



LÓGICA DE PROGRAMAÇÃO

DESENVOLVEDOR

Sumário

Aula 1 – Lógica Básica.....	5
Lógica:.....	5
Algoritmos:	6
Português Estruturado:	6
Lógica booleana:.....	6
Tabela Verdade:	7
Exercícios:.....	9
Aula 2 – Variáveis e Constantes	12
Armazenamento de dados na memória:	12
Definição e utilização de variáveis:	12
Constantes:.....	15
Exercícios:.....	17
Aula 3 – Tipos de Dados	20
Dados Numéricos:	20
Números inteiros:.....	20
LEIA (ano)	21
ESCREVA (idade_hoje).....	22
Números reais:	22
Dados Literais:	22
Dados lógicos:.....	23
Exercícios:.....	25
Aula 4 – Tipos de Operadores	28
Operadores:.....	28
Operadores aritméticos:	28
Operadores relacionais:	28
Operadores lógicos:	28
Prioridades de Operadores:	28
Lista de Prioridades:	29
Exemplos:	29
Exercícios:.....	30
Aula 5 – Estruturas de Controle	33
Estruturas de decisão simples:.....	33
Estruturas de decisão simples encadeada:	34
Exercícios:.....	39
Aula 6 – Estruturas de Decisão Composta e Múltipla Escolha	42

Decisão Composta:.....	42
Pseudocódigo:	42
Estrutura de decisão composta encadeada:	43
Exercícios:.....	46
Aula 7 – Estruturas de Decisão de Múltipla Escolha	49
Exemplo:.....	49
Pseudocódigo	49
Decisão de múltipla escolha ou decisão encadeada?	50
Decisão encadeada:.....	52
Múltipla escolha:	52
Exercícios:.....	53
Aula 8 – Estruturas de Repetição	55
Estrutura Básica.....	56
Tipos de Estruturas de Repetição:.....	57
Estrutura de Repetição com Variável de Controle:.....	57
Pseudocódigo:	57
Exemplo:.....	59
Repita...até – Teste condicional no final:	62
Exercícios:.....	63
Aula 9 – Vetores	66
Estruturas de dados homogêneas:.....	66
Exemplo:.....	66
Vetores:	67
Declaração de um vetor:	68
Atribuição em Vetores:	68
Ordenação de um vetor:	69
Exemplo:.....	73
Exercícios:.....	74
Aula 10 – Matrizes.....	77
Como funciona:	77
Atribuição em matrizes:	79
Solução:	79
Exercícios:.....	81

LÓGICA DE PROGRAMAÇÃO

DESENVOLVEDOR

1
aula

Aula 1 – Lógica Básica

Lógica:

Nessa apostila, iremos abordar a lógica de programação. Mas o que exatamente é lógica?

Lógica é qualquer tipo de pensamento que leva a algum objetivo e, com a lógica da programação não é diferente. Basicamente, a lógica de programação é a maneira com que programamos, ou seja, a maneira como utilizamos sequências de algoritmos para atingir determinado resultados.

Para entendermos melhor o que foi dito acima, vamos observar um exemplo de português estruturado, que nada mais é que uma descrição em etapas de determinada ação. Como exemplo, vamos pegar uma ação que realizamos no nosso cotidiano, como tomar um copo de água.

Para entendermos o princípio da lógica de programação, precisamos dividir por etapas as nossas ações e descrevê-las:

1. Pegar um copo;
2. Levá-lo a uma torneira;
3. Posicionar o copo na saída de água;
4. Abrir a torneira;
5. Quando o copo estiver quase cheio, desligar a torneira;
6. Levar o copo até a boca;
7. Beber;
8. Largar o copo na pia;

Como você pode observar, em todas as nossas ações temos que sequenciar nossos pensamentos com o intuito de atingir nosso objetivo. O ato de sequenciar nossas ações para atingir determinado objetivo na computação é denominado algoritmo. A lógica de programação nos permite sequenciar nossos pensamentos de forma a criar algoritmos que sejam práticos e eficientes.

Porém, precisamos frisar que a lógica de programação não é uma coisa simples e que se aprende de uma hora para outra, mas uma vez que você adquira a mesma, ela ficará para sempre com você. Então, para aprendermos a programar temos de ler, estudar e principalmente praticar muito.

Os algoritmos que aprenderemos nesta disciplina são uma forma primitiva de programação e este conhecimento nos será muito útil quando formos programar em qualquer linguagem de programação, pois a lógica de criação de algoritmos e de qualquer linguagem de programação é igual.

Os passos que utilizamos nos algoritmos e na programação têm uma série de instruções que devem ser seguidas para termos um código correto. Estas instruções são muito mais rígidas nas linguagens de programação que nos algoritmos, pois é necessário que o computador “compreenda” os passos deste código e, para isso, precisamos ter regras mais rígidas. (No decorrer da apostila abordaremos essas instruções).

Algoritmos:

Como dito anteriormente, os algoritmos são formas de sequenciar nossos pensamentos com o objetivo de atingir determinado resultado. Esses algoritmos podem ser feitos de várias maneiras, uma delas, e a mais fácil para quem está começando é criar algoritmos em linguagem literal, ou seja, como descrevemos a ação de tomar água anteriormente.

Português Estruturado:

Escrever algoritmos em linguagem literal é muito simples, apenas devemos dizer os passos que temos que realizar para atingir o objetivo, sem qualquer preocupação com a sintaxe utilizada. Outro exemplo de algoritmo literal é o algoritmo para fritar um ovo, onde:

1. Pegar a panela;
2. Colocar óleo na panela;
3. Acender o fogo;
4. Quebrar o ovo;
5. Colocar a gema e a clara na panela;
6. Esperar ficar pronto;
7. Retirar o ovo;
8. Apagar o fogo;

No exemplo anterior temos todos os princípios básicos para um algoritmo, ou seja, o objetivo, que é ter um ovo frito e os passos para atingir esse objetivo. Observe que apenas uma instrução deste algoritmo não nos leva ao objetivo final e estas instruções devem ser realizadas seguindo uma sequência lógica. Por exemplo, não podemos colocar a gema e a clara na panela antes de quebrar o ovo.

Podemos ainda ter um algoritmo mais detalhado, observe:

1. Pegar a panela da mesa;
2. Colocar a panela em cima da primeira boca do fogão;
3. Pegar o óleo no balcão;
4. Colocar o óleo na panela;
5. Acender a primeira boca do fogão;
6. Quebrar o ovo;
7. Colocar a clara e a gema na panela;
8. Colocar a casca do ovo no lixo;
9. Esperar o ovo ficar frito;
10. Retirar o ovo frito da panela;
11. Colocar o ovo frito no prato;
12. Apagar o fogo;
13. Retirar o óleo da panela;
14. Colocar a panela na pia;

Lógica booleana:

A lógica booleana foi criada por George Boole no século XIX. Por iniciativa própria, ele passou a estudar as operações matemáticas de forma diferente, separando todos os símbolos das coisas sobre as quais eles operavam, com o intuito de criar um sistema simples e totalmente simbólico. Surge assim a lógica matemática. A lógica booleana é baseada apenas em dois valores, 0 e 1 ou Falso e Verdadeiro, respectivamente. Mas pode ser representada também por quaisquer

outros valores desde que eles possuam a mesma relação que Falso e Verdadeiro, sendo um o oposto do outro.

A lógica booleana possui 3 operadores básicos, que são:

- NOT;
- AND;
- OR;

NOT: A porta lógica NOT é também conhecida como inversor por, literalmente, inverter o número de entrada. Se o número for um, por exemplo, o valor na saída será zero, e vice-versa. Esta operação pode ser vista também como inversão ou negação, e seu símbolo é indicado por uma apóstrofe após do valor (X') ou \sim antes do valor.

AND: As operações realizadas com o operador AND resultam em um valor verdadeiro apenas quando todos os valores desta operação forem verdadeiros, ou seja, 1. Caso tivermos um valor falso, o resultado será falso. O símbolo para o operador AND é um asterisco (*).

OR: Com este operador, apenas teremos um retorno falso quando todas as condições forem falsas. A simbologia para este operador é um símbolo de mais (+).

Estes operadores só podem realizar operações com números binários e, tanto o operador AND quanto o operador OR precisam de, no mínimo, dois números binários para realizar as operações. No caso do operador NOT, as operações podem ser realizadas com apenas um número.

Existem ainda outros operadores padrões e, seus comportamentos podem ser observados nas tabelas abaixo, chamadas de tabelas-verdade:

Tabela Verdade:

Veja abaixo um exemplo de tabela da verdade com o operador NOT.

NOT	
A	$\sim A$
0	1
1	0

AND		
A	B	$A * B$
0	0	0
1	0	0
0	1	0
1	1	1

OR		
A	B	$A + B$
0	0	0
1	0	1
0	1	1
1	1	1

Estas são as tabelas da verdade para os operadores lógicos básicos, agora veremos os operadores lógicos que derivados.

Os 3 operadores lógicos básicos derivam mais 4 operadores lógicos, que são:

- NOR
- NAND
- XOR
- XNOR

Veja quais são as operações realizadas por estes operadores.

NOR: Este operador é apenas a negação do operador OR, ou seja, quando tivermos todos os valores verdadeiros na tabela OR, teremos um resultado falso com esse operador, visto que o NOT inverte o valor do mesmo.

NAND: Este operador é apenas a negação do operador AND, ou seja, uma junção do AND seguido do NOT. Basta inverter os valores obtidos na tabela verdade do operador AND, visto que o NOT apenas inverte o valor do mesmo.

XOR: Este operador é considerado como OR exclusivo, ou seja, se todos os valores de entrada forem iguais (1,1 ou 0,0), o valor de saída será 0, mas se os valores forem diferentes (1,0 ou 0,1), o valor de saída será 1.

XNOR: Este operador é a negação do operador XOR, ou seja, se os valores de saída forem iguais o valor de saída será 1, já se os valores de entrada forem diferentes o valor de saída será 0.

Observe a seguir as tabelas verdade dos operadores acima:

NAND		
A	B	A NAND B
0	0	1
1	0	1
0	1	1
1	1	0

NOR		
A	B	A NOR B
0	0	1
1	0	0
0	1	0
1	1	0

XNOR		
A	B	A XNOR B
0	0	1
1	0	0
0	1	0
1	1	1

XOR		
A	B	A XOR B
0	0	0
1	0	1
0	1	1
1	1	0

Note que todas as tabelas são muito simples, basta seguir as regras dos operadores vistos anteriormente. Nas tabelas que vimos até agora utilizamos apenas dois valores e apenas uma operação. Porém, não precisamos nos limitar à isso, e podemos criar tabelas da verdade mais complexas, com mais operações e com mais valores.

Exercícios:

1. Cite os operadores básicos da linguagem booleana:

2. Cite o que os números binários podem representar:

3. O que é um algoritmo?

4. Quais são os 4 operadores lógicos derivados dos operadores básicos?



LÓGICA DE PROGRAMAÇÃO DESENVOLVEDOR

2
aula

Aula 2 – Variáveis e Constantes

Armazenamento de dados na memória:

Na execução de um algoritmo ou de um programa, para receber dados de entrada, processar as instruções e fornecer os dados de saída, é necessário que o computador armazene esses dados e instruções em sua memória, a fim de poder acessá-los posteriormente.

Por exemplo, imagine que você e um amigo desejam calcular o algoritmo da média, “de cabeça”, sem fazer anotações. Veja os passos descritos a seguir.

1. Você diz a primeira nota.
2. Seu amigo, então, guarda esse valor na sua memória humana, enquanto aguarda a segunda nota.
3. Você diz a segunda nota.
4. Seu amigo guarda o valor da segunda nota em outro local da memória diferente do anterior, para não sobrescrever (e acabar esquecendo) a primeira nota.
5. O seu amigo recupera os valores das notas em sua memória, faz o cálculo da média e coloca o valor encontrado em um terceiro local de sua memória.
6. Você pergunta: “Qual o valor da média?”
7. O seu amigo informa o resultado obtido.

De forma similar, o computador precisa guardar os dados e instruções em endereços específicos de sua memória, durante a execução de um algoritmo, a fim de não ‘esquecer’ as informações, podendo, assim, processá-las eficazmente.

Definição e utilização de variáveis:

Agora você já sabe que os dados utilizados nos algoritmos são armazenados na memória do computador para serem posteriormente acessados.

Mas, de que forma conseguimos acessar esses dados? Para acessar os dados, precisamos descobrir em que posição na memória do computador eles estão armazenados. O conceito de variável foi criado para facilitar essa busca.

