em consequência, simplifica a rotina da estrutura de repetição *Enquanto/While* que localiza e conta a quantidade de nomes.

- E finalmente, a estrutura de decisão que controla as mensagens de saída, quando são ou não encontrados os nomes na lista, foi alterada em dois aspectos:
 - as expressões lógicas $\langle (i \leq \text{Ultimo}) \text{ .E. } (\text{Achou} = \text{FALSO}) \rangle$ / $\langle (i \leq \text{Ultimo}) \text{ and } (\text{Achou} = \text{FALSE}) \rangle$, foram substituídas por uma expressão simples $\langle (\text{Qtde_nomes} > 0) \rangle$ em que o gerenciamento para saber se algum nome foi encontrado é o fato da variável estar com um valor maior do que zero;
 - em consequência dessa situação, é fundamental a **variável** ser **inicializada** antes de ser utilizada.

Unidade II

Seção 2

- Atividade teste de mesa pesquisa binária:

Nomes[Medio]	Chave	Medio	Ultimo	Primeiro	Achou
	Zenildo		10	1	.F.
Lauro		5			
				6	
Xandra		8			
Zenildo		9		9	.V.

Nomes[Medio]	Chave	Medio	Ultimo	Primeiro	Achou
	Amélia		10	1	.F.
Lauro		5			
Camila		2	4		
Amélia		1	1		.V.

- Atividade pesquisa binária com números inteiros:

```

Program Pesquisa_Binaria_Numeros;

Var
  Numeros : array[1..1000] of integer;
  i, Primeiro, Medio, Ultimo, Chave : integer;
  Achou : boolean;

Begin
  i := 1;
  // Entrada dos numeros
  Repeat
    write ('Entre com o ', i, 'o. número: ');
    read(Numeros[i]);
    If (Numeros[i] < 0) then
      Ultimo := i - 1
  else
    Begin
      i := i + 1;
      Ultimo := i;
    End;
  Until (Numeros[i] < 0);

  write ('Entre com o numero a ser pesquisado:');
  readln(Chave);

  // Inicialização de variáveis
  Primeiro := 1;
  Achou := FALSE;

  // Rotina da Pesquisa binária
  while((Primeiro <= Ultimo) and (ACHOU = FALSE)) do
    Begin
      Medio := trunc((Primeiro + Ultimo)/2);
      If (Chave = Numeros[Medio]) then
        Begin
          Achou := TRUE
        End
      else
        If (Chave < Numeros[Medio]) then
          Ultimo := Medio - 1
        else
          Primeiro := Medio + 1;
        End;
    End;

  // Saída do programa
  If (Achou = TRUE) then
    writeln('Número: ', Chave, ' - encontrado ')
  else
    writeln('Número: ', Chave, ' - NÃO encontrado');

  readkey;

End.

```

Observação: a partir dessa atividade, as respostas que envolvam algoritmo e programa serão apresentadas apenas em Pascal.

Seção 3

- Atividade teste de mesa – *bubble sort*:

Numeros[i]	Numeros[i+1]	i	j	Aux	Ultimo
			10		10
223	789	1			
789	768	2			
768	789	2		789	
789	001	3			
001	789	3		789	
789	987	4			
987	345	5			
345	987	5		987	
987	006	6			
006	987	6		987	
987	026	7			
026	987	7		987	
987	121	8			
121	987	8		987	
987	003	9			
003	987	9		987	
		1	9		
		.			
		.			

- Atividade método bolha com nomes:

```

Program Metodo_Bolha_Bubble_Sort_Nomes;

Var
  Nomes: array [1..1000] of string[30];
  Aux : string[30];
  i, j, Ultimo : integer;

Begin

  // Entrada dos nomes
  i := 1;
  Repeat
    write ('Entre com o ',i,'o. nome: ');
    read(Nomes[i]);
    If ((Nomes[i] = 'fim') or (Nomes[i] = 'FIM')) then
      Ultimo := i - 1
    else
      Begin
        i := i + 1;
        Ultimo := i;
      End;
  Until ((Nomes[i] = 'fim') or (Nomes[i] = 'FIM') or (Ultimo = 1000));

  j := Ultimo;

  // Rotina do Método Bolha
  While(j > 1) do
    Begin
      For i := 1 to j-1 do
        Begin
          If Nomes[i] > Nomes[i+1] then
            Begin
              Aux := Nomes[i];
              Nomes[i] := Nomes[i+1];
              Nomes[i+1] := Aux;
            End;
          End;
        j := j - 1;
      End;

  // Saída do programa
  For i := 1 to Ultimo do
    writeln(i,'o. - ',Nomes[i]);

  readkey;

End.

