
Apostila de Lógica

Sumário

1 - Introdução aos Sistemas Computacionais	6
Introdução.....	7
O que é um computador?	7
O hardware e seus componentes.....	8
Gabinete	8
Fonte.....	8
Unidade Central de Processamento.....	8
Placa Mãe	10
Memórias	10
Periféricos	11
Sistema de numeração binário.....	11
Classificação das linguagens de programação	14
Linguagens de baixo nível e linguagens de alto nível.....	14
Síntese.....	15
2 - Introdução à lógica	17
Introdução.....	18
Introdução à lógica de relacionamentos	19
Procedimento de decisão	21
Lógica matemática.....	23
Algumas leis fundamentais	25
Tabela-verdade.....	25
3 - Noções de algoritmos de programação.....	31
Introdução.....	32
O que são algoritmos?.....	32
Exercícios	34
Formas de representação de um algoritmo.....	36
Pseudocódigo	36
Fluxogramas	37
Regras para a construção do algoritmo	38
Usar somente um verbo por frase	38
Ser objetivo;.....	38
Fases para a construção do algoritmo	38
Exemplo de algoritmo	40
Algoritmo Cálculo da Média.....	40
4 - Constantes, variáveis e tipos de dados.....	43
Introdução.....	44
Constantes e variáveis	44
Constantes.....	44

Variáveis	45
Tipos de variáveis	45
Numérico	45
Alfanumérico ou literais	46
Lógico	46
EXERCÍCIOS.....	49
Comentando algoritmos.....	51
Atribuindo valores às variáveis	52
Exercícios: atribuindo valores às variáveis.....	52
EXERCÍCIOS:.....	53
Expressões	54
Operadores aritméticos	54
Operadores relacionais.....	56
Operadores lógicos.....	58
Prioridade de operadores	61
Algoritmo A	61
EXERCÍCIOS:.....	66
5 – Processamento seqüencial e condicional	68
Introdução	69
Processamento seqüencial: comandos de entrada e saída.....	69
Comando Leia	70
Comando Escreva	71
Exercícios	75
Processamento condicional.....	78
Exemplos com Processo Condicional.....	79
Exercite “seqüencial e condicional”	85
Exercícios	87
6 – Métodos de Repetição.....	90
Introdução	91
Estruturas de controle de repetição.....	91
Exercícios	104
7 - Manipulação de vetores.....	106
Introdução	107
Conceito e declaração de vetores	108
Operação de vetores	110
Algoritmos com manipulação de Vetores	113
Síntese.....	118
EXERCÍCIOS:.....	118
8 - Manipulação de Matrizes	120
Introdução.....	121

Conceito e declaração de matrizes	122
Operação de matrizes	124
Algoritmos com manipulação de matrizes	128
Síntese.....	130
EXERCÍCIOS.....	130
9 - Manipulação de Registros.....	131
Introdução.....	132
Conceito e declaração de registros	133
Como podemos ler e escrever nessas variáveis?	134
Operação com registros	134
Criando novos tipos de variáveis	136
Criando um conjunto de registros.....	139
Algoritmos com manipulação de registros.....	140
Síntese.....	144
10 - Programação Estruturada.....	148
Introdução.....	149
Modularização: conceitos iniciais	149
Mas, o que é uma função?	150
Por que usar funções?	151
11 - Criando o Banco de Dados	152
Tabela de Clientes.....	155
Tabela de Fornecedores.....	156
Tabela de Pedidos.....	157
Tabela de Itens do Pedido.....	158
Tabela de Produtos	158
Tabela de Notas Fiscais (Vendas).....	160
Tabela de Itens de Notas Fiscais	161
12 - Introdução à Orientação a Objetos.....	162
Conceito.....	163
Classe	163
Objeto	164
Atributo.....	164
Método	164
Construtores	166
Herança	167
Considerações Finais	169

1 - INTRODUÇÃO AOS SISTEMAS COMPUTACIONAIS

Introdução

Devemos sempre lembrar que o computador não faz nada sozinho sem antes ter uma pessoa dizendo a ele o que fazer como fazer e quando fazer. Precisamos ter em mente que a parte inteligente de uma solução de um problema está na pessoa. Ela é quem soluciona um problema logicamente e apenas diz ao computador o que fazer. O computador é apenas um meio de se conseguir programar a solução lógica de maneira que os resultados possam ser obtidos de uma forma bastante rápida.

Bem, Você está começando agora um estudo fascinante e desafiador: o estudo da lógica de programação e o uso de computadores para resolver tarefas com maior rapidez, menor esforço e com melhores resultados.



Sabemos que os computadores têm entrado em nossas vidas sem mesmo serem convidados. Isso porque basta Você ir a uma biblioteca, que lá estão eles. Se você vai visitar um conhecido no hospital, primeiro verificam pelo computador em que quarto o doente está. Se Você se hospeda em um hotel, primeiro é feito o seu cadastro no computador. Até mesmo em supermercados, onde Você verifica os preços, a quantidade comprada etc. Enfim, nos dias atuais, quase tudo parece circular em torno do computador. Em uma linguagem bastante popular, podemos dizer que ele é o “bam-bam- bam”, o mandachuva, o poderoso.

Porem sabe que o computador, por si só, não é nada. Parece até um contra-senso em relação ao que acabei de afirmar, mas pode acreditar o computador, por si só, não passa de um conjunto de circuitos eletrônicos, uma tela, uma caixa de som etc. Para alguns é quase uma caixa preta, um desconhecido. Para que essa ferramenta tenha valor, o ser humano é quem deve dizer a ela o que fazer, quando e como.

O importante é que a inteligência, de fato, está no ser humano e não no computador. Utilizamos o computador apenas para lidar com grandes quantidades de informações e de forma rápida, mas se não usarmos a inteligência para conseguir tirar proveito dessa máquina, então ela continuará sendo uma caixa preta.

Se Você acha que o computador é uma caixa preta indecifrável, então nada melhor do que alguns conceitos para começar a decifrar essa poderosa ferramenta.

O que é um computador?

Um computador é uma máquina composta de um conjunto de partes eletrônicas e eletromecânicas, com capacidade de coletar, armazenar e manipular dados além de fornecer informações, tudo isso de forma automática.

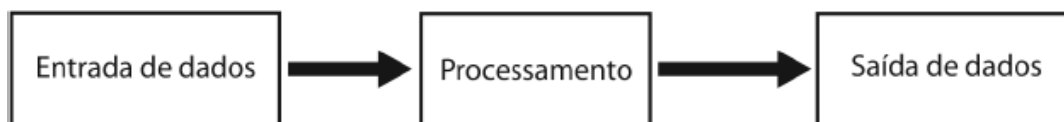


Figura 1.1: Diagrama de computador

O computador funciona, basicamente, como uma máquina que recebe dados, processa-os e retorna um ou mais dados como resultado.

Como funciona a entrada de dados? Quem executa o processamento? Ou ainda, o que é processamento e para que me sirva isso? Onde vejo os resultados de saída?

Bem, todo computador é composto de partes essenciais para o seu funcionamento: hardware e software. Sem uma dessas partes o computador não funciona.

Compare o computador com um carro. Um carro também é composto de motor, pneus, volante etc. Precisamos de todas as partes para que o mesmo funcione

Mais ainda, precisamos de um bom motorista para guiá-lo de maneira adequada. Em termos de analogia, o carro é o computador e o motorista é o desenvolvedor / programador.

Você pode estar pensando: mas eu não preciso conhecer um carro por completo para dirigi-lo! Você está certo. Você precisa saber apenas algumas coisas como: onde está o volante, onde se liga o carro, o que fazer para arrancar o carro, o que fazer para parar etc. Enfim, Você deve manuseá-lo de forma correta para obter os resultados necessários, ou Você já acelerou um carro pensando que estava freando? Para evitar tais enganos, conheça mais um conceito.

hard = rígido, duro (físico) e
ware = mercadoria (parte);
soft = leve, mole.

O hardware e seus componentes

O hardware é a parte física do computador. É o próprio gabinete do computador e seus periféricos (teclado, monitor, mouse etc.).

O hardware de um computador possui diversos componentes:

Gabinete

Conte a fonte, a placa-mãe, os dispositivos de armazenamento, as placas de expansão, a memória, o processador, a placa de rede etc.

Existem vários modelos e tamanhos variados.

Exigem compatibilidade com o tipo de fonte e, em alguns casos, com a placa mãe.

Fonte

Recebe corrente alternada de 110 ou 220 volts, vinda do estabilizador e a transforma em corrente contínua de 5, -5, 12 e -12 volts.

Unidade Central de Processamento

A Unidade Central de Processamento, também conhecida pela sigla inglesa CPU (Central Processor Unit), é o componente vital do sistema de computação, responsável pela realização das operações de processamento (cálculos matemáticos, cálculos Lógicos etc.) e de controle, durante a execução de um programa.