Vamos compreender de que forma isso funciona?!

Imagine um grande arquivo com várias gavetas. Para conseguirmos acessar o conteúdo dessas gavetas, é importante que o armazenamento seja feito de maneira organizada, de forma que possamos encontrar facilmente o que procuramos. Para isso, seria interessante identificar as gavetas com nomes e/ou números. Cada gaveta identificada representa, dessa forma, o endereço do conteúdo que ela armazena.

01 <u> </u> Maria	06 <u> </u> Francisco	11 <u> </u>
02 <u> </u> João	07 <u> </u>	12 <u> </u>
03 <u> </u> Sílvia	08 <u> </u> José	13 <u> </u>
04 <u> </u>	09 <u> </u>	14 <u> </u>
05 <u> </u>	10 <u> </u>	15 <u> </u>

Arquivo de gavetas representando espaços reservados na memória de um computador.

A figura está representando um arquivo com 15 gavetas que podem armazenar os documentos de diferentes pessoas. Veja que as gavetas estão numeradas e cada gaveta só armazena um documento por vez. Cada número corresponde ao ‘endereço’ da

gaveta. Algumas delas estão, também, marcadas com nomes de pessoas. Essas são as que estão reservadas.

Por exemplo, a gaveta de número 01 está reservada para armazenar um documento de Maria. Ninguém além de Maria poderá colocar seus documentos na gaveta 01. Veja que tem gavetas reservadas também para João, Sílvia, Francisco e José. E há gavetas que não foram reservadas, ou seja, estão livres para serem reservadas quando necessário (essas gavetas, porém, só podem ser utilizadas se forem previamente reservadas). Dessa forma, se perguntarmos a Maria onde ela guarda os seus documentos, ela responderá: “na gaveta 01 do arquivo”. É o ‘endereço’ dos documentos dela.

O endereço ao qual nos referimos, em algoritmos, corresponde a uma posição na memória do computador. A essa posição na memória do computador, damos o nome de variável.

Uma variável é uma posição na memória do computador, que é reservada para armazenar os dados que o algoritmo vai manipular.

Uma variável precisa ter um nome (ou identificador), um tipo de dado associado a ela (tipo da variável) e a informação que ela armazena. O identificador serve para diferenciar a variável das demais, por isso deve ser único para cada variável.



O processo de criação de uma variável é chamado de declaração da variável. As variáveis devem ser declaradas no algoritmo antes de serem utilizadas, pois a declaração das variáveis permite que seja reservado um espaço na memória para o dado que vai ser armazenado e utilizado.

Para declarar uma variável, devemos proceder da seguinte forma:

<nome da variável> : <tipo da variável>

ou:

<lista de variáveis> : <tipo da variável>

l -	nota	:	real
			
	nome da variável	:	tipo da variável

II – idade : inteiro

nome da variável : tipo da variável

III – `sexo` : caracter
 nome da variável : tipo da variável

IV– frase : literal
nome da variável : tipo da variável

V– nota1, nota2 : real
lista de variáveis : tipo da variável

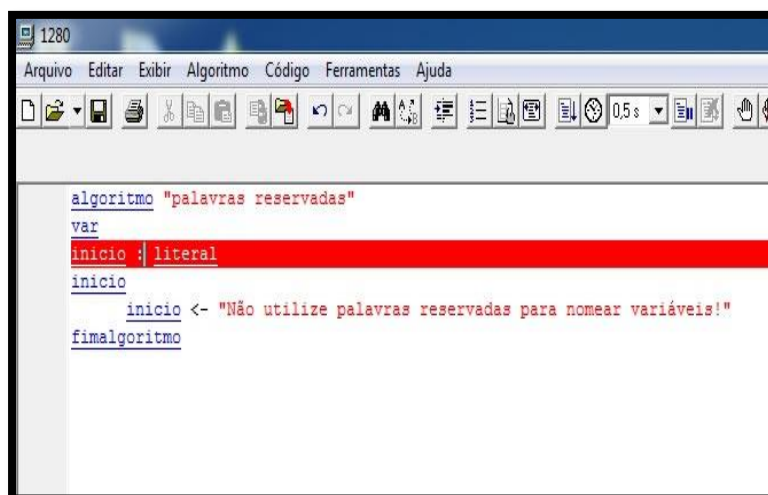
Veja, nos exemplos, que declaramos as variáveis “vazias”, isto é, sem informações associadas a elas (e, então, fornecemos essas informações posteriormente, no corpo do algoritmo). Lembre-se do exemplo das gavetas. Para reservar uma gaveta para uma pessoa, não era necessário armazenar o conteúdo, de imediato, na gaveta. Igualmente, só é possível armazenar conteúdo em uma gaveta que já foi previamente reservada. Ou seja, primeiro deve-se reservar uma gaveta e depois armazenar seu conteúdo.

Veja, também, no exemplo V, que duas ou mais variáveis de um mesmo tipo podem ser declaradas juntas, na mesma linha, separadas por vírgula (lista de variáveis).

Uma variável pode assumir diferentes valores, mas só pode armazenar um valor a cada instante. Além disso, o seu conteúdo pode mudar ao longo na execução do algoritmo.

Existem algumas regras básicas para a definição dos nomes (identificadores) das variáveis. Vamos conhecê-las.

1. Podem-se utilizar números e letras maiúsculas ou minúsculas.
2. Não se devem utilizar caracteres especiais, à exceção do caractere sublinhado, ou underline (_).
3. O primeiro caractere deve ser sempre letra ou sublinhado, mas nunca um número.
4. Não deve utilizar acentos gráficos, til ou cedilha.
5. Além dos símbolos, não é permitido espaço em branco.
6. Não se devem utilizar as palavras reservadas da linguagem



```
algoritmo "palavras reservadas"
var
inicio : literal
inicio
    inicio <- "Não utilize palavras reservadas para nomear variáveis!"
fimalgoritmo
```

Comportamento do VisuAlg durante a tentativa de nomear uma variável com uma palavra reservada.

O termo `inicio`, no VisuAlg, é utilizado para começar o algoritmo. Ao tentar identificar uma variável com o nome `inicio`, a resposta do programa é o encerramento da execução e a linha em que está o erro é realçada em vermelho. Vejamos alguns exemplos de acertos e erros na

identificação

de

variáveis:

Identificadores válidos:

- a. nome1
- b. ano_de_nascimento
- c. salario
- d. nota_aluno
- e. qtd_dias

Identificadores inválidos:

- a. 1nome – não deve começar por número
- b. Ano de nascimento – não deve ter espaço em branco
- c. @salário\$ – não deve conter caracteres especiais (contém @ e \$)
- d. Nota*aluno/01 – não deve conter caracteres especiais (contém * e /)
- e. Fim – não devemos utilizar palavras reservadas
- f. Remédio – não deve utilizar acentos gráficos

Constantes:

Além das variáveis, em algoritmos, também lidamos com constantes. Trata-se de valores fixos ou estáveis, que são escritos no programa de forma literal.

No VisuAlg, temos constantes numéricas, caracteres e lógicos.

São exemplos de constantes numéricas:

- a. 15
- b. 250
- c. 38.5

Exemplos de constantes caracteres ou literais:

- a. "F"
- b. "Maria"
- c. "Avenida São José"

Exemplos de constantes lógicos:

- a. Verdadeiro
- b. Falso

Ao atribuir valores às variáveis, podemos utilizar outras variáveis, expressões ou constantes. Veja:

var

a : inteiro

b : inteiro

c : inteiro

d : lógico

f : literal

inicio

a <- 2 //atribuição de constante

b <- a + 1 //atribuição de expressão

c <- b //atribuição de variável

d <- falso //atribuição de constante

f <- "Teste" //atribuição de constante

fimalgoritmo

Para resolver os exercícios á seguir, peça ao seu professor que lhe mostre o aplicativo VisuAlg (arquivo auxiliar).

Exercícios:

1. Porque é importante declarar uma variável antes de utilizá-la?

2. Porque o identificador de uma variável deve ser único?

3. Digite que tipo de variável é cada exemplo:

- a. 1997
- b. Eduardo
- c. 5,187
- d. -15

4. Explique com suas palavras, o que é uma constante.



LÓGICA DE PROGRAMAÇÃO DESENVOLVEDOR

3
aula

Aula 3 – Tipos de Dados

Vamos lembrar o exemplo do arquivo. Vimos que cada gaveta tem um determinado conteúdo. Para que possamos acessar esse conteúdo, a gaveta precisa estar identificada com um “endereço” representado por um nome e/ou número de identificação.

Imagine agora que, no momento da identificação, seja associado à gaveta um determinado tipo de conteúdo. A partir desse momento, é criada uma regra: essa gaveta só pode armazenar o tipo de conteúdo ao qual ela foi associada. Você viu que, no momento da criação de uma variável, é associado a ela um tipo de dado. A partir desse momento (de forma similar ao que ocorre com a gaveta do nosso exemplo), a variável só pode armazenar o tipo de dado ao qual foi associada.

Os algoritmos lidam com o conceito de tipos de dados para diferenciar dados de naturezas distintas e, assim, saber que operações podem ser realizadas com eles.

Por exemplo, não faria sentido uma operação de soma entre os dados “ana” e “5”. Assim, dependendo da natureza dos dados utilizados, algumas operações podem ou não fazer sentido. Os tipos de dados que um algoritmo pode manipular são: dados numéricos, dados literais e dados lógicos. Vamos conhecer cada tipo e compreender de que forma cada um deles pode ser utilizado.

Dados Numéricos:

No estudo da matemática, você aprendeu que existem diversos conjuntos numéricos (conjunto dos números naturais, inteiros, racionais, irracionais e reais). No estudo dos algoritmos, lidaremos apenas com os números inteiros e os números reais.

Números inteiros:

Os números inteiros são os números positivos ou negativos que não possuem parte decimal ou fracionária. Ex.: 15, -487, 0, 27835, -14937, 100. Em algoritmos, os números inteiros são utilizados geralmente para expressar valores, como quantidades ou idade.

Observe o exemplo abaixo:

Algoritmo que calcula a idade de uma pessoa, dados o ano atual e o ano em que ela nasceu.

algoritmo "calcula		idade"	
var			
ano_atual :		inteiro	
ano :		inteiro	
idade_hoje :		inteiro	
inicio			
ano_atual		<-	2013
			leia (ano)
idade_hoje	<-	ano_atual	- ano
			escreva (idade_hoje)
fimalgoritmo			

Os dados de entrada nesse algoritmo são o ano atual e o ano de nascimento digitado pelo usuário (ano). O dado de saída é a idade atual do usuário (idade_hoje). Todos os dados nesse algoritmo são numéricos inteiros. Vamos analisar o código em detalhes.

Na primeira linha do algoritmo, temos:

```
algoritmo "calcula idade"
```

Nesta primeira linha, estamos nomeando o algoritmo. Veja que o nome do algoritmo está entre aspas duplas. Na segunda linha do algoritmo, o termo *var* indica o campo onde ficarão as declarações das variáveis. Nas linhas 3, 4 e 5, as variáveis são declaradas.

Veja:

Linha		2	- var
Linha	3	- ano_atual:	inteiro
Linha	4	- ano:	inteiro
Linha 5	- idade_hoje: inteiro		

No VisuAlg, a declaração das variáveis é feita fora do bloco de execução dos algoritmos. O bloco de execução tem seu início em seguida, na linha 6, indicado pela palavra reservada *inicio*, e é finalizado com a palavra reservada *fimalgoritmo*.

Na primeira linha do bloco de execução, temos a seguinte instrução:

```
ano_atual <- 2013
```

O termo *ano_atual* é o nome que foi dado à variável. O símbolo "<-" (seta para a esquerda) é um símbolo que indica atribuição. Estamos dizendo, portanto, nessa linha de código, que está sendo atribuído o valor 2013 à variável do tipo inteiro, que foi nomeada *ano_atual*.

Na linha seguinte do código, temos:

```
LEIA (ano)
```

A instrução *leia* é uma solicitação ao usuário para que insira, no programa, alguma informação. No momento da execução dessa linha de código, no VisuAlg, uma caixa de diálogo aparece na tela, com um campo onde o usuário digita a informação pedida (caso o software esteja configurado para executar em modo DOS, aparecerá uma tela escura semelhante ao DOS e você digitará a informação nesta tela).

Quando o usuário insere a informação, o programa a recebe e guarda na variável que foi previamente reservada (*ano*). Essa variável foi criada para armazenar o dado inserido pelo usuário (o seu ano de nascimento).