```

Seção 4

- Atividade matriz de nomes:

```

Program Leitura_Escrita_Matriz_Nomes;

Var
  linha,coluna,ult_linha,ult_coluna,flag: integer;
  Matriz_Nomes: array [1..300, 1..50] of string[30];

Begin
  // Leitura dos dados da matriz
  flag := 0; // inicialização da variável que controla a interrupção dos laços
  For linha := 1 to 300 do
    Begin
      If (linha > ult_linha) then
        ult_linha := linha; // memoriza a ultima, maior linha já utilizada

      For coluna := 1 to 50 do
        Begin
          write('Entre com o nome da posição [',linha, ', ',coluna, ' ] : ');
          readln(Matriz_Nomes[linha,coluna]);
          If (coluna > ult_coluna) then
            ult_coluna := coluna; // memoriza a última, maior coluna já utilizada

          If ((Matriz_Nomes[linha,coluna] = 'fim') or
            (Matriz_Nomes[linha,coluna] = 'FIM')) then
            Begin
              flag := 1; // variável marcada para a saída do laço externo;
              break; // saída do laço do for interno colunas
            End;
          End;

          If (flag = 1) then
            break; // saída do laço do for externo linhas

        End;

      End;

  // Escrita dos dados da matriz
  writeln('Conteúdo da Matriz: ');

  For linha := 1 to ult_linha do
    Begin
      For coluna := 1 to ult_coluna do
        If ((Matriz_Nomes[linha,coluna] <> 'fim') and
          (Matriz_Nomes[linha,coluna] <> 'FIM')) then
          write(Matriz_Nomes[linha,coluna], ' - ');

        writeln;

      End;

    readkey;

  End.

```

Observações:

- uso adicional de comentários explicativos em relação a outros códigos já apresentados;
- utilização de uma variável auxiliar *flag*.

Seção 4

- Atividade matriz - troca de conteúdos entre linhas e colunas:


```
Program Tratamento_Matriz_trocas_linhas_colunas;

Var
  i,j,aux : integer;
  Matriz : array [1..5, 1..5] of integer;

Const // Declaração de constantes
  linhas = 5;
  colunas = 5;

Begin
  // Inicialização da matriz com números ímpares
  aux := 1;
  For i := 1 to linhas do
    For j := 1 to colunas do
      Begin
        Matriz[i,j] := aux;
        aux := aux + 2;
      End;

  // Exibição dos dados da matriz original
  writeln('Conteúdo da Matriz - original: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz[i,j]:5);

        writeln;
      End;

  // Troca dos elementos da 2a linha com os da 5a
  For j := 1 to colunas do
    Begin
      aux := Matriz[2,j];
      Matriz[2,j] := Matriz[5,j];
      Matriz[5,j] := aux;
    End;

  // Troca dos elementos da 3a coluna com os da 4a
  For i := 1 to colunas do
    Begin
      aux := Matriz[i,3];
      Matriz[i,3] := Matriz[i,4];
      Matriz[i,4] := aux;
    End;

  // Exibição dos dados da matriz modificada
  writeln('Conteúdo da Matriz - modificada: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz[i,j]:5);

        writeln;
      End;

  readkey;

End.
```

Observação:

- uso de uma variável auxiliar, *aux*, para inicializar a matriz e também na troca de conteúdos entre os elementos das linhas e colunas.