Figura 1.2: Gabinete

As funções da CPU são:

- Buscar uma instrução na memória, uma de cada vez - fase de leitura.
- Interpretar a instrução - decodificar.
- Buscar os dados onde estiverem armazenados, para trazê-los a CPU.
- Executar a operação com os dados.
- Guardar se for o caso, o resultado no local definido na instrução.
- Reiniciar o processo, apanhando nova instrução.

Para efetuar tais procedimentos, a CPU é composta por vários componentes:

Unidade Lógica e Aritmética (ULA) – responsável por realizar as operações matemáticas com os dados.

Registradores - destinados ao armazenamento temporário de dados.

Unidade de Controle (UC) - é o dispositivo mais complexo da CPU, responsável pela busca de instruções na memória principal e determinação de seus tipos, controla a ação da ULA, realiza a movimentação de dados e instruções de e para a CPU.

Relógio – dispositivo gerador de pulsos cuja duração é chamada de ciclo. A quantidade de vezes em que este pulso se repete em um segundo define a unidade de medida do relógio, denominada de frequência.

A unidade de medida usual para a frequência dos relógios da CPU é o Hertz (HZ), que significa um ciclo por segundo. Como se trata de frequências elevadas abreviam-se os valores usando-se milhões de Hertz, ou ciclos por segundo – MHz.

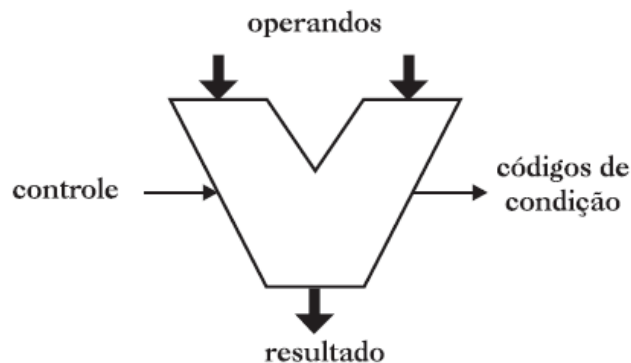


Figura 1.3: ULA

Placa Mãe

A placa mãe, ou motherboard, é possivelmente a parte mais importante do computador. Ela gerencia toda a transação de dados entre a CPU e os periféricos. Ela define a arquitetura do seu computador.

A placa mãe possui os seguintes componentes:

- CPU;
- Memória;
- Barramentos;
- Slot.

Memórias

Quando um programa está sendo executado, são processados os dados através de vários comandos.

Para que esse processamento tenha velocidade, é necessário que dados e comandos estejam sempre à disposição e que não sejam perdidos.

Isso é viabilizado pela memória principal, onde a Unidade de Controle (UC) armazena dados e comandos e carrega-os, sempre que necessário, de volta para o armazenamento.

Fazendo uma analogia, a memória principal é como uma grande sala com vários armários, contendo gavetas e pastas, que você pode utilizar para guardar suas coisas.

Cada gaveta corresponde a uma parte de memória; assim como a gaveta pode guardar pastas, cada parte da memória pode armazenar dados, e esse lugar de cada memória chamamos de endereço de memória, reconhecido pelo computador.

O exemplo do que você pode fazer com as gavetas de sua sala, colocando nomes de identificação, o mesmo ocorre com os endereços de memória do computador.

Podemos referenciá-los com nomes.

As memórias podem ser de vários tipos:

a) RAM

A memória principal do computador é conhecida por RAM (Random Access Memory). Na memória principal estão as instruções que estão sendo executadas e os dados necessários à sua execução.

Todo programa que você executa é armazenado na memória RAM, seja ele um software antivírus, um protetor de tela, impressão, ou o próprio sistema operacional.

A memória principal, também chamada de memória de trabalho ou memória temporária, é uma memória de leitura e escrita (read/write).

Suas características são: rápido acesso (da ordem de nanossegundos em computadores mais modernos), acesso aleatório e volatilidade.

b) ROM

O computador possui, também, uma memória chamada ROM (Read Only Memory) onde são guardadas informações para iniciar o computador, ativando o sistema operacional.

Esta memória é não-volátil e, em geral, gravada pelo fabricante e com pequena capacidade de armazenamento.

Geralmente, depois de gravada, a ROM não pode ser mais gravada pelo usuário.

c) Memória secundária (discos)

A memória secundária ou memória auxiliar é usada para armazenar grandes quantidades de informações.

Um exemplo comum de memória secundária são os discos rígidos que são usados para armazenar grandes volumes de informações. Outros exemplos são: disco flexível, CD-ROM, o Zip Drive, DVD e Pen Drive.

A leitura e a gravação de dados nas memórias secundárias são realizadas pelos periféricos de entrada/saída.

Periféricos

Unidades de entrada/saída.

São dispositivos que recebem dados (imagem, letras, números, sons e outros) do meio externo e são capazes de traduzi-los para pulsos (sinais) elétricos compreensíveis para o computador.

Por exemplo, quando você pressiona alguma tecla no teclado do seu computador, um dado é enviado para o computador que o interpreta e executa a operação.

Exemplos: teclado, mouse, monitor, impressora, scanner etc.

Sistema de numeração binário

Como são armazenados os dados em um computador?

Que linguagem ele conhece? Será a mesma que nós conhecemos

É simples e fácil de entender. O computador, como ferramenta de processamento, transforma os dados de entrada em sinais elétricos.

Cada sinal elétrico é chamado de Bit (Binary digit) ou dígito binário.

Bit "é uma palavra formada pelas duas primeiras letras da palavra binárias" e pela última letra de "dígito" (binary digit, em inglês).

Quem inventou a palavra foi um engenheiro belga, Claude Shannon, em sua obra Teoria Matemática da Computação, de Nela, Shannon descrevia um bit como sendo uma unidade de informação.

O bit é à base de toda a linguagem usada pelos computadores, o sistema binário, ou de base dois, graficamente é representado por duas alternativas possíveis: ou o algarismo 0, ou o 1.

É como se lá dentro da máquina houvesse um sistema de tráfego com duas lâmpadas: a informação entra e encontra a lâmpada 1, segue em frente até a lâmpada seguinte. Se encontra com a lâmpada 0, muda de direção. São bilhões de informações repetindo essas manobras a cada milésimo de segundo sequencialmente.

Se um microcomputador é de 32 bits, isso significa que ele consegue processar 32 unidades de informação ao mesmo tempo e, portanto, é mais rápido que um de 16 bits.

8 (oito) bits compõem um byte, uma unidade completa de informação.

Os bits são geralmente usados como medida de velocidade na transmissão de dados, enquanto os bytes são normalmente associados à capacidade de armazenamento de dados (um disco rígido com memória de 20 gigabytes).

Num sistema binário usam-se só dois dígitos (o 0 e o 1) para representar qualquer número. Comparado com o sistema de base decimal, a relação é a seguinte:

Sistema decimal

1 2 3 4 5 6 7 8 9 10

Sistema binário

0001 0010 0011 0100 0101 0110 0111 1000 1001 1010

Por exemplo, em notação binária, 1000000000000 – 13 algarismos - corresponde ao numeral 4096, de apenas quatro algarismos.

Parece complicado, mas a vantagem do sistema binário e a sua simplicidade, pelo menos do ponto de vista do computador: se cada um dos dez algarismos arábicos tivesse que ser reconhecido individualmente pelo sistema, os cálculos demorariam muito mais.

Um modem transmite bits por segundo (bps).

Os bits não servem apenas para representar números, mas para qualquer coisa que precise ser informada a um computador. De uma letra ou uma vírgula, até a cor que iremos usar. Cada uma dessas informações é transformada em um código binário é interpretada pelo sistema.

Por exemplo, ao ler 01001010 01000001 0100001101001011, o computador saberia que isso, obviamente, queria dizer JACK.

Tal qual no sistema numérico decimal, cada posição de "bit" (dígito) de um número binário tem um peso particular, o qual determina a magnitude daquele número. O peso de cada posição é determinado por alguma potência da base do sistema numérico.

Para calcular o valor total do número, considere os "bits" específicos e os pesos de suas posições (a tabela a seguir mostra uma lista condensada das potências de 2).

Por exemplo, o número binário 110101 pode ser escrito com notação posicional como segue: $(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

Para determinar o valor decimal ao número binário 1101012, multiplique cada "bit" por seu peso posicional e some os resultados.

$(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 32 + 16 + 0 + 4 + 0 + 1 = 5310$

Agora, se temos um número decimal, como vamos convertê-lo para a base binária?

Dividir sucessivamente por 2 o número decimal e os quocientes que vão sendo obtidos até que o quociente de uma das divisões seja 0.