Na linha seguinte, temos outra atribuição:

```
idade_hoje <- ano_atual - ano
```

Este comando indica que estamos armazenando o valor encontrado na operação "*ano_atual - ano*" (operação de subtração) no espaço da memória representado pela variável *idade_hoje*.

Nesse momento, o objetivo do algoritmo foi atingido. A última instrução exibe na tela o resultado encontrado:

ESCREVA (idade_hoje)

O comando escreva é utilizado para exibir, na tela, alguma informação. É através dele que o computador comunica-se com você, informando o resultado encontrado no algoritmo.

Números reais:

Os números reais são os números positivos ou negativos que englobam números decimais ou fracionários. Ex.: 15, -487, 1.78, 0.254, 27835, 100, 8.50. São também chamados de pontos flutuantes, nas linguagens de programação. Valores reais são aplicáveis em algoritmos que manipulam dados que expressam valores fracionários, como salário, média, preço, porcentagem, entre outros.

O algoritmo da média, que vimos nas duas primeiras aulas, utiliza dados numéricos reais.

Observe:

Algoritmo que calcula a média do aluno:

```
algoritmo "calcula                                média"
var
nota1                                           : real
nota2                                           : real
media                                           : real
inicio
                                                leia (nota1)
                                                leia (nota2)
media <- (nota1 + nota2)/2
                                                escreva (media)
finalgoritmo
```

Note que os dados de entrada (as notas do aluno) e o dado de saída (a média) são do tipo real, pois a nota de um aluno pode assumir valores fracionários, como 8.5 ou 5.2, por exemplo.

Obs.: as casas decimais, nos números reais, devem ser separadas por ponto (.) e não por vírgula (,). Por exemplo: use "8.5" e não "8,5".

Dados Literais:

Os dados literais são formados por um único caractere ou uma sequência de caracteres, que podem ser letras (maiúsculas ou minúsculas), números ou símbolos especiais (como #, \$, @, ?, &, entre outros). Os números, quando representados como caracteres, não podem ser utilizados para cálculos.

Uma sequência de caracteres pode ser chamada também de cadeia de caracteres ou string.

Exemplos de dados literais: "Fone: 3222-2222", "Av. Senador Salgado Filho, Nº 1550", "João Silveira", "M", "152", "CEP: 59052-250", "F".

Nos nossos algoritmos, representaremos todos os dados literais sempre entre aspas duplas, sejam eles caracteres isolados ou strings, pois essa é a convenção utilizada no VisuAlg.

Observe o exemplo abaixo, utilizando uma string (cadeia de caracteres).

```
algoritmo "literais"
var
nome: literal
sobrenome: literal
inicio

nome<-"Maria"
escreva ("Digite o sobrenome:")

leia (sobrenome)
escreva (nome,"",sobrenome)
finalgoritmo
```

Note que utilizaremos a notação literal quando nos referirmos a strings.

Veja outro exemplo, com um caractere isolado:

```
algoritmo "caractere"
var

sexo: caractere

inicio

sexo<-"F"
escreva (sexo)
finalgoritmo
```

Para um caractere isolado, utilizaremos a notação caracter ou caractere e não literal, como no exemplo anterior.

Execute, também, esses dois códigos no VisuAlg (salve em arquivos separados).

Dados lógicos:

Os dados lógicos são também chamados de booleanos, por ter sua origem na álgebra booleana. Os valores que esses tipos de dados podem assumir são Verdadeiro ou Falso, podendo apenas representar um desses dois valores.

Um exemplo simples:

```
algoritmo "logicos"  
var  
igualdade: logico  
inicio  
igualdade<-12=10  
escreva (igualdade)  
fim algoritmo
```

Veja que utilizamos a notação *logico* para representar o dado lógico. Sabemos que 12 não é igual a 10. Logo, a saída para esse algoritmo será “FALSO”.

Cole o código no editor de textos do VisuAlg e execute. Veja a saída, no simulador de saída. Agora, no lugar do número 12, coloque o número 10 (de forma que fique “10=10”) e execute novamente. Observe a nova saída no simulador de saída.

Exercícios:

1. Explique, com suas palavras, o processo de armazenamento de dados na memória de um computador.

2. Cite três exemplos que demonstrem o processo de declaração de uma variável:

3. Defina o conceito de constante e cite exemplos de atribuição de valores constantes às variáveis.



LÓGICA DE PROGRAMAÇÃO

DESENVOLVEDOR

4
aula

Aula 4 – Tipos de Operadores

Operadores:

Estes operadores são empregados com muita frequência em programação. É com o seu uso (muitas vezes da combinação de vários deles) é que são feitas as tarefas mais comuns de processamento de dados.

Operadores aritméticos:

Operador	Operação	Tipos dos Op.	Tipo do Resultado
+	adição	inteiro real	inteiro real
-	subtração	inteiro real	inteiro real
*	multiplicação	inteiro real	inteiro real
/	divisão	inteiro real	real real
div mod	divisão inteira, resto da divisão	inteiro inteiro	inteiro inteiro

Operadores relacionais:

Operador	Operação
<	Menor que
<=	Menor ou igual A
==	Igual A
>=	Maior ou igual A
>	Maior Que
!=	Diferente De

Operadores lógicos:

Operador	Operação	Tipo Resultante	Tipo dos Operandos
NOT	negação	lógico	lógico
AND	E (conjunção lógica)	lógico	lógico
OR	OU (disjunção lógica)	lógico	lógico
XOR	OU exclusivo	lógico	lógico

Prioridades de Operadores:

Prioridade mais alta: * / div mod

Prioridade mais baixa: + -

Obs: Parênteses ganham prioridade.

Ex: Para a expressão $2*(4+2)$ o resultado será 12 e não 10, que resultaria de:

$$2*4+2.$$

Se houver uma sequência de operadores de igual prioridade, a execução será na ordem em que aparecerem as operações.

Atenção: o resultado da expressão $8/(4*2)$ será diferente de $8/4*2$. Já no caso de $3*4/2$ e $3*(4/2)$, o resultado será exatamente o mesmo. Na dúvida, recomenda-se empregar parênteses para forçar a ordem de execução desejada.

Lista de Prioridades:

Prioridade mais alta: NOT

Prioridade média: AND

Prioridade mais baixa: OR XOR

Exemplos:

A expressão: $(A>2) \text{ AND NOT } (A>20)$ equivale a $(A>2) \text{ AND } (\text{NOT } (A>20))$.

Na expressão $(X=0) \text{ OR } (X>=2) \text{ AND } (X<=5)$, será resolvido primeiro o AND. Assim, a expressão equivale a $(X=0) \text{ OR } ((X>=2) \text{ AND } (X<=5))$.

OBS: Operações com precedência igual normalmente são executadas da esquerda para a direita, embora o compilador possa, às vezes, rearranjar os operadores para otimização de código.

Na dúvida, é aconselhável usar parênteses para garantir a prioridade desejada.

Exercícios:

1. Adição e multiplicação são dois tipos de operadores:

2. Cite 5 exemplos de operadores lógicos e descreva as suas respectivas funções:

3. Cite 3 exemplos de operadores lógicos e quais as suas funções:

4. Qual é o resultado da seguinte expressão:

$$4 + ((10 \times 6) - 8 * 4) / 2?$$

Mostre o desenvolvimento do exercício.



LÓGICA DE PROGRAMAÇÃO DESENVOLVEDOR

5
aula

Aula 5 – Estruturas de Controle

Estruturas de decisão simples:

Nesta aula, você verá que, muitas vezes, existirá a necessidade de estabelecer desvios na execução dos comandos. Esses desvios devem ocorrer quando houver uma decisão a ser tomada. Ou seja, quando houver dois caminhos possíveis, o algoritmo terá que “decidir” qual dos dois caminhos irá seguir para atingir a solução do problema. É dessa forma que introduzimos as estruturas de decisão. A estrutura de decisão pode ser simples, encadeada ou composta. Nesta aula, você conhecerá as estruturas de decisão simples e construirá algoritmos utilizando-as.

As estruturas de decisão são também denominadas estruturas de seleção ou condicionais. Nesse tipo de estrutura de controle, há uma decisão a ser tomada, sempre com base em uma condição específica, pré-estabelecida. Ou seja, de acordo com uma determinada condição, o algoritmo decide, entre dois caminhos possíveis, qual ele irá executar.

Como já dissemos na apresentação, a estrutura de decisão pode ser simples, encadeada ou composta. Vamos conhecer cada um desses tipos.

A estrutura de decisão simples executa um comando ou bloco de comandos se a condição for verdadeira. Se a condição for falsa, a estrutura é finalizada sem executar comandos. O comando básico que define a estrutura de decisão é representado pela palavra reservada SE.

Veja o algoritmo abaixo (em português estruturado), sobre o que fazer no sábado pela manhã:

```
INICIO
acordar

SE fizer sol ENTÃO

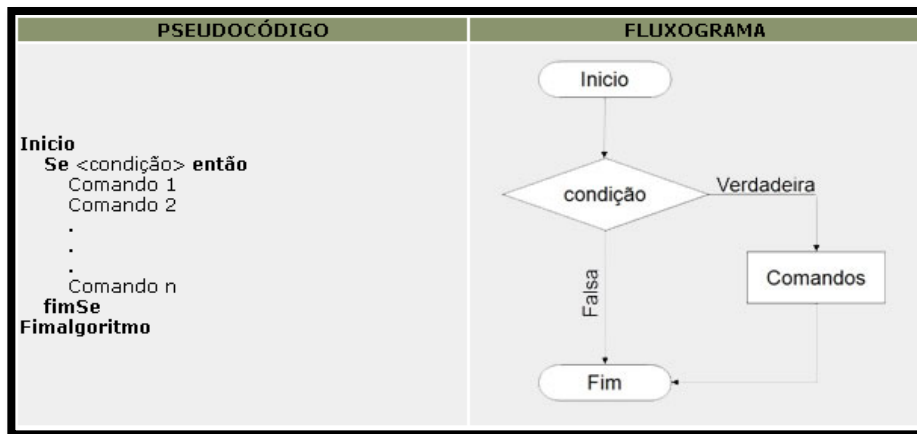
  Ir à praia

FIM SE

FIM
```

Veja que, no exemplo, tratamos de uma estrutura simples. Se a condição (fazer sol) é satisfeita, eu vou à praia. Caso contrário, não faço nada (nenhum comando é executado, caso a condição não seja atendida).

A estrutura de decisão obedece à seguinte sintaxe:



O termo condição, nessa estrutura, representa uma expressão lógica.

Você deve recordar que uma expressão lógica só pode assumir dois possíveis resultados: verdadeiro ou falso. Esse resultado, na estrutura de decisão, determina qual caminho o algoritmo vai escolher. Ou seja, dependendo do resultado da expressão lógica, o algoritmo segue para esse ou aquele caminho.

Vamos analisar o algoritmo da média mais uma vez. Só que, dessa vez, com comandos de desvios.

```
algoritmo "calcula média com desvio"
var
  nota1, nota2 : real
  media: real
início
  escreva ("Digite o valor da primeira nota: ")
  leia (nota1)
  escreva ("Digite o valor da segunda nota: ")
  leia (nota2)
  media <- (nota1 + nota2)/2
  escreval ("A média é =",media)

  se media >= 7 entao
    escreva ("Aluno aprovado!")
  fimse
fimalgoritmo
```

Da mesma forma que o algoritmo tem um início e um fim, a estrutura de decisão também deve ser devidamente inicializada e finalizada. Veja que nos exemplos citados acima, todo comando SE (IF, em linguagem C) é chamado no início e o comando FIM SE é chamado ao final da execução dessa estrutura.

Estruturas de decisão simples encadeada:

Vamos ver novamente o algoritmo sobre o que fazer no sábado pela manhã:

```
INICIO
acordar
SE fizer sol ENTÃO //inicio do primeiro SE

SE tiver dinheiro ENTÃO //inicio do segundo SE

Ir à praia

FIM SE //fim do segundo SE

FIM SE //fim do primeiro SE

FIM
```

Nota: “//” significa um comentário, muito comum para desenvolvedores se lembrarem de que se refere o código usado.

Antes, tínhamos uma condição para ir à praia no sábado pela manhã: fazer sol. Agora, além do sol, a nossa ida à praia depende de outro importante fator: ter dinheiro. Temos, então, não só uma, mas duas condições.