Seção 4

- Atividade matriz - troca de conteúdos entre diagonais:

```
Program Tratamento_Matriz_trocas_diagonais;

Var
  i,j,k,aux : integer;
  Matriz : array [1..10, 1..10] of integer;

Const // Declaração de constantes
  linhas = 10;
  colunas = 10;

Begin

  // Inicialização da matriz com números ímpares
  aux := 20;
  For i := 1 to linhas do
    For j := 1 to colunas do
      Begin
        Matriz[i,j] := aux;
        aux := aux + 2;
      End;

  // Exibição dos dados da matriz original
  writeln('Conteúdo da Matriz - original: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz[i,j]:5);
      writeln;
    End;

  // Troca dos elementos entre as diagonais principal e secundária
  j := 1;
  k := colunas;
  For i := 1 to colunas do
    Begin
      aux := Matriz[i,j];
      Matriz[i,j] := Matriz[i,k];
      Matriz[i,k] := aux;
      j := j + 1;
      k := k - 1;
    End;

  // Exibição dos dados da matriz modificada
  writeln('Conteúdo da Matriz - modificada: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz[i,j]:5);
      writeln;
    End;

  readkey;

End.
```

Seção 4

- Atividade matriz - troca de conteúdos entre linha 1 e coluna 10:

```

Program Tratamento_Matriz_trocas_linhas_colunas;

Var
  i,j,aux : integer;
  Matriz : array [1..10, 1..10] of integer;

Const // Declaração de constantes
  linhas = 10;
  colunas = 10;

Begin

  // Inicialização da matriz com números ímpares
  aux := 20;
  For i := 1 to linhas do
    For j := 1 to colunas do
      Begin
        Matriz[i,j] := aux;
        aux := aux + 2;
      End;

  // Exibição dos dados da matriz original
  writeln('Conteúdo da Matriz - original: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz[i,j]:5);
      writeln;
    End;

  // Troca dos elementos da 1a linha com os da 10a coluna
  j := linhas;
  For i := 1 to colunas do
    Begin
      aux := Matriz[1,i];
      Matriz[1,i] := Matriz[j,10];
      Matriz[j,10] := aux;
      j := j - 1;
    End;

  // Exibição dos dados da matriz modificada
  writeln('Conteúdo da Matriz - modificada: ');

  For i := 1 to linhas do
    Begin
      For j := 1 to colunas do
        write(Matriz[i,j]:5);
      writeln;
    End;

  readkey;

End.

```

UNIDADE III

Seção 2

- Atividade – cálculo de área e perímetro de triângulo retângulo com funções:

```
Program Triangulo_area_perimetro;

Var
  lado_1, lado_2, lado_3, base, altura : real;

// Definição da função area-----
Function area(b,h : real) : real;
  Begin
    area := (b*h)/2;
  End;

// Definição da função perimetro-----
Function perimetro(a,b,c : real) : real;
  Begin
    perimetro := a + b + c;
  End;

// Programa principal-----
  Begin
    writeln('Cálculo da área e o perímetro');
    writeln('de um triângulo retângulo. ');
    write('Entre com a base: ');
    read(base);
    write('Entre com a altura: ');
    read(altura);
    write('Entre com o lado 1: ');
    read(lado_1);
    write('Entre com o lado 2: ');
    read(lado_2);
    write('Entre com o lado 3: ');
    read(lado_3);
    // Saídas do Programa -----
    writeln('A área do triângulo é ', area(base,altura));
    writeln('O perímetro é ', perimetro(lado_1,lado_2,lado_3));
    readkey;
  End.
```

Observação: como destaque principal dessa atividade, uso de mais de um parâmetro em cada função.

- Atividade – determinar o menor, o maior valor em um vetor de inteiros e a posição de cada um dos valores localizados.

```

Program Funcao_leitura_menor_maior_valor_posicao_vetor_inteiros;

Var
  Numeros: array [1..1000] of integer;
  Ultimo, pos_menor, pos_maior: integer;
// -----
Function Maior_valor(x : integer): integer;
  Var
    i, maior : integer;

  Begin
    maior := Numeros[1];
    pos_maior := 1;

    For i := 2 to x do
      If Numeros[i] > maior then
        Begin
          maior := Numeros[i];
          pos_maior := i;
        End;

    Maior_valor := maior;
  End;

// -----
Function Menor_valor(x : integer): integer;
  Var
    i, menor : integer;

  Begin
    menor := Numeros[1];
    pos_menor := 1;

    For i := 2 to x do
      If Numeros[i] < menor then
        Begin
          menor := Numeros[i];
          pos_menor := i;
        End;

    Menor_valor := menor;
  End;

// -----
Function Leitura_Vetor(): integer;
  Var
    i, Ult : integer;

  Begin
    i := 1;
    Repeat
      write ('Entre com o ',i,'o. número: ');
      read(Numeros[i]);
      If (Numeros[i] < 0) then
        Begin
          Ult := i - 1;
          Leitura_Vetor := Ult;
        End
      else
        Begin
          i := i + 1;
          Ult := i;
        End;
    Until ((Numeros[i] < 0) or (Ult = 1000));
  End;

// Programa Principal-----
Begin
  Ultimo := Leitura_Vetor();
  writeln('Menor valor do vetor é ',Menor_valor(Ultimo),' posição ',pos_menor);
  writeln('Maior valor do vetor é ',Maior_valor(Ultimo),' posição ',pos_maior);

  readkey;

End.