O resultado é a sequência de baixo para cima de todos os restos obtidos.

1010 = 01010 = 10102

Também na base 2 os zeros à esquerda não são necessários.

Além de representar os números decimais, os números binários podem também representar os números fracionários, as letras do alfabeto, enfim, todos os caracteres que você possa encontrar no teclado do seu computador (til, acentuações, colchetes, aspas, ponto de exclamação etc.)

Tudo é representado através de uma numeração binária. Com isso, o computador sabe que informação está processando e armazenando quando necessário.

Para organizar tais informações podemos representar todos os caracteres em uma tabela, chamada tabela ASCII (American Standard Code for Information Interchange).

A tabela ASCII é usada pela maior parte da indústria de computadores para a troca de informações. Cada caractere é representado por um código de 8 bits (um byte).

Veja a seguir a tabela ASCII de 7 bits para alguns caracteres apenas.

Caracter	Decimal	Hexadecimal	Binário Comentário
Espaço	32	20	0010 0000
!	33	21	0010 0001
"	34	22	0010 0010

#	35	23	0010 0011
\$	36	24	0010 0100
%	37	25	0010 0101
&	38	26	0010 0110
'	39	27	0010 0111
(40	28	0010 1000
)	41	29	0010 1001
*	42	2A	0010 1010
+	43	2B	0010 1011
,	44	2C	0010 1100
-	45	2D	0010 1101
.	46	2E	0010 1110

Existe uma tabela estendida para bits que inclui os caracteres acentuados.

Caracter	Decimal	Hexadecimal	Binário Comentário
/	47	2F	0010 1111
0	48	30	0011 0000
1	49	31	0011 0001
2	50	32	0011 0010
3	51	33	0011 0011
4	52	34	0011 0100
5	53	35	0011 0101
6	54	36	0011 0110
7	55	37	0011 0111
8	56	38	0011 1000

É bom lembrar que o sistema binário é bem antigo – em relação à televisão, por exemplo, é 100 anos mais velho. Os cartões perfurados do século XIX já o utilizavam: ou uma determinada posição tinha um furo (1) ou não tinha (0). Depois, viriam as fitas perfuradas de papel (mais estreitas que um cartão, e em bobinas, contínuas) e finalmente as finas fitas magnéticas, tipo fita de áudio. Mas o princípio continuou o mesmo: na fita, ou um espaço estava magnetizado (1) ou não estava (0).

Atualmente, se fala muito em GigaByte, vulgo giga, que equivale a 1073741824 bytes ou o número 2 elevado a potência 30.

Uma página normal de um livro tem cerca de 3000 caracteres, logo um gigabyte seria equivalente a mais de 6000 livros com 500 páginas cada um.

O software é um conjunto de programas que comandam o funcionamento do hardware.

Os softwares são elaborados a partir de algoritmos de programação (foco do nosso estudo), que são seqüências de passos que levam a solução de um problema.

Um programa é um conjunto de operações necessárias para, a partir de dados de entrada, obter um resultado que será disponibilizado por algum dispositivo de saída.

Por exemplo, quando você digita as teclas representando 2, +, 3 e = na calculadora do Windows, o programa receberá os números 2 e 3, a operação + e o sinal de = executará os comandos de realizar a soma dos dois números 2 e 3 e retornará o resultado na saída (vídeo) que é o número 5.

Para fazer programas, utilizamos linguagem de programação .

Uma linguagem de programação precisa suportar a definição de ações e prover meios para especificar operações básicas de computação, além de permitir que os usuários especifiquem como estes passos devem ser sequenciados para resolver um problema.

Uma linguagem de programação pode ser considerada como sendo uma notação, que pode ser usada para especificar algoritmos com precisão.

Classificação das linguagens de programação

As linguagens de programação podem ser agrupadas em dois grandes grupos:

Linguagens de baixo nível e linguagens de alto nível.

As linguagens de baixo nível são restritas à linguagem de máquina e tem uma forte relação entre as operações implementadas pela linguagem e as operações implementadas pelo hardware.

As linguagens de alto nível, por outro lado, aproximam-se das linguagens utilizadas por humanos para expressar problemas e algoritmos. Cada declaração em uma linguagem de alto nível equivale a várias declarações numa linguagem de baixo nível.

A vantagem principal das linguagens de alto nível é a abstração.

Isto é o processo em que as propriedades essenciais requeridas para a solução do problema são extraídas enquanto esconde os detalhes da implementação da solução adotada pelo programador.

Com o nível de abstração aumentado, o programador pode concentrar-se mais na solução do problema em vez de preocupar-se como o hardware vai tratar do problema.

As linguagens de programação também têm sua gramática, que vamos chamar de sintaxe.

É necessário conhecer a sintaxe de uma linguagem para poder programar. Por analogia, é necessário conhecer bem a gramática da Língua Portuguesa para escrever corretamente um texto, certo? Mais ainda, fazer com que outros possam entender adequadamente o que você colocou no texto.

Dessa forma, a linguagem de programação, por estar muito próxima do ser humano, está mais longe da linguagem do computador, tendo em vista que o mesmo só entende zeros e uns (bits).

Como posso fazer com que o meu computador possa entender essa linguagem de alto nível?

Para isso, as linguagens de programação têm o que chamamos de compilador.

O compilador é um programa que converte a linguagem de alto nível (conhecida por você) em uma linguagem de baixo nível (conhecida pela máquina).

Assim, todo o programa escrito por uma linguagem de programação deve ser compilado para que o mesmo gere uma linguagem de zeros e uns (bits), conhecida pelo computador.

Na compilação, toda a sintaxe é verificada. Se você digitou algo que o programa compilador não conheça, ele indicará o erro através de uma mensagem e não irá gerar a linguagem de baixo nível conhecida pelo computador.

Sintaxe: toda a linguagem de programação, incluindo aqui o pseudocódigo possui um conjunto de representações e/ou regras que são utilizadas para criar um programa.

Por meio dessas representações é que podemos estabelecer uma comunicação com o computador, fazendo com que ele execute o que nos queremos que ele faça. A esse conjunto de representações e/ou regras damos o nome de sintaxe da linguagem.

Há uma grande quantidade de linguagens de programação hoje no mercado. Elas foram evoluindo com o tempo e, atualmente, há linguagens de programação para todos os tipos e gostos.

É necessário aprender todas elas?

A resposta é não. Você não precisa dominar todas as linguagens, mas precisa dominar a lógica de programação.

Essa sim é importante, porque é ela que expressa a solução de um problema, e a própria inteligência descrita passo a passo.

Portanto, não se apresse em entender uma linguagem de programação sem antes entender a lógica de programação.

Na próxima unidade Você começará a trabalhar com essa lógica de programação e suas representações.

EXERCÍCIOS

1. Sabemos que todos os números podem ser representados na sua forma binária, forma que o computador entende. De acordo com o processo explicado nesta unidade de conversão de números decimais para números binários, faça as conversões dos seguintes números:

20
35
100
256
1000
255

2. Você viu também nesta unidade que a cada 8 bits temos um byte. Para os números convertidos na atividade 1, quantos bytes cada um ocupa na memória?

3. Os caracteres alfanuméricos também são armazenados em forma de bits. Para a palavra LÓGICA, mostre o conjunto de bits que representa cada uma das letras. Importante: é essa forma binária que o computador entende. Você deve procurar em um livro de programação ou em um site para verificar a tabela ASCII completa.

Síntese

Vimos que um computador nada mais é do que um amontoado de circuitos eletrônicos, formado por um gabinete, vídeo, teclado etc., com capacidade de coletar, armazenar e manipular dados além de fornecer informações, tudo isso de forma automática.

Vimos, também, que um dos principais componentes do computador é sua Unidade Central de Processamento, também conhecida pela sigla inglesa CPU (Central Processor Unit). É o componente vital do sistema de computação, responsável pela realização das operações de processamento (cálculos matemáticos, cálculos lógicos etc.) e de controle, durante a execução de um programa.

Entre os componentes, também vimos a placa mãe, ou motherboard, que é possivelmente a parte mais importante do computador.

Ela gerencia toda a transação de dados entre a CPU e os periféricos. Ela define a arquitetura do seu computador, possuindo os seguintes componentes: CPU, Memória, Barramentos e Slot.

Por fim, as memórias. Elas estão divididas em memória ROM (Read-Only Memory) ou memória somente para leitura, RAM (Random Access Memory) ou memória de acesso aleatório e as memórias secundárias, também chamadas de auxiliares, usada para armazenar grandes quantidades de informações (discos rígidos, CD-ROM, disquete etc.).

Tudo no computador é representado por dígitos binários. Vimos que o bit é a base de toda a linguagem usada pelos computadores, o sistema binário, ou de base dois, e graficamente é representado por duas alternativas possíveis: ou o algarismo 0, ou o 1. 8 (Oito) bits compõem um byte, uma unidade completa de informação.