Da mesma forma, você verá, em alguns algoritmos, a necessidade de atender a mais de uma condição ao mesmo tempo.

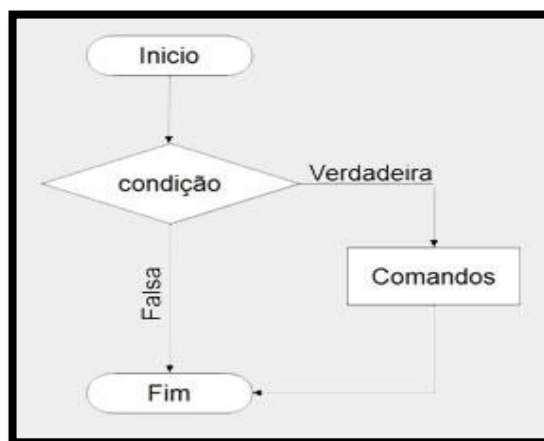
Para atender duas ou mais condições simultâneas, utilizamos a estrutura de decisão encadeada, que podemos chamar de SE's encadeados ou SE's aninhados.

Veja como fica a sintaxe de estrutura de decisão simples com seleções encadeadas:

Pseudocódigo:

Fluxograma:

```
Inicio
Se <condição> então
Comando 1
Comando 2
.
.
Comando n
fimse
finalgoritmo
```



Observe que cada SE, na estrutura encadeada, tem a sua devida finalização, ou seja, cada SE tem o seu respectivo FIM SE. É extremamente importante que você esteja atento a esse detalhe, pois o compilador sempre associará o FIM SE ao SE que estiver mais próximo. E se você se esquecer de finalizar um bloco de comandos de decisão, um erro será gerado e o seu algoritmo não será executado.

Vamos resolver o problema a seguir.

Escreva um algoritmo que solicite ao usuário que coloque os tamanhos de três lados de um triângulo e informe se os lados de fato compõem um triângulo. Lembre-se de que, em um triângulo, cada lado é menor que a soma dos outros dois lados.

```
algoritmo "triangulo"
var
lado1, lado2, lado3 : inteiro
inicio
escreva ("Digite o valor do primeiro lado: ")
leia (lado1)
escreva ("Digite o valor do segundo lado: ")
leia (lado2)
escreva ("Digite o valor do terceiro lado: ")
leia (lado3)

se (lado3<lado1+lado2) entao
se (lado2<lado1+lado3) entao
se (lado1<lado2+lado3) entao
escreva ("Os lados formam um triângulo.")
fimse
fimse
fimse
fimalgoritmo
```

A área destacada em verde, no algoritmo, mostra a estrutura de decisão encadeada. Lembre-se da finalização de cada SE.

Agora, veja bem:

INICIO	INICIO
Acordar	acordar
SE fizer sol	SE (fizer sol) E (tiver dinheiro)
SE tiver dinheiro	
Ir à praia	Ir à praia
FIM SE	FIM SE
FIM SE	FIM
FIM	

O que cada um dos algoritmos acima faz?

Observando os algoritmos acima, você pode concluir que um SE encadeado pode, algumas vezes, ser equivalente a um SE simples, com a utilização de duas expressões lógicas combinadas. Você se lembra, que os operadores lógicos E, NÃO e OU são utilizados para combinar duas ou mais expressões relacionais ou lógicas?

No algoritmo da direita, vemos uma combinação de duas expressões que podem ter por resultado um valor verdadeiro ou falso. Está fazendo sol? Além disso, tenho dinheiro? Se as duas condições forem atendidas, eu vou à praia.

Ou seja, utilizamos apenas um comando SE, porém com uma expressão combinada. Já no algoritmo da esquerda, como vimos anteriormente, ocorre o mesmo, porém, utilizamos dois comandos SE separadamente.

Veja que o código exibido na área destacada, no algoritmo do triângulo, que resolvemos há pouco, poderia ser substituído, sem alterações no resultado final, pelo seguinte código:

```
se (lado3<lado1+lado2) e (lado2<lado1+lado3) e (lado1<lado2+lado3) entao  
  escreva ("Os lados formam um triângulo.")  
fimse
```

Na programação, isso é comumente chamado de "otimização de código".

Existem, porém, casos de estruturas encadeadas que NÃO PODEM ser substituídas por combinações de expressões lógicas. Isso ocorre quando, após o teste da primeira condição, há algum comando ou bloco de comandos, que deve ser executado antes do teste da condição seguinte. Por exemplo:

```
SE fizer sol ENTÃO  
  
  Eu vou à praia  
  
SE tiver muita gente lá ENTÃO  
  
  Eu vou ao clube  
  
FIMSE  
  
FIMSE
```

No exemplo, não poderíamos utilizar a expressão: "Se fizer sol E Se tiver muita gente lá", pois só vou até a praia se fizer sol. Mas, só posso saber se tem muita gente lá, depois que eu tiver ido. Ou seja, só posso testar a segunda condição após a execução do comando que depende da primeira.

Vejamos outro exemplo:

Faça um algoritmo para calcular o dobro de um número inteiro, caso seja par e, caso o dobro seja menor do que 10, escrever seu quadrado.

```
algoritmo "quadrado do dobro"
var
num, dobro, quadrado : inteiro
inicio
escreval ("Digite um número:")
leia (num)
se (num % 2 = 0) entao
dobro <- num * 2
escreval ("O dobro do número digitado é: ",dobro)
se (dobro < 10) entao
quadrado <- dobro * dobro
escreval ("O quadrado do dobro é: ",quadrado)
fimse
fimse
fimalgoritmo
```

Veja que a segunda condição ($\text{dobro} < 10$) só pode ser testada após a execução do comando que determina esse dobro. Este comando depende da primeira condição (o número tem que ser par). Assim, essa estrutura encadeada também não pode ser substituída por uma decisão simples com expressões lógicas combinadas.

O comando escreval, faz com que o VisuAlg quebre as linhas automaticamente.

Para reforçar uma tabela com a descrição de tipos de variáveis.

Tipo	Descrição
Inteiro	Representa valores inteiros. Ex: 10, 5, -5, -10
Real ou Numérico	Representa valores reais. Ex: 10, 15.5, -14.67
Literal ou Caractere	Representa texto. Ex: "Esta é uma cadeia de caracteres", "B", "1234"
Lógico	Representa valores lógicos (VERDADEIRO ou FALSO).

Exercícios:

1. O que caracteriza as estruturas de decisão simples?

2. Qual a sintaxe dessa estrutura?

3. Qual a sintaxe da estrutura de decisão encadeada?



LÓGICA DE PROGRAMAÇÃO

DESENVOLVEDOR

6
aula

Aula 6 – Estruturas de Decisão Composta e Múltipla Escolha

Decisão Composta:

Agora que você já sabe que a estrutura de decisão simples executa um comando ou bloco de comandos se uma determinada condição for atendida. Se a condição não for atendida, a estrutura é finalizada sem executar comandos.

Então, a estrutura composta segue do mesmo princípio, com diferença que, quando a condição não é feita, há um desvio para outro comando ou até bloco de comandos.

A estrutura de decisão composta executa um comando ou bloco de comandos quando a condição é feita e outro comando diferente quando a condição proposta não é feita.

Agora, você deve começar a utilizar além do SE, a palavra reservada SENÃO. (no VisuAlg usa-se “senao”)

Vamos ver um exemplo simples para mostrar o uso do SENÃO.

```
Inicio
Acordar
SE fizer sol ENTÃO
  Ir à praia
SENÃO
  Ler um livro
FIM SE
FIM
```

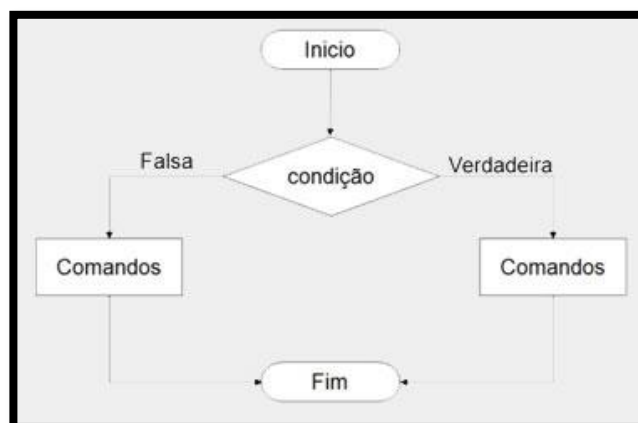
Note que desta vez estamos tratando com uma estrutura composta. Se a condição (fazer sol) é verdadeira eu vou à praia. Caso contrário, (falsa) faço outra coisa: leio um livro. Ou seja, a condição não sendo verdadeira, logo, eu executo outra atividade.

A estrutura de decisão composta obedece à seguinte sintaxe:

Pseudocódigo:

```
inicio
se <condição> então
  comando 1
  comando 2
  .
  .
  comando n
senao
  comando 1
  comando 2
  .
  .
  comando n
fimse
fim
```

Fluxograma:



Veja que, da mesma forma que na estrutura simples, o algoritmo executa um comando ou bloco de comandos, caso a situação seja verdadeira.

Faremos um algoritmo para calcular a média aritmética de um aluno e dizer se o mesmo foi aprovado ou não.

```
algoritmo "calcula média com desvio"
var
  nota1, nota2, media: real
inicio
  escreval ("Digite o valor da primeira nota:")
  leia (nota1)
  escreval ("Digite o valor da segunda nota:")
  leia (nota2)
  media <- (nota1 + nota2) % 2
  escreval ("A média é =", media)
  se (media >= 7) entao
    escreval ("Aluno aprovado!") //instrução com condição verdadeira
  senao
    escreval ("Aluno reprovado!") //instrução com condição falsa.
fimse
fimalgoritmo
```

Contudo, de forma diferente da estrutura simples, a estrutura composta executa outro comando ou bloco de comandos, quando a condição seja falsa.

Agora, o algoritmo pode executar uma instrução quando a condição (nota >= 7) for verdadeira e outra instrução quando a condição for falsa.

Estrutura de decisão composta encadeada:

Da mesma forma que na estrutura simples, na estrutura composta também podemos utilizar "SE" encadeados. Você sabe que isso ocorre quando há várias condições a serem testadas.

Agora, observe as duas situações abaixo:

Situação 01:

Início

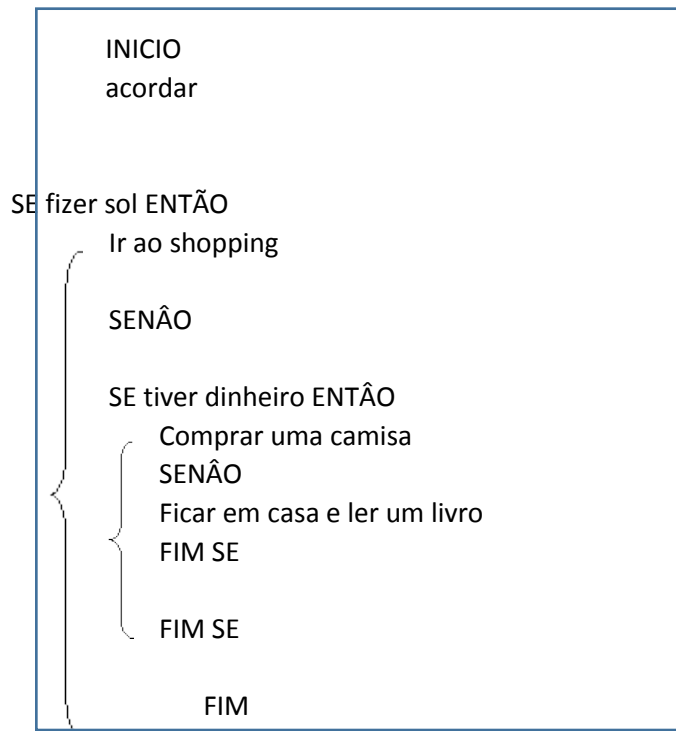
Acordar

{ SE fizer sol ENTÃO
 Ir ao shopping
 SENÃO

{ SE estiver nublado ENTÃO
 Assistir um filme
 SENÃO

{ Está chovendo.
 Ficar em casa e ler um livro
 FIM SE
 FIM SE
 FIM

Situação 02:



É fácil perceber que, em ambas as situações há uma utilização de estruturas encadeadas. Mas o que difere a Situação 01 da Situação 02?