```

Observações:

- Esse código ainda tem uma deficiência, a possibilidade de que sejam inseridos valores iguais no vetor, ou seja, podem ter mais de um valor correspondendo ao maior ou menor em posições diferentes do vetor.
- Para resolver essa situação seria necessário modificar a função de entrada de modo que cada valor inserido fosse pesquisado no vetor e se houvesse redundância, solicitasse um novo valor.

Seção 3

- Atividade – procedimento que retorna o maior e o menor valor e quantas vezes cada um desses valores se repetem no vetor:


```

Program Leitura_menor_maior_repeticoes_vetor_inteiros;

Var
  Numeros: array [1..1000] of integer;

// -----
Procedure Pesquisa_valores(x : integer);
  Var
    i, maior, menor : integer;
    rep_maior, rep_menor: integer;

  Begin
    maior := Numeros[1];
    menor := Numeros[1];
    rep_maior := 0;
    rep_menor := 0;

    For i := 2 to x do
      Begin
        If Numeros[i] > maior then
          maior := Numeros[i];
        If Numeros[i] < menor then
          menor := Numeros[i];
      End;

    For i := 1 to x do
      Begin
        If Numeros[i] = maior then
          rep_maior := rep_maior+1;
        If Numeros[i] = menor then
          rep_menor := rep_menor+1;;
      End;

    writeln('Menor valor do vetor é ',menor,' - encontrado ', rep_menor, ' Vez(s)');
    writeln('Maior valor do vetor é ',maior,' - encontrado ', rep_maior, ' Vez(s)');
  End;

// -----
Function Leitura_Vetor(): integer;
  Var
    i, Ult : integer;

  Begin
    i := 1;
    Repeat
      write ('Entre com o ',i,'o. número: ');
      read(Numeros[i]);
      If (Numeros[i] < 0) then
        Begin
          Ult := i - 1;
          Leitura_Vetor := Ult;
        End
      else
        Begin
          i := i + 1;
          Ult := i;
        End;
    Until ((Numeros[i] < 0) or (Ult = 1000));
  End;

// Programa Principal-----
Begin
  Pesquisa_valores(Leitura_Vetor);
  readkey;
End.

```

Observações:

- Apenas uma variável global - o vetor.
- A maioria das variáveis necessárias no programa estão restritas como locais à função e ao procedimento.
- O parâmetro do procedimento é a função:
 - Pesquisa_valores(Leitura_Vetor);
- Redução significativa do programa principal, apenas duas linhas.