Por fim, definimos linguagem de programação como sendo uma notação que pode ser usada para especificar algoritmos com precisão. Uma linguagem de programação é utilizada para elaborar programas e um conjunto de programas forma um software.

Vale à pena lembrar, que a ideia geral do curso é fazer você pensar logicamente, resolver um problema sistematicamente através de representações lógicas que permitam construir programas/software utilizando uma linguagem de programação.

Quanto melhor for o seu algoritmo, melhores serão os seus resultados em qualquer linguagem de programação que você utilizar.

2 - INTRODUÇÃO À LÓGICA

Introdução

As pessoas utilizam a lógica no cotidiano sem prestar muita atenção, chegando mesmo a citá-la, sem saber o seu significado.

Segundo o dicionário Aurélio, lógico é a "coerência de raciocínio, de idéias, seqüência coerente, regular e necessária de acontecimentos, de coisas". Você pode verificar essas coerências de raciocínio em exemplos práticos da vida real.

Veja a seguir.

1. O número 5 é menor que 10.
2. O número 15 é maior que 10.
3. Logo, 5 é menor que 15. Parece lógico, não?

Outro exemplo bem fácil.

Dada a seqüência de números, qual é o próximo da lista, ou seja, o valor de X?

0, 2, 4, 6, X,.....

A resposta certa é o número 8 (oito), visto que os números mostrados são pares e o seu antecessor é o número 6 (seis). Estamos simplesmente utilizando lógica, mesmo que de maneira intuitiva.

Você acha que a lógica está limitada a exemplos de matemática?

Não. Veja mais alguns exemplos a seguir.

1. Só me atraso para o serviço quando o ônibus se atrasa.
2. Hoje cheguei atrasado;
3. Logo, o ônibus estava atrasado. É uma boa desculpa para o seu chefe.
1. Quando chove, não preciso regar as plantas do jardim.
2. Ontem choveu e hoje não;
3. Logo, ontem não precisei regar as plantas. Já no dia de hoje, precisei.

São inúmeros os exemplos, dos mais fáceis até os mais complicados, mas todos eles sempre obedecem a uma coerência lógica a fim de que possamos alcançar soluções para um determinado problema.

O objetivo principal desta disciplina é demonstrar técnicas para resolução de problemas e, conseqüentemente, tornar as tarefas rotineiras fáceis de fazer.

O aprendizado da lógica é essencial para formação de um bom programador, servindo como base para o aprendizado de todas as linguagens de programação.

Portanto, a coisa mais importante a fazer quando estiver aprendendo lógica e mantiver o foco de atenção nos conceitos é evitar perder-se em detalhes técnicos.

O objetivo de aprender lógica de programação é tornar-se um bom programador, isto é, tornarem-se mais hábil no desenvolvimento de programas e/ou sistemas, incluindo aqui os desenvolvidos para ambiente web.

Introdução à lógica de relacionamentos

No nosso dia a dia estamos sempre fazendo relacionamentos, mesmo sem perceber.

Por exemplo, ontem gastei no meu almoço R\$ 7,00. Hoje, como estava com mais fome, gastei R\$ 10,00.

É fácil pensar que, se gastei mais, gastei mais em relação a alguma coisa. Estou fazendo, na verdade, uma comparação em relação ao dia de ontem.

Portanto, quando comparo valores, medidas etc., estão fazendo relacionamentos entre esses valores, medidas etc., respectivamente.

No estudo da lógica matemática existe, também, o chamado raciocínio de relacionamento, que nos permite tirar conclusões sobre um determinado fato.

A nota mínima para passar no meu colégio é 7.0. Fiz duas provas na matéria de Português. Na primeira prova tirei 8.0 e, na segunda, 10.0. Logo, a minha média é $(8.0 + 10.0) / 2 = 9.0$. Como a nota 9.0 é maior que a nota para passar no colégio (7.0), concluo que passei de ano em Português.

Outro exemplo, agora não matemático: se Jonas é filho de Paulo que é casado com Ana; e Clara é filha de Ana com Paulo, o que Jonas e Clara são?

Podemos montar várias frases simples para deduzir o resultado.

1. Jonas é filho de Paulo; (relação Pai/Filho)
2. Paulo é casado com Ana; (relação Marido/Esposa)
3. Clarisse é filha de Ana e Paulo; (relação Pai/Filha)
4. Logo, Clarisse e Jonas são irmãos. (relação Irmãos).

Exercícios:

1. O pai de Mário é irmão do pai de Júlia. Mário tem uma filha chamada Margarida. O que Júlia é de Margarida?
(a) Prima (b) Tia (c) Irmã (d) Nora.
2. Se o pai de Paulo é sogro de Maria, e esta é cunhada de Ana por parte de Paulo, o que a Mãe de Paulo é para Ana?
(a) Tia (b) Avó (c) Sogra (d) Mãe.

Analise bem os passos que estão sendo executados como forma de se atingir um determinado objetivo. Quando queremos saber o que a mãe de Paulo é para Ana, realizamos relacionamentos entre as pessoas envolvidas para deduzir uma resposta, nesse caso, infalível e sem ambigüidades.

3. Que número fica diretamente acima de 119 na seguinte disposição de números?

Que número fica diretamente acima de 119 na seguinte disposição de números?

				1					
				2	3	4			
		5	6	7	8	9			
	10	11	12	13	14	15	16		
17	18	–	–	–	–	–	–	–	–
	(a)	98							
	(b)	99							
	(c)	100							
	(d)	101							

A resposta é a letra "b". Basta observar que o ultimo numero de cada linha e sempre um numero elevado a 2, ou seja, $(1)^2 = 1$, $(2)^2 = 4$, $(3)^2 = 9$, $(4)^2 = 16$, ...

Logo, a linha que possui o numero 119 termina com o número 121, que é $(11)^2$.

				1					
			2	3	4				
	5	6	7	8	9				
10	11	12	13	14	15	16			
17	18	–	–	–	–	–	–	–	–
								99	100
								119	120 121

Então, a linha anterior deve acabar com o numero 100, pois $10^2 = 100$. O número 100 fica acima do numero 120 conforme você acabou de ver e o numero 99 fica acima dos 119. Portanto, nossa resposta e a letra b.

4. Certamente você já ouviu falar da banda de rock U2, certo? Vamos ao problema.

A banda U2 tem um concerto que começa daqui a 17 minutos e todos precisam cruzar a ponte para chegar lá.

Todos os quatro integrantes da banda estão do mesmo lado da ponte.

Você deve ajudá-los a passar de um lado para o outro.

É noite.

Na ponte só podem passar, no máximo, duas pessoas de cada vez.

Só há uma lanterna.

Qualquer pessoa que passe, uma ou duas, deve passar com a lanterna na mão.

A lanterna deve ser levada de um lado para o outro, e não pode ser jogada etc.

Cada membro da banda tem um tempo diferente para passar de um lado para o outro.

O par deve andar junto no tempo do menos veloz:

Bono - 1 minuto para passar

Edge - 2 minutos para passar

Adam - 5 minutos para passar

Larry - 10 minutos para passar

Por exemplo, se o Bono e o Larry passarem juntos, vai demorar 10 minutos para eles chegarem do outro lado. Se o Larry retornar com a lanterna, 20 minutos terão passados e o show sofrerá um atraso. Como organizar a travessia?

É um exercício que mostra certo grau de dificuldade em sua solução, bem mais complicada do que nos primeiros exercícios.



Procedimento de decisão

Os procedimentos de decisão representam a forma pela qual iremos tomar uma determinada decisão ao tentarmos resolver um problema qualquer.

Ninguém resolve uma questão de uma prova sem antes compreendê-la

Após ler o problema, precisamos entender o enunciado, saber qual é o verdadeiro problema a ser resolvido. De nada adianta buscarmos uma solução sem saber para que ela sirva e sem saber o que ela está propondo resolver.

O estudo de enunciados consiste em verificar se o que está escrito, ou o que foi dito, é verdadeiro ou falso.

Se hoje estiver chovendo...



vou ver um filme.



Senão,
vou caminhar.

Neste caso, o problema consiste em saber o que fazer se estiver chovendo.

Observe que é um enunciado que não deixa qualquer dúvida.

Não estou preocupado e nem tentando resolver outras questões como, por exemplo, o que vou fazer se o dia estiver com sol ou apenas nublado.

A minha questão é com a chuva e não com o sol nesse momento.

Uma possível solução me diz que, se o enunciado é verdadeiro, ou seja, se estiver chovendo realmente, então vou ver um filme.

Caso contrário, se o enunciado se verificar não verdadeiro, ou seja, não está chovendo, vou caminhar.