Na Situação 01, temos várias estruturas separadas, com condições que são mutuamente exclusivas, ou seja, se uma das condições for verdadeira, todas as outras são falsas. Se não fizer sol, nem estiver nublado, com certeza estará chovendo. Há uma ação diferente para cada condição possível: ir ao shopping, assistir um filme, ler um livro.

Na Situação 02, há uma estrutura composta inserida em outra estrutura composta. Se fizer sol, assistirei a um filme. Se não fizer sol, não assistirei a um filme, mas me restam duas opções possíveis: comprar uma camisa se tiver dinheiro, ou ficar em casa e ler um livro, caso não tenha dinheiro.

Para ilustrar melhor, vamos incrementar um pouco mais o algoritmo da média:

algoritmo “calcula média com desvio composto encadeado”

```
var
nota1, nota2, media: real
inicio
escreva (“Digite o valor da primeira nota: ”)
leia (nota1)
escreva (“Digite o valor da segunda nota: ”)
leia (nota2)
media <- (nota1 + nota2)/2
escreval (“A média é =”,media)
se media >= 7 entao
escreval (“Aluno aprovado! Parabéns!”)
```



```
senao
```

```
se (media<7) e (media>=4) entao  
escreval ("Aluno em recuperação! Estude"),
```

```
senao  
escreval ("Aluno reprovado!")
```

```
fimse  
fimse  
finalgoritmo
```

Veja que, no exemplo, temos a situação de exclusão mútua. Se a média não for maior que 7.0, nem está entre 4.0 e 7.0, só pode ser menor que 4.0.

Nesse caso, ou o aluno é aprovado, ou fica em recuperação, ou é reprovado.

Exercícios:

1. O que diferencia uma estrutura de decisão composta da estrutura simples?

2. Qual a sintaxe de estrutura de decisão composta?

3. Quais as possíveis sintaxes de estrutura de decisão composta encadeada?

4. A partir do que você responder no exercício 3, dê pelo menos um exemplo.



LÓGICA DE PROGRAMAÇÃO

DESENVOLVEDOR

7
aula

Aula 7 – Estruturas de Decisão de Múltipla Escolha

Nas estruturas de decisão vimos que nas aulas anteriores, um comando ou bloco de comandos é executado de acordo com uma ou mais condições. Viu que, quando a condição não é verdadeira, a estrutura pode encerrar (estrutura de decisão simples) ou seguir outro caminho (estrutura de decisão composta).

Existem problemas, porém, em que uma expressão (ou uma variável) pode assumir diversos valores e que, cada valor assumido, comandos diferentes são executados. Neste caso, os valores são mutuamente exclusivos. Nestes casos utilizaremos as estruturas de múltipla escolha.

Imagine que você quer decidir o que fazer no seu dia de folga:

Exemplo:

VAR

Opção: Inteiro

INICIO

escreval ("Digite "1", para ir ao parque")

escreval ("Digite "2", para andar de bicicleta")

escreval ("Digite "3", para comer pizza")

leia opção

ESCOLHA opção

CASO 1

escreva ("Sair de casa às 10 horas da manhã.")

CASO 2

escreva ("Sair de casa às 4 horas da tarde.")

CASO 3

escreva ("Sair de casa às 8 horas da noite.")

OUTROCASO

escreva ("Já que não escolheu nenhuma das opções, fique em casa assistindo televisão.")

FIM ESCOLHA

FIM

Para cada, há um comando diferente (neste caso, um horário diferente para sair de casa). Isso mostra que os comandos são como dissemos a pouco, mutuamente exclusivos.

Veja que, caso não escolha nenhuma das opções anteriores, há um comando definido por padrão (que, neste caso seria assistir televisão).

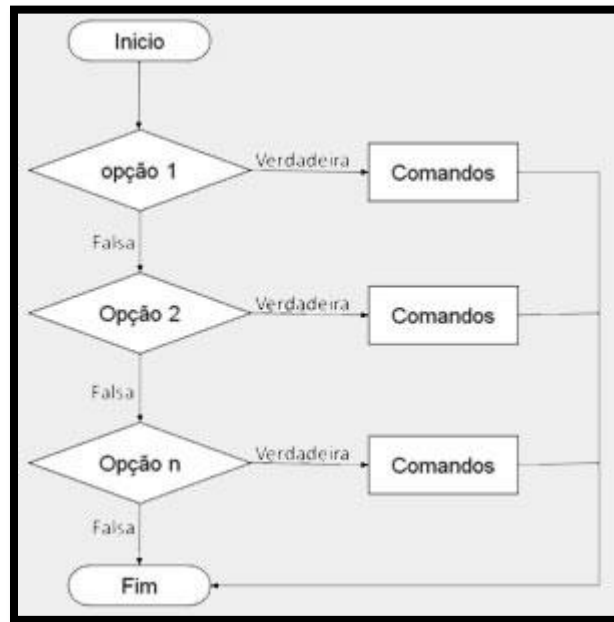
A sintaxe da estrutura de múltipla escolha é a seguinte:

Pseudocódigo

Fluxograma

Início

```
escolha <opção>  
caso <opção 1>  
  <comandos1>  
caso <opção 2>  
  <comandos2>  
caso <opção n>  
  <comandos>  
outrocaso  
  <comandos_padrão>  
fimescolha  
Fim
```



A palavra reservada outrocaso pode ser entendida como um desvio, em caso do usuário escolher uma opção não esteja entre as opções existentes. Sua utilização é opcional. Execute o algoritmo no VisuAlg.

Para uma melhor fixação, evite copiar e colar o texto da aula. Prefira reescrever o algoritmo no VisuAlg. Dessa forma, você ficará mais familiarizado com a sintaxe e, logo, sua habilidade em utilizar a estrutura será melhorada.

Decisão de múltipla escolha ou decisão encadeada?

Se você compreendeu tudo o que vimos até agora sobre as estruturas de decisão, pode estar se perguntando: “Mas, essa estrutura de múltipla escolha não é a mesma coisa que uma estrutura de decisão composta encadeada?”.

De fato, elas podem ser similares. Os dois algoritmos abaixo fazem a mesma coisa:

Decisão Encadeada:

algoritmo "Decisão encadeada"

var

opcao: inteiro

inicio

escreval ("O que fazer no final de semana:")

escreval ("Opções: 1 - Parque, 2 - Pizza, 3 - Churrasco, Outro Número - Descansar")

leia (opcao)

se opcao = 1 entao

escreval ("No final de semana iremos ao parque!")

senao

se opcao = 2 entao

escreval ("No final de semana comeremos pizza!")

senao

se opcao=3 entao

escreval ("No final de semana iremos a churrascaria!")

senao

escreval ("No final de semana iremos descansar.")

fimse

fimse

fimse

finalgoritmo

Decisão de Múltipla Escolha:

algoritmo "Múltipla Escolha"

var

opcao: inteiro

inicio

escreval ("O que fazer no final de semana:")

escreval ("Opções: 1 - Parque, 2 - Pizza, 3 - Churrasco, Outro Número - Descansar")

leia (opcao)

escolha opcao

caso 1

escreval ("No final de semana iremos ao parque!")

caso 2

escreval ("No final de semana comeremos pizza!")

caso 3

escreval ("No final de semana iremos a churrascaria!")

outro caso

escreval ("No final de semana iremos descansar.")

fimescolha

fimalgoritmo

Execute os dois algoritmos no VisuAlg e veja que, em ambos os casos, a saída é a mesma.

Cada palavra reservada que utilizamos em um algoritmo representa um comando que o programa vai executar, desde a inicialização do algoritmo, passando pelos comandos básicos (as instruções primitivas) e os comandos de cada estrutura de controle, até a finalização do algoritmo. Tudo isso são instruções que passamos para que o computador execute.

Sabendo disso, podemos ver que o algoritmo que utiliza a decisão encadeada tem um número maior de instruções. Isso demanda uma quantidade maior de memória e processamento do computador para executar o algoritmo, o que afeta diretamente o desempenho.

Seria como se você precisasse caminhar de um ponto a outro, tendo duas opções de caminho: em uma, você segue em linha reta e, na outra, você vai à ziguezague. Em qual dessas opções você acha que chegaria mais rápido? Em qual chegaria menos cansado?

Em um algoritmo simples, provavelmente, você não perceberá a diferença. Mas, em programas maiores e mais complexos, fica evidente a queda no desempenho. Por isso, é mais conveniente, nesses casos (de seleção mutuamente exclusiva), utilizar a estrutura de múltipla escolha.

Há, também, outra diferença básica entre as duas estruturas, que pode determinar quando utilizar uma ou outra:

Decisão encadeada:

Pode testar mais de um valor ao mesmo tempo. Ou seja, podem ser utilizadas expressões relacionais.

Múltipla escolha:

Só pode testar igualdade e só testa um valor por vez, por isso, não pode utilizar expressões lógicas ou relacionais.

Exercícios:

1. Qual a sintaxe da estrutura de decisão múltipla?

2. Escreva o algoritmo do dia de folga no VisuAlg e execute. Lembre-se de usar a sintaxe de estrutura múltipla.



LÓGICA DE PROGRAMAÇÃO

DESENVOLVEDOR

8
aula

Aula 8 – Estruturas de Repetição

Estrutura Básica

Você já sabe que alguns algoritmos precisam executar instruções com base em uma condição pré-estabelecida. Nas estruturas de decisão que vimos nas aulas anteriores, se uma condição é verdadeira, um comando ou grupo de comandos é executado. Depois disso, a estrutura é finalizada. Ou há o caminho alternativo, caso a condição não seja atendida.

Nas estruturas de repetição, condições também são testadas. Mas, o que, de fato, caracteriza essa estrutura é que um conjunto de ações é executado de forma repetida enquanto a condição permanece válida ou não.

Por exemplo, se alguém lhe solicita que faça um algoritmo que escreva cinco vezes a frase: “Olá, bom dia!”, de que forma você faria?

Com o que aprendemos até aqui, seria da seguinte forma:

```
algoritmo “repete frase”  
inicio  
  escreval (“Olá, bom dia!”)  
  escreval (“Olá, bom dia!”)  
  escreval (“Olá, bom dia!”)  
  escreval (“Olá, bom dia!”)  
  escreval (“Olá, bom dia!”)  
finalgoritmo
```

Simples, não? Mas e se mudássemos o número de repetições? Se, ao invés de cinco vezes tivéssemos que escrever essa mesma frase cem vezes? Ou mil vezes?

Seria extremamente cansativo para você ficar digitando tantas vezes o mesmo comando. Para o computador também não seria interessante. Já que quando o mesmo comando é repetido, há um aumento considerável na demanda por processamento e memória.

É justamente aí que entra a utilização das estruturas de repetição. O mesmo algoritmo ficaria da seguinte maneira:

```
algoritmo “repete frase”  
var  
  contador: inteiro  
inicio  
  para contador de 1 ate 5 passo 1 faca  
    escreval (“Olá, bom dia!”)  
  fimpara  
finalgoritmo
```

No algoritmo acima, criamos uma variável de controle (*contador*) que é que determinam quantas vezes o comando *escreval* será executado. A repetição só para quando a condição não for atendida, ou seja, quando o contador for maior que cinco.

Repetição ou looping é um conjunto de ações que são executadas repetidamente.

O objetivo das estruturas de repetição é executar uma série de instruções em looping, enquanto (ou até que) uma dada condição analisada seja verdadeira.

Tipos de Estruturas de Repetição:

As estruturas de repetição podem funcionar de diferentes formas, de acordo com a necessidade do algoritmo que você vai criar. Assim como nas estruturas de decisão, há uma classificação para as estruturas de repetição:

1. Repetição com teste de condição no início (comandos ENQUANTO...FAÇA) – repete as instruções enquanto a condição for verdadeira.
2. Repetição com variável de controle (comando PARA) – repete as instruções sob controle de um contador que percorre valores, de acordo com limites iniciais e finais pré-estabelecidos.
3. Repetição com teste da condição no final (comandos REPITA...ATÉ) – repete as instruções até que a condição seja verdadeira.

Nesta aula, especificamente, nos ateremos à estrutura de repetição com variável de controle.

Estrutura de Repetição com Variável de Controle:

Na repetição com variável de controle, a quantidade de execuções do comando que irá repetir é conhecida previamente. Para isso, precisamos de uma variável que funcione como um contador para que as repetições possam cessar no momento em que a contagem chegar ao fim. Existe também, um valor de incremento, que define de quantas em quantas unidades a repetição irá ser executada.

No exemplo dado no começo desta aula (que escreve a mesma frase cinco vezes), utilizamos esse tipo de estrutura de repetição.