UNIDADE IV**Seção 3**

```

Program Definicao_Manipulacao_Registros;

Type
Endereco = Record
    Rua : string[30];
    Numero : string[5];
    CEP : string[8];
    Cidade : string[20];
    Estado : string[2];
End;

Telefones = Record
    Residencial : string[15];
    Celular: string[15];
End;

Dias_semana = array[1..6] of integer ;

Cadastro_alunos = Record
    RA : string[10];
    Nome : string[30];
    Ender : Endereco;
    CPF : string[11];
    Idade : integer;
    Curso : string[20];
    Fones : Telefones;
    E_mail : string[50];
    Carga_horaria : Dias_semana;
End;

Var
Aluno_graduacao, Aluno_pos_graduacao : Cadastro_alunos;
//-----
Function Le_formulario() : Cadastro_alunos;
    Var
        Aluno : Cadastro_alunos;
        i : integer;
    Begin
        writeln('Cadastro de aluno:');
        writeln; write('Registro Acadêmico: '); readln(Aluno.RA);
        write('Nome: '); readln(Aluno.Nome);
        write('Endereço: '); readln(Aluno.Ender.Rua);
        write('Número: '); readln(Aluno.Ender.Numero);
        write('CEP: '); readln(Aluno.Ender.CEP);
        write('Cidade: '); readln(Aluno.Ender.Cidade);
        write('Estado: '); readln(Aluno.Ender.Estado);
        write('CPF: '); readln(Aluno.CPF);
        write('Idade: '); readln(Aluno.Idade);
        write('Curso: '); readln(Aluno.Curso);
        write('Telefone celular: '); readln(Aluno.Fones.Celular);
        write('Telefone residencial: '); readln(Aluno.Fones.Residencial);
        write('e-mail: '); readln(Aluno.E_mail);
        For i := 1 to 6 do
            Begin
                write('Carga horaria - dia : ',i, ' '); readln(Aluno.Carga_horaria[i]);
            End;
        writeln; Le_formulario := Aluno;
    End;
//-----
Procedure Exibe_formulario(Aluno : Cadastro_alunos);
    Var
        i : integer;
    Begin
        writeln('Dados do aluno:');
        writeln;
        writeln('RA: ', Aluno.RA, ' - Nome: ', Aluno.Nome);
        writeln('Endereço ', Aluno.Ender.Rua, ', ', Aluno.Ender.Numero);
        writeln( Aluno.Ender.CEP, ' - ', Aluno.Ender.Cidade, ' - ', Aluno.Ender.Estado);
        writeln('CPF: ', Aluno.CPF);
        writeln('Idade: ', Aluno.Idade);
        writeln('Curso: ', Aluno.Curso);
        writeln('Telefone celular: ', Aluno.Fones.Celular, ' - ', 'Telefone residencial: ',
        Aluno.Fones.Residencial);
        writeln('e-mail: ',Aluno.E_mail);
        For i := 1 to 6 do
            writeln('Carga horaria - dia : ',i, ' ',Aluno.Carga_horaria[i]);
        writeln;
    End;
//-----
Begin
Aluno_graduacao := Le_formulario();
Exibe_Formulario(Aluno_graduacao);
write('Pressione qualquer tecla para continuar...');
readkey; clrscr;
Aluno_pos_graduacao := Le_formulario();
Exibe_Formulario(Aluno_pos_graduacao);
write('Pressione qualquer tecla para encerrar...');
readkey;
End.

```

UNIDADE V

```

Program Fatorial_funcao_nao_recur_siva;

Var
    n : integer;

//-----
Function Fatorial(n:integer): integer ;
    Var
        i, fat : integer;
    Begin
        fat := 1;
        For i:=1 to n do
            fat := fat * i;

        Fatorial := fat
    End;
//-----
Begin
    write('Cálculo do fatorial, entre com o valor: ');
    readln(n);
    writeln('Fatorial de ', n, ' => ', Fatorial(n));
    readkey;
End.

```

```

Program Fatorial_funcao_recur_siva;

Var
    n : integer;

//-----
Function Fatorial(n:integer): integer ;
    Begin
        If n = 0 then
            Fatorial:= 1
        else
            Fatorial := (n * Fatorial(n-1));
    End;
//-----
Begin
    write('Cálculo do fatorial, entre com o valor: ');
    readln(n);
    writeln('Fatorial de ', n, ' => ', Fatorial(n));
    readkey;
End.

```


NOTAS SOBRE O AUTOR

IVO MARIO MATHIAS

Doutor em Agronomia pela Universidade Estadual Paulista - Júlio de Mesquita Filho - UNESP. Mestre em Informática pela Universidade Federal do Paraná. Graduado em Administração pela Universidade Estadual de Ponta Grossa. Graduado em Ciências Contábeis pela Universidade Estadual do Ponta Grossa. Professor Associado da Universidade Estadual de Ponta Grossa, desde 1987, lotado junto ao Departamento de Informática. Atuei como Professor permanente no Mestrado em Computação Aplicada. Trabalhei como pesquisador e orientador de iniciação científica em vários grupos de pesquisa, dentre eles fui Líder do grupo de pesquisa em informática aplicada a agricultura INFOAGRO-UEPG registrado junto ao CNPq. Ministrei disciplinas de Algoritmos e Programação de Computadores, Estruturas de Programação, Lógica Computacional, Fluxos em Redes, Organização de Computadores, Inteligência Artificial e Redes Neurais Aplicadas a Agricultura.