Perceba que a solução está completa para o meu problema, ou seja, determino que ação ou ações tome caso chova, ou não.

Os enunciados também podem ser compostos, com mais de uma instrução vinculada.

Exemplo:.

Se hoje estiver chovendo e estiver passando um filme bom no cinema, vou ver um filme.

Problema: consiste em saber o que fazer se estiver chovendo e estiver passando um bom filme no cinema.

Perceba que a ação de "vou ver um filme" é verdadeira se, e somente se, estiver chovendo e estiver passando um filme bom no cinema.

Caso uma ou outra condição não ser verdadeira, ou seja, se não estiver chovendo ou não estiver passando um filme bom, vou fazer outra coisa qualquer.

Mas que condições são essas mesmo? Há várias outras condições que posso testar para tomar decisões.

Não está chovendo e está passando um filme bom;

Não está chovendo e não está passando um filme bom;

Está chovendo e não está passando um filme bom.

Para cada condição, um conjunto de ações pode ser tomado, por exemplo, caso esteja chovendo e não está passando um filme bom, posso decidir por:

Inicialmente, estudar lógica de programação;
Depois, assistir a um jogo de futebol;
Ir dormir.

Perceba nos exemplos citados as questões colocadas: “Está chovendo ou não está chovendo”; “Está passando um bom filme ou não está passando um bom filme”, são situações que expressam uma resposta do tipo: Está ou não Está; Sim ou Não; ou é Verdade ou é Falso.

Não existe o talvez. Estamos trabalhando com respostas que sempre irão dizer se é sim ou não, verdadeiro ou falso etc., e mais nenhuma outra condição. Chamamos isso de respostas lógicas ou valores lógicos.

Para lembrar: quando falarmos em valores lógicos, estaremos falando em valores que possuem como respostas: Verdadeiro (V) ou Falso (F).

Daqui por diante, sempre vamos destacar essas palavras especiais:

Verdadeiro e Falso.

Lógica matemática

A lógica matemática, em síntese, pode ser considerada como a ciência do raciocínio e da demonstração. Este importante ramo da Matemática desenvolveu-se no século XIX, sobretudo por meio das idéias de George Boole, criador da Álgebra Booleana, que utiliza símbolos e operações algébricas para representar proposições e suas inter-relações.

As idéias de Boole tornaram-se a base da Lógica Simbólica, cuja aplicação estende-se por alguns ramos da eletricidade, da computação e da eletrônica.

A lógica matemática (ou lógica simbólica), trata do estudo das sentenças declarativas também conhecidas como proposições.

Diz-se então que uma proposição verdadeira possui valor lógico V (Verdade) e uma proposição falsa possui valor lógico F (Falso).

Matemático inglês (1815 - 1864).

Os valores lógicos também costumam ser representados por 0 (zero) para proposições falsas (0 ou F) e 1 (um) para proposições verdadeiras (1 ou V).

Segundo Quine (1942), “toda proposição é uma frase mas nem toda frase é uma proposição; uma frase é uma proposição apenas quando admite um dos dois valores lógicos: Falso (F) ou Verdadeiro (V)”.

Frases que não são proposições

Alguns alunos são pontuais.

Hoje é quarta-feira.

O dia está bonito.

Essas frases não são proposições, pois não conseguimos atribuir valores lógicos definidos (verdadeiro ou falso).

Por exemplo, a frase “o dia está bonito”, podemos dizer que está um dia daqueles, mais ou menos, nem feio e nem bonito.

E a frase “Hoje é quarta-feira”? Você consegue identificar por que ela não é uma proposição?

Frases que são proposições

A lua é o único satélite do planeta terra. (Verdadeiro)

Goiânia é a capital do estado do Amazonas. (Falso)

O número 714 é ímpar. (Falso)

A raiz quadrada de dois é um número irracional. (Verdadeiro)

Para representar as proposições, utilizamos as letras minúsculas do nosso alfabeto (a, b, a,...).

a = "A lua é o único satélite do planeta terra".

b = "Goiânia é a capital do estado do Amazonas".

Podemos dizer que a proposição "a" é Verdadeira (V) e que a proposição "b" é Falsa (F).

Composição de proposições

É possível construir proposições a partir de proposições já existentes.

Este processo é conhecido por composição de proposições.

Vamos supor que queremos abrir o cofre de um banco. Será aberto apenas se colocarmos a senha 62, ou seja, dígito 6 e, após, o dígito 2.

Nesse exemplo, temos duas proposições que são pelas letras "a" e "b".



Esse cofre
posiciono o

indicadas

a = "o primeiro dígito e o valor 6"

b = "o segundo dígito e o valor 2"

Poderíamos perguntar: como conseguiremos abrir o cofre?

A resposta certa é que precisamos colocar primeiro o dígito 6 e depois o número 2, certo? Sómente nesse caso, iremos obter sucesso. Qualquer alternativa, o cofre não abrirá.

Se colocarmos a proposição c = "o cofre está aberto", podemos perceber que "c" é uma composição das proposições anteriores "a" e "b", e a proposição "c" é verdadeira se, e somente se "a" é verdadeira e "b" também é verdadeira. Estamos realizando uma operação de conjunção e. A letra e está destacada na proposição justamente para mostrar essa operação de conjunção.

Agora, se colocarmos a proposição d = "o cofre não está aberto", o que posso imaginar? Pense bem...

Parece óbvio não é, que, ou eu não digitei o número 6 primeiro, ou não digitei o número 2 por último, ou mesmo os dois casos, nem digitei 6 e nem 2 na sequência.

Então pergunto o que faz a proposição "d" ficar verdadeira, ou seja, que o cofre não está aberto? Resposta: quando a proposição "a" e/ou "b" são falsas. Estamos realizando uma operação de disjunção ou. A palavra ou está destacada na proposição justamente para mostrar essa operação de disjunção.

Observe que a proposição "d" é a negação da proposição "a". Podemos representar isso por d = não (a).

Agora pense bem, se eu negar a proposição "d", o que estou afirmando? Nesse caso, estou negando a afirmação de que o cofre não está aberto, ou seja, o cofre está aberto, certo?

Poderíamos escrever essa situação como sendo a = não (d). Dizemos, neste caso, que estamos realizando uma operação de negação.

Sómente um pouco mais de raciocínio. O que você deduz se a proposição "c" "o cofre está aberto" for verdadeira? Se "c" for uma proposição verdadeira, isso implica que tanto a proposição "a" quanto "b" são verdadeiras.

Esse conjunto de relações pode ser melhor compreendido por meio de algumas leis básicas e de uma tabela expressando as relações entre as proposições citadas.

Algumas leis fundamentais

Lei do meio excluído

Uma proposição é falsa (F) ou verdadeira (V): não há meio termo.

Lei da contradição

Uma proposição não pode ser simultaneamente, V e F.

Lei da funcionalidade

O valor lógico (V ou F) de uma proposição composta é unicamente determinada pelos valores lógicos de suas proposições constituintes.

Exemplos da lei do meio excluído: o cofre está aberto ou fechado; o semáforo está aberto ou fechado; o número é par ou é ímpar; é cara ou coroa etc.

Da lei da contradição: não posso ter cara e coroa ao mesmo tempo; o semáforo não pode estar aberto e fechado ao mesmo tempo; se o cofre estiver aberto, ele não pode estar fechado etc.

Da lei da funcionalidade: a proposição "c" é verdadeira, ou seja, o cofre está aberto se, e somente se, as proposições "a" (o primeiro dígito é o valor 6) e "b" (o segundo dígito é o valor 2) forem verdadeiras, ou seja, defino o valor lógico de "c" pelas das proposições constituintes "a" e "b". Em função de "a" e "b" defino o valor da proposição "c".

Agora Você já sabe que as proposições são representadas por letras.

Para facilitar o raciocínio lógico, podemos representar as relações e expressões lógicas por meio de uma tabela, popularmente chamada de tabela-verdade.

Tabela-verdade

A tabela-verdade é uma estrutura lógica que facilita o raciocínio humano. É utilizada por programadores como forma de facilitar a construção de algoritmos de programação .

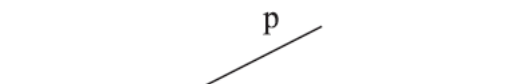
O nome tabela-verdade é apenas intuitivo. Ela é muito útil para representar as proposições e combinações entre elas (negação , conjunção e disjunção).

Para todos os exemplos a seguir, vamos considerar duas proposições, p e q. Para cada operação, será mostrada a tabela com explicações e exemplos.

Será utilizada a idéia de um interruptor.

Um interruptor é um dispositivo ligado a um ponto de um circuito, que pode assumir um dos dois estados, "fechado" ou "aberto".