A sintaxe é a seguinte:

Pseudocódigo:

para <variável inteiro> de <valor inicial> ate <valor final> passo <valor de incremento>
faça

<instruções>

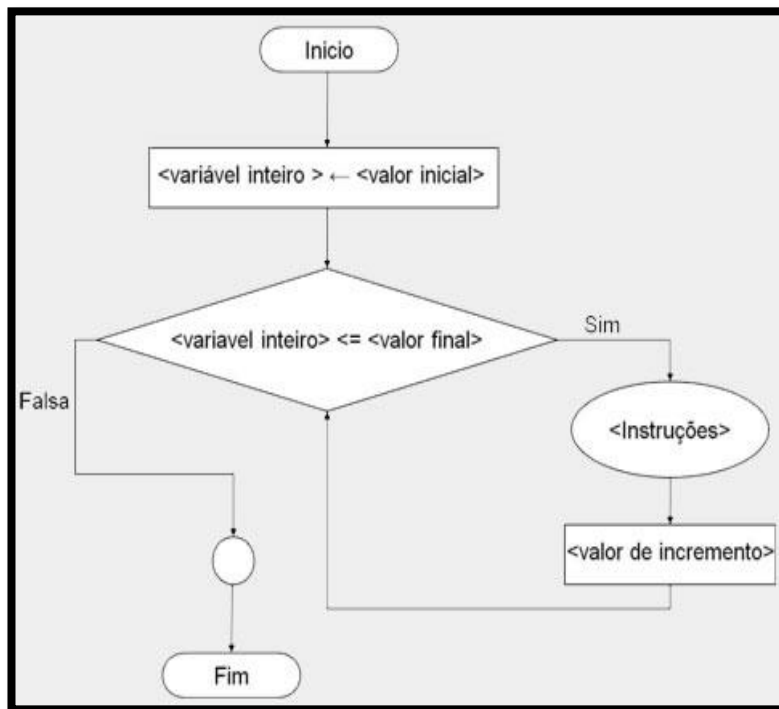
Fimpara

OU

para <variável inteiro> de <valor inicial> ate <valor final> faça
<instruções>

Fimpara

Fluxograma: Veja a seguir algumas considerações sobre a sintaxe da estrutura.



- A variável de controle deve ser uma variável numérica do tipo inteiro, pois servirá de contador e terá seu valor incrementado a cada passo.

- O valor inicial corresponde ao valor de inicialização da variável antes da primeira repetição.

- O valor final corresponde ao valor máximo que a variável pode alcançar.

- O valor inicial, o valor final e o valor de incremento podem ser variáveis ou

constantes.

- Se o valor de incremento não for definido, instantaneamente o programa assume o valor = 1.
- O valor de incremento não pode ser nulo.
- Pode ser atribuído um valor negativo ao valor de incremento, caso se deseje que o laço seja percorrido em ordem inversa. Nesse caso, o valor inicial e o valor final devem ser invertidos, também, para evitar erros na execução. Ou seja, se o valor de incremento for negativo, o valor inicial deve ser maior que o valor final.

Essa estrutura de repetição obedece a seguinte ordem:

1. É declarada uma variável de controle numérica do tipo inteiro (que será o contador), antes da execução da estrutura de repetição.
2. São definidos um valor inicial e um valor final para essa variável (que vai definir a quantidade de repetições).
3. É atribuído o valor inicial à variável.
4. O valor que está armazenado na variável é comparado com o valor final.
5. Se o valor da variável for menor ou igual ao valor final, as instruções contidas na estrutura são executadas.
6. O valor da variável é incrementado de acordo com o que foi definido no comando passo.
7. Isso se repete até que o valor armazenado na variável seja maior que o valor final. Quando isso ocorre, a estrutura termina e o algoritmo segue após o comando fimpara.

Vamos rever o nosso exemplo da média, só que desta vez usando a estrutura de repetição com variável de controle.

Exemplo:

Calculando a média entre duas notas para 50 alunos:

```
algoritmo "calcula média com repetição"
var

nota1, nota2, media : real //declaração das variáveis

contador : inteiro //declaração de variável de controle

inicio

para contador de 1 ate 50 passo 1 faca

  escreval ("Digite as notas do aluno ", contador, ":")

  escreval ("Primeira nota:")

  leia (nota1)

  escreval ("Segunda nota:")

  leia (nota2)

  media <- (nota1+nota2)/2

  escreval ("O aluno ", contador, " teve média igual a: ", media)

fimpara

finalgoritmo
```

Como na estrutura sequencial, declaramos as variáveis para as notas e a média. Em seguida, declaramos uma variável inteira, que será o contador.

Definimos, depois, os valores inicial e final (de 1 a 50, que é a quantidade de alunos) e o valor de incremento (1, pois o laço percorrerá um a um, os alunos).

Os comandos do "para" são executados em *loop* até que seja satisfeito o número 50 na variável contadora, e assim, quando é exibida a média do último aluno, a estrutura termina.

Enquanto, faça – Teste condicional no começo do algoritmo:

Antes, fizemos o cálculo de médio dos 50 alunos utilizando as estruturas com variável de controle, com os comandos PARA...FAÇA. Agora, veremos esse mesmo algoritmo de forma diferente:

```
algoritmo
var
nota1, nota2, media: real //declaração das variáveis do alg.
contador: inteiro //declaração do contador
inicio
enquanto contador < 5 faça //início da estrutura de repetição
escreval ("Digite as notas do aluno ",contador,":")
escreval ("Primeira nota:")
leia (nota1)
escreval ("Segunda nota:")
leia (nota2)
media <- (nota1+nota2)/2
escreval ("O aluno ",contador," teve média igual a:", media)
contador <- contador + 1
fimenquanto
finalgoritmo
```

Veja que o algoritmo acima, utilizando o comando ENQUANTO, faz a mesma coisa que o algoritmo que vimos na aula anterior, com os comandos PARA...FAÇA.

No algoritmo da média, sabemos previamente a quantidade de repetições, que equivale ao número de alunos da turma. Por isso, esse algoritmo pode ser resolvido com qualquer uma das duas estruturas de repetição: o PARA ou o ENQUANTO.

Você sabe que o comando PARA define o valor de incremento do contador antes do início da execução. Essa definição não é feita na sintaxe do comando ENQUANTO. Por isso, é de extrema importância que você se lembre de incrementar o contador, antes de finalizar a estrutura.

Veja que antes do comando fim enquanto, temos a seguinte expressão.

```
contador <- contador + 1
```

Você declarou, no início do algoritmo, a variável contador, do tipo inteiro, mas não atribuiu a ela valor algum. Sabemos que, quando não atribuímos valor a uma variável do tipo inteiro, o valor default assumido é 0. Portanto, o valor do *contador*, antes do início da execução da estrutura de repetição, é igual a 0.

A cada repetição, o valor do contador deve ser incrementado em 1. É isso que o algoritmo faz quando chega à linha "contador <- contador + 1".

Início	0	contador <- contador + 1
Primeira execução	contador = 0	contador = 0 + 1
Segunda execução	contador = 1	contador = 1 + 1
Terceira execução	contador = 2	contador = 2 + 1
Quarta execução	contador = 3	contador = 3 + 1
Quinta execução	contador = 4	contador = 4 + 1
Fim da execução	contador = 5	contador atingiu valor limite

Veja que, a cada repetição, o valor da expressão de incremento (contador + 1) é armazenado na variável contador. Dessa forma, a cada repetição, essa variável assume um novo valor. E assim, quando o contador chegar em 5 irá terminar a execução, já que foi definido para que a condição máxima sendo menor que 5.

Outra diferença importante entre os comandos PARA e ENQUANTO é que o comando ENQUANTO pode ser utilizado em situações que não podem ser resolvidas com variável de controle. São situações em que não se sabe previamente a quantidade de repetições da estrutura.

O comando ENQUANTO testa uma condição e, enquanto essa condição for verdadeira, as instruções da estrutura de repetição são executadas. No momento em que a condição deixar de ser verdadeira, a estrutura encerrará.

Como não podemos prever qual será o momento em que o usuário irá digitar um número negativo, não há como saber previamente a quantidade de repetições.

O comando ENQUANTO é utilizado em estruturas de repetição em que não sabemos previamente a quantidade de repetições do comando ou grupo de comandos da estrutura.

Observe a sintaxe desta estrutura:

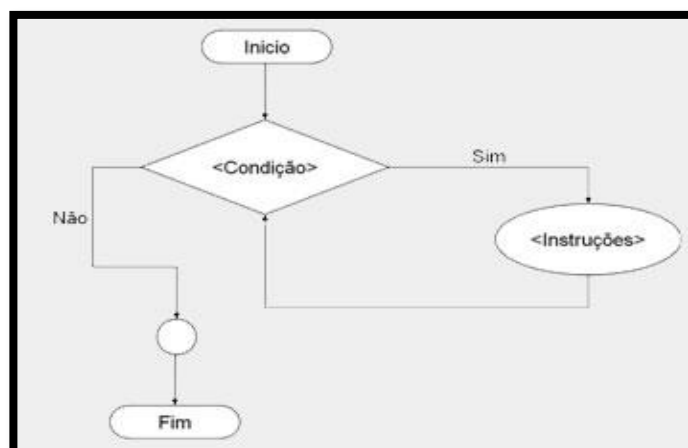
Pseudocódigo:

enquanto <condição> faça

<instruções>

fimenquanto

Fluxograma:



Atenção: se houver alguma situação em que seja possível a condição ser sempre verdadeira, as instruções da estrutura executarão para sempre. É o que chamamos de loop infinito. É importante que você tenha o cuidado de evitar situações como essa.

Quer ver um exemplo de algoritmo que resulta em loop infinito? Retorne ao algoritmo da média e tente executá-lo no VisuAlg removendo a linha em que o contador é incrementado (contador <-

contador + 1). Veja o que acontece.

Como o contador não está sendo incrementado terá sempre o valor inicial (zero). Dessa forma, a condição (contador < 5) será sempre verdadeira, causando, portanto, a situação de loop infinito.

Repita...até – Teste condicional no final:

Veremos novamente, o algoritmo da média:

```
algoritmo "media com repita...até"
var
Nota1, nota2, media : real //declaração de variáveis do algoritmo
contador: inteiro //declaração do contador
inicio
contador <- 1 //atribuição de valor ao contador
repita //início da estrutura de repetição
escreval ("Digite as notas do aluno ",contador, ":")
escreval ("Primeira nota:")
leia (nota1)
escreval ("Segunda nota:")
leia (nota2)
media <- (nota1 + nota2)/2
escreval ("O aluno ",contador," teve média igual a:", media)
contador <- contador + 1
ate contador > 50
fimalgoritmo
```

Na estrutura REPITA...ATÉ, a repetição continua enquanto a condição (contador > 50) for falsa. E cessa quando essa condição se tornar verdadeira.

Veja que o problema da média pôde ser resolvido com qualquer uma das estruturas de repetição, alterando-se somente a sintaxe.

Apesar disso, nem sempre um problema, em algoritmos, poderá ser solucionado com qualquer uma das estruturas. Sempre haverá uma das estruturas que se enquadre melhor na solução de determinado problema.

Ao contrário do ENQUANTO, o teste da condição verifica, na estrutura REPITA...ATÉ, se a condição é falsa. O looping termina quando a condição é verdadeira. Além disso, a estrutura REPITA...ATÉ sempre executa, pelo menos uma vez, os comandos, pois o teste da condição é realizado no final.

Exercícios:

1. Qual a sintaxe da estrutura de repetição?

2. Qual o objetivo das estruturas de repetição?

3. Explique a sintaxe da estrutura ENQUANTO...FAÇA.



LÓGICA DE PROGRAMAÇÃO

DESENVOLVEDOR

9
aula

Aula 9 – Vetores

Nas aulas anteriores, estudamos as estruturas de controle. Você viu que tais estruturas são utilizadas para definir o fluxo de execução dos comandos nos algoritmos. Viu também as estruturas sequenciais, com comandos básicos e ordem simples na execução desses comandos; as estruturas de decisão, que estabelecem desvios na execução dos comandos, de acordo com condições específicas; e as estruturas de repetição, em que determinados comandos são executados em laço.

Até agora, na execução dos comandos com as estruturas de controle, utilizamos tipos de dados básicos, com variáveis simples: real, inteiro, literal, caractere e lógico. Nesta aula, você verá que há situações em que os tipos de dados básicos não são suficientes para resolver os problemas que se apresentam.