No estado "fechado" (que indicaremos por 1) o interruptor permite que a corrente passe através do ponto, enquanto no estado "aberto" (que indicaremos por 0) nenhuma corrente pode passar pelo ponto.



Vamos ao primeiro caso. A operação de negação. A representação da negação é informada pelo sinal de acentuação til (\sim) antes da proposição. Assim, a negação da proposição p é $\sim p$.

Negação

p	$\sim p$
V	F
F	V

Você acabou de ver a tabela-verdade da negação de uma proposição.

Assim, se a proposição p é verdadeira, sua negação será falsa e vice-versa.

Se tivermos a seguinte proposição p = "o circuito está aberto". A negação de p é "o circuito não está aberto", ou seja, "o circuito está fechado". Então se a proposição p for verdadeira, a negação de p ($\sim p$) é falsa e vice-versa.

A frase "Hoje é quarta-feira" é uma proposição?

Você deve lembrar que uma proposição é uma frase que assume valores lógicos verdadeiro ou falso apenas.

A princípio você poderia pensar o seguinte: se hoje for realmente quarta-feira, então a sentença é verdadeira. Se for um outro dia qualquer, a sentença está falsa.

É necessário tomar certo cuidado na determinação de negações na linguagem natural, pois há uma propriedade bem simples a ser satisfeita que é a da dupla negação de uma proposição.

Uma proposição deve ter o mesmo valor de sua negação dupla.

Por exemplo, a proposição "a porta está aberta".

A negação da proposição "a porta está aberta" é "a porta não está aberta", ou seja, "a porta está fechada".

A negação desta última proposição "a porta está fechada" é "a porta não está fechada", ou seja, "a porta está aberta".

Ou seja, negando duas vezes a frase "a porta está aberta" chegamos à frase "a porta está aberta".

Então, concluímos que a frase é uma proposição entendeu?

Voltando ao caso "Hoje é quarta-feira".

"Hoje é quarta-feira" não é a negação de "Hoje é terça-feira".

Apesar de ambas satisfazerem a propriedade de uma só ser verdadeira se a outra for falsa, as duas podem ser simultaneamente falsas: Hoje pode ser sexta-feira.

Análise mais profundamente as proposições "Hoje é quarta-feira" e "Hoje é quinta-feira", como se uma fosse a negação da outra. É verdade que se uma for verdadeira, a outra será necessariamente falsa, mas o Princípio do Meio Excluído não está sendo satisfeito, pois há uma possibilidade de termos uma proposição e sua negação, ambas falsas.

Por exemplo, a negação de "Hoje é quarta-feira" não é necessariamente a afirmação de que "Hoje é quinta-feira", certo?

Hoje pode ser Sábado e já é hora de descansar.

Um outro caso a ser considerado é o da negação de proposições que utilizam o que os lógicos chamam de quantificadores.

Podemos afirmar a partir da intuição, que as proposições “Alguns alunos são pontuais” e “Alguns alunos não são pontuais” sejam uma a negação da outra.

Observe que ambas as proposições podem ser simultaneamente verdadeiras (e de fato é o que acontece com nossos alunos!) não satisfazendo, assim, o Princípio da Não-Contradição. Assim, a negação da primeira proposição deverá ser “Todos os alunos não são pontuais”.

Conjunção

p	q	p e q
V	V	V
V	F	F
F	V	F
F	F	F

Também, aqui, é necessário que tomemos cuidados ao traduzirmos de uma linguagem natural para a linguagem simbólica.

A tabela-verdade da conjunção nos fornece a propriedade comutativa para uma tal operação. Assim, a proposição p e q será equivalente a q e p. Perceba que o resultado de uma conjunção de proposições será verdadeira se, e somente se ambas forem verdadeiras.

Na representação do nosso circuito, temos dois interruptores em série:



Em série indicado por $p \bullet q$ ou pq ou $p * q$.

Neste caso, passa corrente se, e somente se $p=1$ e $q=1$ ou seja, estão ambos “fechados” o que corresponde na tabela-verdade a conjunção p e q .

Disjunção

p	q	p ou q
V	V	V
V	F	V
F	V	V
F	F	F

Aprenda agora a necessária distinção entre os dois tipos de disjunção existentes.

Uma delas é o que chamamos de disjunção inclusiva, que também é nomeada pela expressão e/ou, e que aparece na proposição “Ela não veio à prova por estar doente ou cansada”, pois é possível que as duas coisas tenham ocorrido.

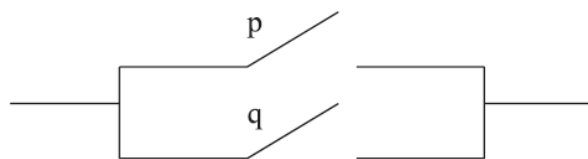
Disjunção: do Latim Disjunctione.

Ato de disjuntir; separação; afastamento.

Conjunção: do Latim Conjunctione. É uma junção; ligação; união.

O outro caso é o da disjunção exclusiva, também chamada de ou/ou, como em “Ele veio à prova ou não veio à prova”, em que não é possível que as duas coisas tenham acontecido. Na tabela-verdade anterior, tratamos como Você pode verificar, da disjunção inclusiva.

Na representação do nosso circuito, temos dois interruptores em paralelo:



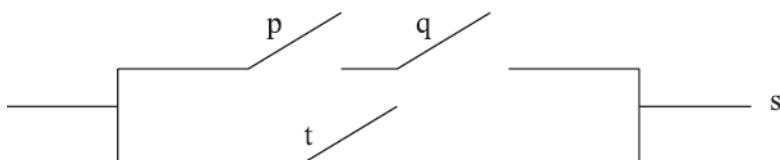
Em paralelo indicado por $p + q$.

Neste caso, não passa corrente se e somente se $p=0$ e $q=0$, ou seja, estão ambos "abertos" o que corresponde na tabela-verdade a disjunção p ou q .

Podemos realizar varias combinações de proposições também.

Por exemplo, uma proposição $s = p*q + t$. Como você leria essa proposição? Como seria o circuito representando a proposição?

Bem, para responder essa pergunta, podemos construir exemplos da vida real. Vamos supor que a proposição s seja "a lâmpada está acesa". Quando s é verdadeira? Imagine que temos três interruptores, 2 em série, que são p e q , e um em paralelo t , como mostra a figura a seguir.



Quando s é verdadeira? Quando fechamos o contato em p e q , ou seja, fazemos com que p e q valham 1, ou fechamos o contato t , fazendo com que o mesmo valha 1. Quando s será falsa? Quando o contato t estiver aberto e pelo menos p ou q estiverem abertos. Em qualquer outro caso, s será verdadeira, ou seja, "a lâmpada estará acesa".

Se fossemos construir uma tabela-verdade para o exemplo, teremos uma combinação de todas as situações possíveis, veja:

p	q	t	s
V	V	V	V
V	V	F	V
V	F	V	V
V	F	F	F
F	V	V	V
F	V	F	F
F	F	V	V
F	F	F	F

Observe quando a proposição s é verdadeira.

1. Quando todas as proposições p , q e t são verdadeiras;
2. Quando p e q são verdadeiras, independente da proposição t ;
3. Quando t é verdadeira independente da proposição p e q .

A grande vantagem de construir uma tabela – verdade é que ela facilita o raciocínio humano, mostrando sempre de forma clara quando as proposições serão verdadeiras ou falsas, principalmente quando temos combinações de proposições mais complicadas.

Síntese

Nesta unidade Você viu que no estudo da lógica matemática, existe, também, o chamado raciocínio de relacionamento que nos permite tirar conclusões sobre um determinado fato. Relacionamos números, pessoas etc., com o objetivo de alcançar uma resposta.

Baseado no valor lógico (verdadeiro ou falso) da proposição ou da combinação de proposições, podemos tomar decisões. Os procedimentos de decisão representam a forma pela qual iremos tomar uma determinada decisão ao tentarmos resolver um problema qualquer. Por exemplo, podemos ter a seguinte proposição:

a = "está chovendo".

O mecanismo de tomada de decisão poderia ser algo assim:

Se a proposição "a" for verdadeira então...

Fico em casa e assisto a um filme. Senão, (...) vou à praia.

Ou seja, tomamos uma decisão em função da proposição a ("está chovendo") ser verdadeira, ou falsa. Quando falarmos em valores lógicos, estaremos falando em valores que possuem como respostas:

Verdadeiro (V) ou Falso (F).

Você estudou, também, que podemos organizar melhor nossos procedimentos de tomada de decisão por meio de proposições.

Uma proposição verdadeira possui valor lógico V (Verdade) e uma proposição falsa possui valor lógico F (Falso).

Por fim, Você viu que as proposições podem se relacionar e que esse conjunto de relações pode ser melhor compreendido por meio de algumas leis básicas e de uma tabela expressando as relações entre as proposições. Chamamos essa tabela intuitivamente de tabela-verdade. Pela tabela-verdade podemos representar as funções de negação, conjunção e disjunção, facilitando o raciocínio humano no processo de tomada de decisão.