Em muitos momentos, precisaremos, para atingir a solução de tais problemas, utilizar o que chamamos de estruturas de dados homogêneas. Você entenderá o conceito dessas estruturas e verá que, utilizando-as, podemos armazenar diversos dados de um mesmo tipo em uma única variável. Você estudará a utilidade dessas estruturas e as classificaremos em vetores (unidimensionais) e matrizes (bidimensionais), contemplando a sintaxe e a construção de algoritmos utilizando as duas estruturas.

Estruturas de dados homogêneas:

Ao longo da nossa disciplina, você viu que uma variável é um espaço na memória que é reservado para armazenar determinados tipos de dados. Até agora, porém, nós trabalhamos apenas com os tipos de dados básicos (reais, inteiros, caracteres, literais e lógicos) e variáveis simples para armazenar esses dados.

No entanto, nem sempre os tipos de dados básicos são suficientes para representar as estruturas de dados necessárias para resolver os problemas que se apresentam.

Imagine, por exemplo, que você quer escrever um programa que solicita ao usuário a entrada dos nomes de 50 alunos de uma turma de sua escola. Parece simples, não? E é. Basta você utilizar uma das estruturas de repetição que vimos nas aulas anteriores.

Exemplo:

```
algoritmo "50 nomes"
var
nome: literal //conhecido como "caractere" em algumas linguagens
contador: inteiro
inicio
escreval ("Digite os nomes dos alunos:")
para contador de 1 ate 50 passo 1 faca
escreval ("Aluno",contador,":")
leia (nome)
escreval ("Aluno", contador,": ",nome)
fimpara
fimalgoritmo
```

O algoritmo acima solicita que o usuário digite os nomes e lê um a um, conforme o usuário os digita. Neste caso, precisamos apenas de uma variável inteira que servirá como

contador e uma variável do tipo literal (também conhecida como caractere) para armazenar o nome que será digitado.

O nome é digitado, armazenado na variável, exibido ao usuário, e a variável é liberada para armazenar o próximo nome. Se você solicitar ao programa que exiba o conteúdo da variável nome, verá que somente o último nome digitado será exibido, já que, da maneira como foi escrito o algoritmo, é impossível exibir os nomes anteriores.

Mas, se quisermos que todos os nomes digitados sejam exibidos em uma lista, imediatamente após a digitação dos mesmos? Como fazer isso?

Para que os nomes dos alunos do nosso exemplo sejam exibidos em uma lista, eles precisam primeiro, ser acessados. E para ser acessados, é necessário que todos eles estejam armazenados nessa lista. Para esse fim, existem as estruturas de dados homogêneas.

Estruturas de dados homogêneas são estruturas que permitem armazenar conjuntos de dados de um mesmo tipo (daí vem o nome “homogêneas”) em uma única variável.

As estruturas de dados homogêneas são classificadas em dois tipos:

1. Os vetores (ou arrays), estruturas que armazenam os dados em uma única linha e várias colunas (dizemos que são unidimensionais).
2. As matrizes, estruturas que armazenam os dados em forma de tabela, com várias linhas e várias colunas (são bidimensionais).

Vetores:

Voltando ao nosso exemplo, como exibir os nomes dos alunos em uma lista, utilizando as estruturas de dados homogêneas? Antes de resolver esse problema, vamos analisar o passo a passo dessa situação.

Primeiro, devemos inserir no programa esses dados. Ou seja, devemos escrever, um a um, os nomes dos alunos, armazenando-os em uma lista. Em seguida, solicitamos que o programa exiba a lista dos alunos. Os dados armazenados são, então, acessados e, em seguida, exibidos.

Se tentássemos resolver esse problema com o que aprendemos até aqui, teríamos que declarar 50 variáveis literais para armazenar os 50 nomes e depois teríamos que instruir o computador a ler, uma a uma, cada variável. Algo similar ao trecho de programa abaixo:

```
leia (nome1,nome2,nome3,...nome50)

escreva (nome1,nome2,nome3,...nome50)
```

Não precisa dizer que a solução acima seria inviável, não é mesmo? Na realidade, a solução ideal para o problema pode ser encontrada se utilizarmos uma variável composta unidimensional, ou seja, um vetor.

Declaração de um vetor:

Assim como as variáveis simples, os vetores precisam ser declarados antes de serem utilizados. A declaração de um vetor, porém, é um pouco diferente da declaração de uma variável comum, pois se trata de uma variável indexada. É como se estivéssemos declarando diversas variáveis dentro de uma só, diferenciada por um índice.

Essas “variáveis” correspondem aos elementos do vetor (em nosso exemplo, os nomes dos alunos). Já o índice é um valor numérico do tipo inteiro, que sempre começa em 1 e corresponde à posição de cada elemento no vetor.

Exemplo de um vetor:

1	2	3	4	5
Maria	Roberto	Gustavo	Fernanda	Filipe

O exemplo representa um vetor de 5 elementos. Os números de 1 a 5 representam os índices, que são posições de cada elemento no vetor, por exemplo, o elemento “Gustavo” ocupa a posição 3 do vetor.

Ao declarar um vetor, o seu “tamanho” deve ser informado. O “tamanho” de um vetor é a quantidade de dados que será armazenada na variável. Na tabela acima, o tamanho do vetor é 5. No nosso exemplo (dos nomes dos alunos), o tamanho do vetor é 50, pois queremos armazenar os nomes de 50 alunos distintos.

A sintaxe de declaração de um vetor fica da seguinte forma:

<identificador> : vetor [tamanho] de [tipo]

Tamanho = [Vi..Vf], onde: Vi = valor inicial e Vf = valor final

No exemplo dos nomes:

nomes: vetor [1..50] de literal

No nosso exemplo, ao invés de declararmos diversas variáveis (nomes1, nomes2..., nomes50), estamos declarando diversos elementos (os nomes) em uma variável.

Atribuição em Vetores:

Assim como as variáveis comuns, os elementos de um vetor também podem ser inicializados, ou seja, ter seus valores atribuídos no momento da declaração.

Ao atribuir valores a um elemento do vetor, deve ser obedecida a seguinte sintaxe:

<identificador>[posição] <- <valor>

Exemplos:

Nomes [3] <- “Eduardo Carvalho de Almeida”

i <- 5

Nomes [i] <- “Maria Alice da Silva”

Agora que você já conheceu os conceitos e a sintaxe dos vetores, vamos resolver nosso problema.

Ao digitar os nomes dos alunos, utilizamos uma estrutura de repetição. Fizemos isso, você sabe, porque se trata de uma execução em loop. Para exibir os dados, teremos que utilizar outra estrutura de repetição, pois a exibição também é feita em loop. Ou seja, o vetor exibirá os dados um após o outro, até que todos sejam exibidos.

Veja a solução a seguir:

```
algoritmo "nomes dos alunos"
var
  nomes: vetor[1..50] de literal //declaramos aqui, o vetor (o tamanho fica nos números
entre colchetes)
  contador: inteiro //declaração da variável contador
inicio
  escreva ("Digite os nomes dos alunos:")
  para contador de 1 ate 50 passo 1 faca //inicio da primeira estrutura de repetição
    escreva ("Aluno ", contador, ":")
    leia (nomes[contador]) //os dados são armazenados um a um, no vetor
  fimpara
  para contador de 1 ate 50 passo 1 faca //inicio da segunda estrutura de repetição
    escreva (nomes[contador]) //os dados são exibidos
  fimpara //fim da segunda estrutura de repetição
finalgoritmo
```

Perceba que há uma estrutura de repetição para armazenar os dados e outra para exibi-los. Execute o algoritmo no VisuAlg. Utilizando a tecla F8 para o passo a passo.

Ordenação de um vetor:

Ao escrever algoritmos com vetores, haverá momentos em que você sentirá a necessidade de ordenar os seus elementos.

Você já deve ter visto ou utilizado algum programa de computador que tem funcionalidades que classificam palavras em ordem alfabética ou números em ordem crescente.

A ordenação é um tipo de funcionalidade bastante útil e necessária a alguns programas de computador.

Essa ordenação é realizada por meio de comparações entre os elementos do vetor.

1	2	Posição na memória
A = 7	B = 5	Elementos a ordenar

$A > B$, então:

1	2	Posição na memória
A = 5	B = 7	Elementos ordenados

Quando há uma quantidade maior de elementos a serem ordenados, há a necessidade de técnicas específicas de ordenação. Vamos conhecer uma delas.

Veja bem a tabela a seguir:

1	2	3	4	5
5	2	12	8	1

Os números 01 a 05 na primeira linha da tabela representam as posições na memória (os índices do vetor) de cada elemento. Na linha de baixo, estão os elementos, fora de ordem.

Para ordenar os elementos, é necessário fazer a comparação entre todos eles, da seguinte maneira: primeiro, comparamos o número do índice 01 com cada um dos outros.

O menor é colocado na posição 01. Depois, é realizada a comparação do segundo elemento com os restantes e assim por diante.

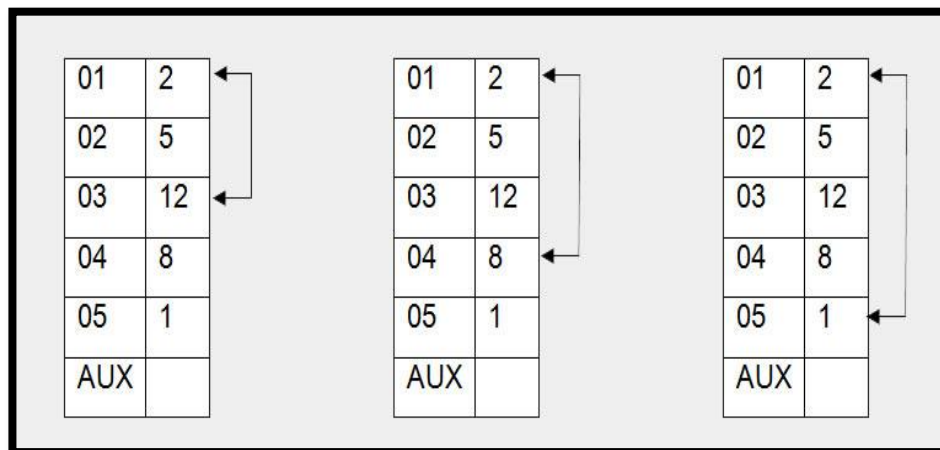
Porém, como cada elemento ocupa um espaço na memória, precisamos reservar um espaço auxiliar (uma variável temporária) para armazenar um dos elementos da comparação, quando eles tiverem que trocar de posição.

O passo a passo da troca de valores ocorre da seguinte forma:

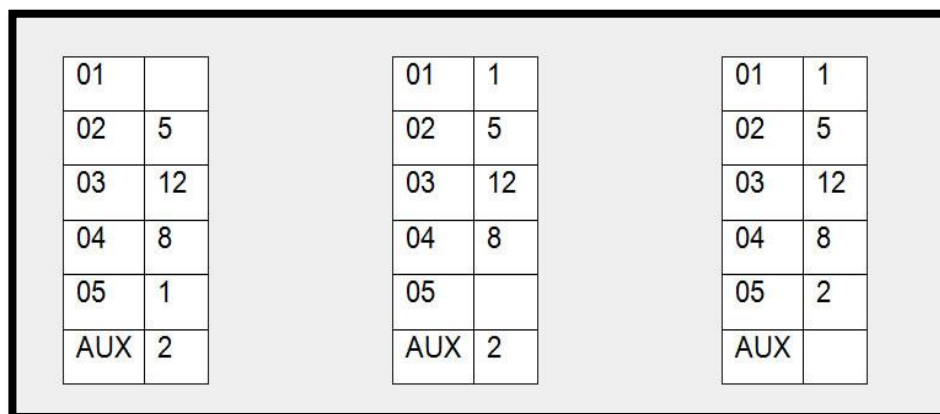
I - Criamos uma variável auxiliar e comparamos o número do primeiro índice com todos os demais, até encontrar o menor, armazenando-o na primeira posição.