EXERCÍCIOS

1. O que é lógica de relacionamento?

2. O que são proposições?

3. Cite, pelo menos, três frases que não são proposições.

4. Construa tabela-verdade para as seguintes proposições.

4.1 $c = a + b$

4.2 $d = s * t + e$

4.3 $d = (a + b) * c$

4.4 $d = a * b * c$

3 - NOÇÕES DE ALGORITMOS DE PROGRAMAÇÃO

Introdução

Os temas desta unidade foram organizados de forma a apresentar alguns conceitos de Algoritmos de Programação, por meio de algumas formas de representação dos mesmos (Pseudocódigo e Fluxogramas).

Na sequência, você vai estudar as variáveis de programação e comandos de entrada e saída. Finalmente irá conhecer os comandos lógicos para solucionar um problema específico.

Nesta unidade vamos trabalhar com os comandos sequenciais, condicionais e laços de repetição.

O que são algoritmos?

Diariamente, executamos uma série de ações com vista a alcançar um determinado objetivo. Intuitivamente, aquilo que estamos executando é um algoritmo.

Veja algumas definições.

Um algoritmo é uma sequência de instruções ordenadas de forma lógica para a resolução de uma determinada tarefa ou problema.

O conceito central da programação e da Ciência da Computação é o conceito de algoritmos, isto é, programar é basicamente construir algoritmos.

Um algoritmo é uma receita para um processo computacional e consiste de uma série de operações primitivas, interconectadas devidamente, sobre um conjunto de objetos. Os objetos manipulados por essas receitas são as variáveis.

Serve como modelo para programas, pois sua linguagem é intermediária a linguagem humana e as linguagens de programação.

Um algoritmo é formalmente uma sequência finita de passos que levam à execução de uma tarefa.

Podemos pensar em algoritmo como uma receita, uma sequência de instruções que tem a função de atingir uma meta específica. Estas tarefas não podem ser redundantes nem subjetivas na sua definição, devendo ser claras e precisas.

Como exemplos, podemos citar:

Os algoritmos das operações básicas (adição, multiplicação, divisão e subtração) de números reais decimais;

Um manual de instruções de um videocassete, que explica passo a passo como fazer uma gravação em VHS, por exemplo.

Até mesmo as coisas mais simples, podem ser descritas por sequências lógicas.

Por exemplo:

"Chupar uma bala".

*Pegar a bala;
retirar o papel;
chupar a bala;*

jogar o papel no lixo.

"Somar dois números quaisquer".

Escreva o primeiro número;

Escreva o segundo número;

Some o primeiro número com o segundo e escreva o resultado.

Na verdade, você certamente utiliza algoritmos todos os dias sem perceber. Sempre que você realiza um conjunto de ações lógicas para alcançar algo, você está executando um algoritmo.

Veja o exemplo a seguir:

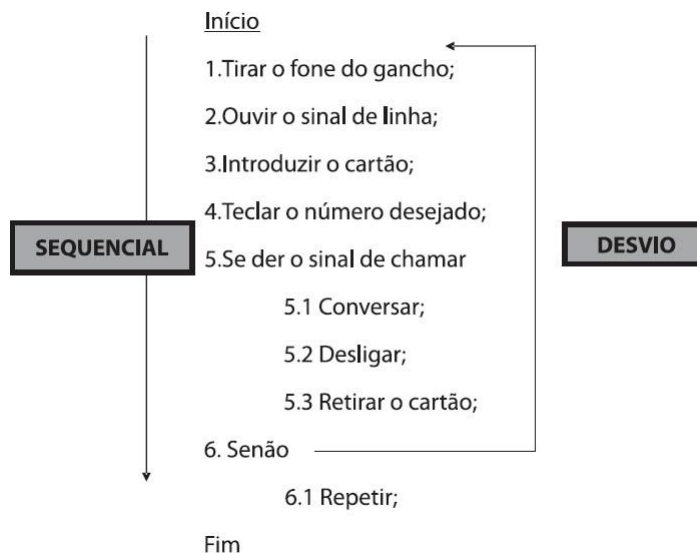
Criando o Primeiro Algoritmo.

Problema: Utilizar um telefone público.

O que você achou? Certamente, em sua vida, você já passou para utilizar um telefone. Você executou um conjunto de passos lógicos para conseguir o que você queria: utilizar o



executou esses
conjunto de
telefone.



Bem, olhando e revisando os exemplos que acabamos de citar, podemos perceber que tudo não passa de uma seqüência lógica ordenada de forma a se alcançar um objetivo específico, com prazos para começar (início) e terminar (fim).

Será que você conseguiria atender um telefone sem antes tirá-lo do gancho? Você pode chupar uma bala antes de retirar o papel?

Se você voltar à definição de algoritmos, vai perceber que colocamos que a solução não pode ser redundante e nem imprecisa, ou seja, deve ser clara e objetiva de forma que qualquer um possa entender e executá-la corretamente.

Voltamos para o caso da utilização do telefone público. Você consegue definir claramente todas as ações?

Um bom algoritmo deve mostrar uma solução precisa para o problema em questão.

Vamos imaginar o seguinte: o telefone não deu sinal de chamada na primeira vez que você ligou. O que você faz? Repete as ações conforme o algoritmo: tirar telefone do gancho, ouvir sinal de linha, introduzir o cartão etc.

E se novamente você não obter sinal de linha? Deve repetir novamente. Pergunto: até quando?

Percebeu que há um fato mal resolvido? Não sabemos até quando repetir as ações para obter o sinal de linha. Um bom algoritmo, ou pelo menos um melhor do que apresentei no exemplo, devem se prevenir problemas como esses. Uma solução simples, por exemplo, seria tentar uma ligação até cinco vezes. Caso não consiga, desista e tente outro dia ou em outro telefone. Agora sim, nossa sequência lógica esta finita (apresenta um fim).

Sempre vamos alcançar o fim, qualquer que seja o resultado:

Sucesso (conseguimos fazer a ligação).

Ou fracasso (não conseguimos fazer a ligação depois de tentativas, por exemplo).

Exercícios

1. Crie uma sequência lógica para tomar banho.

2. Monte um algoritmo com ações para trocar o pneu de um carro.

Você realmente conseguiu trocar o pneu do seu carro?

Para lembrar!

1. A ordem lógica da execução das tarefas é importante.
2. Todo algoritmo tem início e fim.
3. Um algoritmo tem que ser completo.
4. Um algoritmo deve ter um alto índice de detalhamento.
5. Cada tarefa ou etapa é chamada de instrução.

Para que isso me serve? Meu objetivo é aprender programação e não ficar pensando nas atividades lógicas ou mesmo ilógicas que executo todos os dias.

Se fossemos fazer analogias, o algoritmo está para a fundação de uma casa, assim como a programação propriamente dita está para construção das paredes da casa.

Até podemos levantar uma casa sem fundação, mas os problemas que virão no futuro.

No mesmo raciocínio até podemos construir programas sem algoritmos, mas, pode ter certeza, quanto maior for o seu programa, maiores serão suas dores de cabeça, principalmente se tiver que fazer manutenções e atualizações no programa. Você deixará de ser eficiente e perderá mais tempo corrigindo problemas do que criando inovações.



paredes

imagine

eficaz e

Se você for construir uma página na internet, mesmo que bastante simples, o algoritmo continua sendo importante.

Imagine, por exemplo, se você, usuário de uma página qualquer, selecionar um link e ele se mostrar indisponível ou levar você a outro lugar não correspondente ao que você selecionou? Isso pode ser um erro de lógica de programação de quem projetou a página, e isso pode irritar profundamente o usuário que acessou a página, no exemplo, você.

Muitas vezes esse é o final de uma carreira promissora, principalmente para iniciantes na área de programação. Cansado de fazer tantas correções e não evoluir no programa, o aluno acaba por desistir porque ele deixa de ser criativo e passa a ser corretivo.

Por isso, se fazendo uma boa fundação para uma casa, seus problemas estruturais serão diminuídos. Então, não perca tempo, faça um bom algoritmo e seu programa se tornará melhor.

A resolução de qualquer problema para o mundo da informática (e os sistemas web são processos informáticos) estrutura-se segundo três fases essenciais:

Análise do problema.

Desenho do algoritmo.

Codificação.



Figura 3.1 – Passos para desenvolvimento de programas computacionais

A melhor codificação será alcançada quanto melhor for o algoritmo que representa seu problema.