(i)	(ii)	(iii)	(iv)																																																
<table><tr><td>01</td><td>5</td></tr><tr><td>02</td><td>2</td></tr><tr><td>03</td><td>12</td></tr><tr><td>04</td><td>8</td></tr><tr><td>05</td><td>1</td></tr><tr><td>AUX</td><td></td></tr></table>	01	5	02	2	03	12	04	8	05	1	AUX		<table><tr><td>01</td><td></td></tr><tr><td>02</td><td>2</td></tr><tr><td>03</td><td>12</td></tr><tr><td>04</td><td>8</td></tr><tr><td>05</td><td>1</td></tr><tr><td>AUX</td><td>5</td></tr></table>	01		02	2	03	12	04	8	05	1	AUX	5	<table><tr><td>01</td><td>2</td></tr><tr><td>02</td><td></td></tr><tr><td>03</td><td>12</td></tr><tr><td>04</td><td>8</td></tr><tr><td>05</td><td>1</td></tr><tr><td>AUX</td><td>5</td></tr></table>	01	2	02		03	12	04	8	05	1	AUX	5	<table><tr><td>01</td><td>2</td></tr><tr><td>02</td><td>5</td></tr><tr><td>03</td><td>12</td></tr><tr><td>04</td><td>8</td></tr><tr><td>05</td><td>1</td></tr><tr><td>AUX</td><td></td></tr></table>	01	2	02	5	03	12	04	8	05	1	AUX	
01	5																																																		
02	2																																																		
03	12																																																		
04	8																																																		
05	1																																																		
AUX																																																			
01																																																			
02	2																																																		
03	12																																																		
04	8																																																		
05	1																																																		
AUX	5																																																		
01	2																																																		
02																																																			
03	12																																																		
04	8																																																		
05	1																																																		
AUX	5																																																		
01	2																																																		
02	5																																																		
03	12																																																		
04	8																																																		
05	1																																																		
AUX																																																			

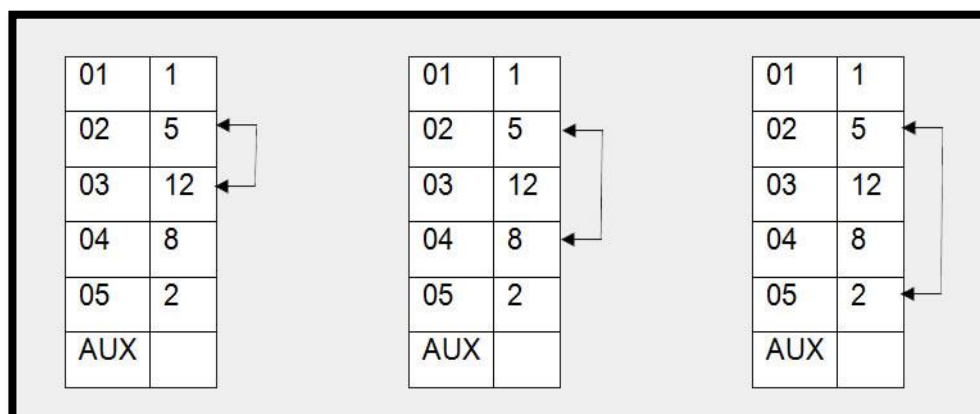
II - Como houve troca de posições entre os elementos, realizamos a comparação entre o primeiro elemento (que agora é o "2") com os demais, até achar um menor.



III – Ao encontrar um número menor ("1"), realizamos a troca de posições entre os dois, da mesma forma que a troca anterior, utilizando a variável auxiliar.



IV – Já temos o número menor. Agora, fazemos as comparações entre o número que está na segunda posição e os demais, até encontrar o segundo menor e trocamos os dois de posição.



V – Procedemos à troca de posições.

01	1
02	
03	12
04	8
05	2
AUX	5

01	1
02	2
03	12
04	8
05	
AUX	5

01	1
02	2
03	12
04	8
05	5
AUX	

VI - Com o primeiro e o segundo números definidos, fazemos as comparações entre o terceiro número e os restantes, trocando sempre as posições, a fim de que os menores fiquem nas primeiras posições.

Comparamos e trocamos o "12" com o "8".

01	1
02	2
03	12
04	8
05	5
AUX	

01	1
02	2
03	
04	8
05	5
AUX	12

01	1
02	2
03	8
04	
05	5
AUX	12

01	1
02	2
03	8
04	12
05	5
AUX	

Comparamos e trocamos o "8" com o "5".

01	1
02	2
03	8
04	12
05	5
AUX	

01	1
02	2
03	
04	12
05	5
AUX	8

01	1
02	2
03	5
04	12
05	
AUX	8

01	1
02	2
03	5
04	12
05	8
AUX	

Novamente trocamos o 12 com o 8. A ordenação está completa.

01	1	01	1	01	1	01	1
02	2	02	2	02	2	02	2
03	5	03	5	03	5	03	5
04	12	04		04	8	04	8
05	8	05	8	05		05	12
AUX		AUX	12	AUX	12	AUX	

Consegue perceber o princípio desse tipo de ordenação? Vamos ver o funcionamento desse algoritmo em pseudocódigo:

Exemplo:

algoritmo "ordenação"

var

i:inteiro //contador1

j:inteiro //contador2

num: vetor[1..5] de inteiro //declaração do vetor que será ordenado

temp: inteiro

inicio

para i de 1 ate 5 faca

escreval ("Numeros" ,i, "?") //solicita a entrada dos números

leia (num[i])

fimpara

para i de 1 ate 5 faca //duas estruturas de repetição, pois dois números será comparados

para j de 1 ate 5 faca

se num[i]<num[j] entao //compara dois números

temp <- num[j] //coloca o maior na variável temporária

num[j] <- num [i] //coloca o menor no lugar do maior

num[i] <- temp //coloca o maior que estava na variável temporária, no lugar em que antes

tinha o menor.

fimse

fimpara

fimpara

para i de 1 ate 5 faca

escreva (num[i]) //exibe os números ordenados

fimpara

fimalgoritmo

Exercícios:

1. De que forma é declarado um vetor? Exemplifique com uma sintaxe.

2. O que é o tamanho de um vetor? Explique pra que serve a indexação.

3. Para que serve a variável temporária (usada como “temp” no exemplo) nos métodos de ordenação?



LÓGICA DE PROGRAMAÇÃO

DESENVOLVEDOR

10
aula

Aula 10 – Matrizes

Como funciona:

Vimos na aula anterior, que nem sempre os tipos de dados básicos são suficientes para representar as estruturas de dados necessárias para resolver os problemas que se apresentam.

Viu a necessidade de, em algumas situações, exibir dados em listas. E, para atender a essa necessidade, conhecemos os vetores, como estruturas de dados unidimensionais, com variáveis indexadas referenciadas por um único índice.

Você aprendeu que, armazenando os dados em um vetor e acessando-o, em seguida, podemos exibir suas informações em uma lista.

Porém, os vetores lidam com apenas uma dimensão. Ou seja, se fôssemos representar o vetor através de uma tabela, essa só teria uma linha com várias colunas, ou vice-versa. Como podemos ver na tabela abaixo, que vimos há pouco no estudo sobre vetores.

Veja:

1	2	3	4	5
Paulo	Jéssica	Márcio	Viviane	Leonardo

Os elementos do vetor, na tabela, são os nomes dos alunos Paulo, Jéssica, Márcio, Viviane e Leonardo. Dessa forma, os elementos e seus índices são:

```
nomes[1] <- "Paulo"  
nomes[2] <- "Jéssica"  
nomes[3] <- "Márcio"  
nomes[4] <- "Viviane"  
nomes[5] <- "Leonardo"
```

Agora, imagine que o professor de determinada disciplina passou um trabalho a ser feito em dupla e quer organizar os nomes das duplas. Veja a tabela abaixo:

Grupos	Componente 1	Componente 1
1	Paulo	Jéssica
2	Márcio	Viviane
3	Leonardo	Pedro

Os elementos dos grupos, segundo a tabela, são os seguintes:

```
Componentes [1,1] <- Paulo (componente 1 do grupo 1)  
Componentes [2,1] <- Jéssica (componente 2 do grupo 1)  
Componentes [1,2] <- Márcio (componente 1 do grupo 2)  
Componentes [2,2] <- Viviane (componente 2 do grupo 2)  
Componentes [1,3] <- Leonardo (componente 1 do grupo 3)  
Componentes [2,3] <- Pedro (componente 2 do grupo 3)
```

Temos, portanto, 3 grupos, cada um com 2 componentes. Fátima, por exemplo, é o componente 1 do grupo 2. Estamos tratando, portanto, de uma matriz.

Portanto, uma matriz é uma estrutura de dados homogênea de duas (ou mais) dimensões. Uma matriz utiliza variáveis indexadas de mais de um índice.

Agora, passaremos isso aos algoritmos.

Primeiramente, vamos conhecer a sintaxe de uma matriz.

```
<identificador> : vetor [<tamanho1>, <tamanho2>] de <tipo>  
Tamanho1 = tamanho da linha da matriz  
Tamanho2 = tamanho da coluna da matriz
```

O exemplo fica então:

```
componentes : vetor [1..3,1..2] de literal  
1..3 = 3 linhas (3 grupos)  
1..2 = 2 colunas (2 componentes por grupo)
```

Observe que a matriz, no VisuAlg, é um vetor com duas dimensões, ou seja, com dois índices. Dessa forma, sua declaração é feita de forma semelhante à declaração do vetor, com a única diferença de que devemos informar o tamanho da segunda dimensão. A dimensão 1, no caso, é referente aos grupos (1..3) e a dimensão 2 é referente aos nomes dos alunos de cada grupo (1..2).

Vamos ver o algoritmo, abaixo:

```
algoritmo "matriz grupos"  
var  
componente : vetor[1..3,1..2]de literal //declaração da matriz  
cont1, cont2: inteiro //declaração dos contadores  
inicio  
  
para cont1 de 1 ate 3 faca  
para cont2 de 1 ate 2 faca  
escreval ("Digite o nome do componente",cont2," do grupo ",cont1, ":")  
leia (componente [cont1, cont2]) //recebe os dados  
fimpara  
fimpara //estrutura para exibir os dados recebidos e armazenados  
  
para cont1 de 1 ate 3 faca  
escreval ("Grupo ",cont1,":") //mostra o nome do grupo  
para cont2 de 1 ate 2 faca  
escreval (componente[cont1, cont2])  
fimpara  
fimpara  
fimalgoritmo
```

Observe que declaramos, além do vetor, dois contadores. Fizemos isso, porque temos dois índices (um para contar os grupos e outro para contar os componentes de cada grupo). Se tivéssemos três índices, teríamos que declarar três contadores, e assim por diante.

Você sabe que, para percorrer os elementos do vetor, precisamos fazer isso índice a índice. Por isso, precisamos do contador e de uma estrutura de repetição para armazenar os dados e outra para exibi-los.

Com as matrizes ocorre o mesmo. Porém, como temos dois índices, utilizamos duas estruturas de repetição, uma dentro da outra, para receber os dados, e mais duas para exibi-los. Se tivéssemos três índices, teríamos que utilizar três estruturas de repetição para cada uma dessas atividades.

Atribuição em matrizes:

Assim como com os vetores, os elementos de uma matriz também podem ser inicializados, ou seja, ter seus valores atribuídos no momento da declaração.

Ao atribuir valores a um elemento da matriz, fazemos de modo similar aos vetores:

```
<identificador>[posição] <- <valor>
```

Exemplos:

```
componentes[2,3] <- "José"
```

```
i <- 2
```

```
j <- 1
```

```
componentes [i,j] <- "João"
```

Solução:

Vamos ver um exemplo de como ficaria um algoritmo de Matriz para declaramos dois grupos de uma Copa do Mundo, por exemplo.

Veja á seguir:

```
algoritmo "Exemplo de Matriz - Copa do Mundo"
var
componente : vetor[1..2,1..4] de literal //declaração da matriz
cont1, cont2: inteiro //declaração dos contadores
inicio
escreval ("Exemplo de Matriz - Copa do Mundo")
para cont1 de 1 ate 2 faca
para cont2 de 1 ate 4 faca
escreval ("Digite o Nome do País ",cont2," do grupo ",cont1," : ")
```

```
leia (componente [cont1, cont2]) //recebe os dados  
  
fimpara  
  
fimpara //estrutura para exibir os dados recebidos e armazenados  
  
para cont1 de 1 ate 2 faca  
  
  escreval ("Grupo ",cont1,":") //mostra o nome do grupo  
  
  para cont2 de 1 ate 4 faca  
  
    escreval (componente[cont1, cont2])  
  
  fimpara  
  
fimpara  
  
finalgoritmo
```

Exercícios:

1. De que forma é declarada uma matriz? Explique sua sintaxe e exemplifique.

2. Analise o algoritmo do exercício resolvido e o implemente no VisuAlg.

3. Execute também, o algoritmo dos grupos, que vimos no começo desta aula.