Se você passar direto do problema para a codificação, sem a solução em forma de algoritmos, certamente estará criando um outro problema e, nesse caso, você terá dois problemas para resolver: o primeiro e o próprio problema inicial que você gostaria de estar resolvendo, o segundo é o problema criado por você mesmo na codificação em uma linguagem de programação, onde você gastará horas tentando corrigir e fazer com que o mesmo resolva o seu problema inicial.

Formas de representação de um algoritmo

Até agora os algoritmos foram representados por palavras, porém há outras maneiras de representá-los. Você vai conhecer duas delas:

Pseudocódigo e fluxogramas.

Pseudocódigo

O nome Pseudocódigo é uma alusão a posterior implementação em uma linguagem de programação, ou seja, quando formos programar em uma linguagem, por exemplo, Visual Basic ou Java, estará gerando código em Visual Basic ou Java. Por isso os algoritmos são independentes das Linguagens de programação.

Ao contrário de uma linguagem de programação, não existe um formalismo rígido de como deve ser escrito o algoritmo. O fato importante é que um algoritmo deve ser fácil de interpretar e fácil de codificar. Ou seja, ele deve ser o intermediário entre a linguagem falada e a linguagem de programação.

A representação de um algoritmo em Pseudocódigo é a seguinte:

```
Início  
< comando 1 >  
< comando 2 >  
.....  
< comando n >  
Fim
```

Início: Como o próprio nome já diz, indica o início do algoritmo.

Fim: Como o próprio nome já diz, indica o fim do algoritmo.

Os comandos são as ações que são executadas como forma de se atingir o objetivo final, ou seja, solucionar um problema qualquer.

Estamos colocando as palavras Início/Fim destacadas, como forma de expressar os comandos do Pseudocódigo e diferenciá-los das demais instruções. Chamamos essas palavras de reservadas, como fazendo parte da linguagem em pseudocódigo.

Na linguagem Pseudocódigo, há um conjunto de palavras que expressam ações que determinam o que será feito. Por exemplo, escreva, leia, enquanto etc., e são reservadas no sentido de que elas são únicas e não podem ser refinadas, ou seja, detalhadas. Fim e fim e ponto final. Todas as palavras reservadas da linguagem serão sublinhadas.

Fluxogramas

É uma representação gráfica de algoritmos onde formas geométricas diferentes implicam ações (instruções, comandos) distintos. Tal propriedade facilita o entendimento das idéias contidas nos algoritmos e justifica sua popularidade.

Esta forma é aproximadamente intermediária à descrição narrativa e ao pseudocódigo (subitem seguinte), pois é menos imprecisa que a primeira e, no entanto, não se preocupa com detalhes de implementação do programa, como o tipo das variáveis usadas.

Nota-se que os fluxogramas convencionais preocupam-se com detalhes de nível físico da implementação do algoritmo. Por exemplo, figuras geométricas diferentes são adotadas para representar operações de saída de dados realizadas em dispositivos distintos, como uma fita magnética ou um monitor de vídeo. Como esta apostila não está interessada em detalhes físicos da implementação, mas tão somente com o nível lógico das instruções do algoritmo, será adotada a notação simplificada da Figura 2.1 para os fluxogramas. De qualquer modo, o Apêndice A contém uma tabela com os símbolos mais comuns nos fluxogramas convencionais.

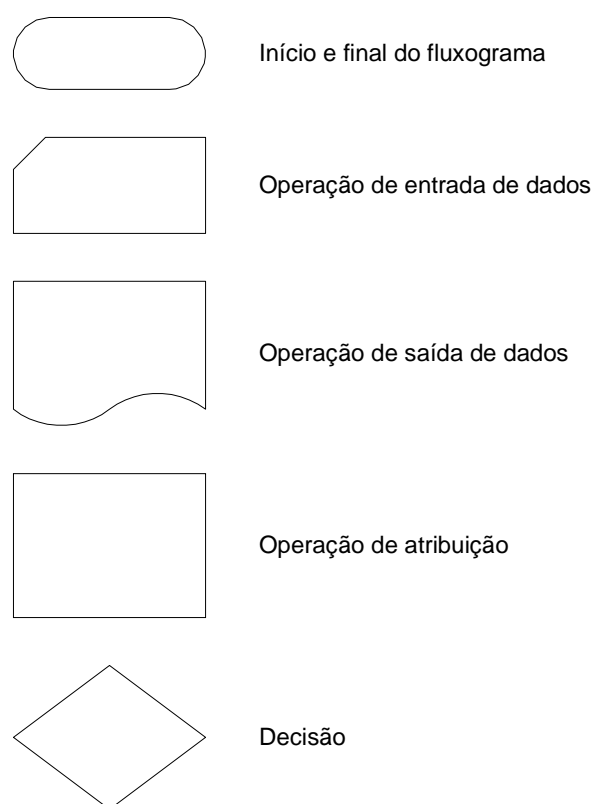


Figura 2.1 Principais formas geométricas usadas em fluxogramas.

De modo geral, um fluxograma se resume a um único símbolo inicial por onde a execução do algoritmo começa, e um ou mais símbolos finais, que são pontos onde a execução do algoritmo se encerra. Partindo do símbolo inicial, há sempre um único caminho orientado a ser seguido, representando a existência de uma única seqüência de execução das instruções. Isto pode ser melhor visualizado pelo fato de que, apesar de vários caminhos poderem convergir para uma mesma figura do diagrama, há sempre um único caminho saindo desta. Exceções a esta regra são os símbolos finais, dos quais não há nenhum fluxo saindo, e os símbolos de decisão, de onde pode haver mais de um caminho de saída (usualmente dois caminhos), representando uma bifurcação no fluxo.

A Figura 2.2 mostra a representação do algoritmo de cálculo da média de um aluno sob a forma de um fluxograma.

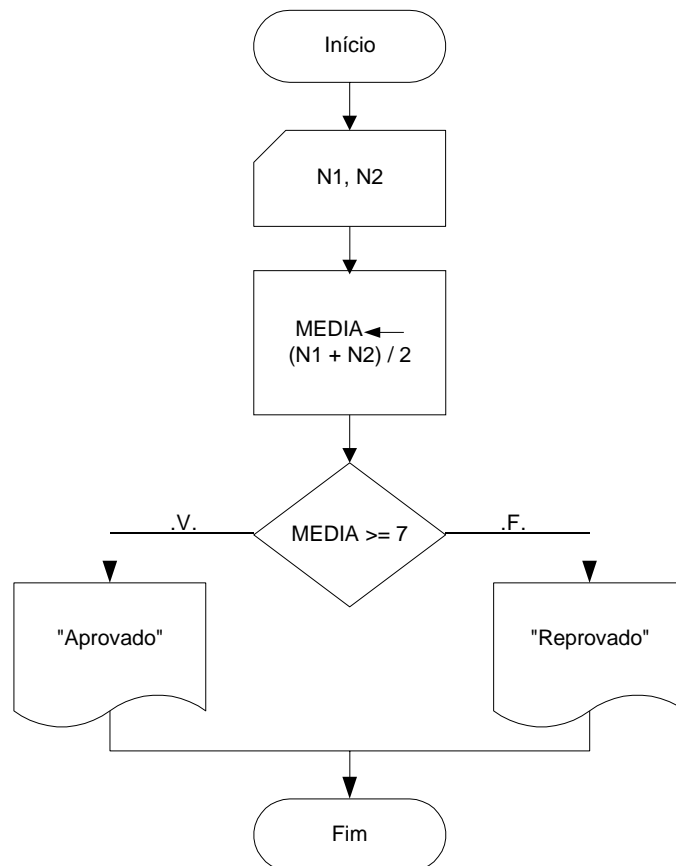


Figura 2.2 Exemplo de um fluxograma convencional.

Regras para a construção do algoritmo

Para escrever um algoritmo precisamos descrever a sequência de instruções, de maneira simples e objetiva. Para isso utilizaremos algumas técnicas:

Usar somente um verbo por frase

Imaginar que você está desenvolvendo um algoritmo para pessoas que não trabalham com informática;

Usar frases curtas e simples;

Ser objetivo;

Procurar usar palavras que não tenham sentido dúbio.

Fases para a construção do algoritmo

No início da unidade você viu que algoritmo é uma seqüência lógica de instruções que podem ser executadas. É importante ressaltar que qualquer tarefa que siga determinado padrão pode ser descrita por um algoritmo, como por exemplo:

COMO FAZER ARROZ DOCE

ou então

CALCULAR O SALDO FINANCEIRO DE UM ESTOQUE

Entretanto, ao montar um algoritmo, precisamos primeiro dividir o problema apresentado em três fases fundamentais:

ENTRADA DE DADOS → PROCESSAMENTO → SAÍDA DE DADOS

Onde temos:

ENTRADA: são os dados de entrada do algoritmo.

PROCESSAMENTO: são os procedimentos utilizados para chegar ao resultado final.

SAÍDA: são os dados já processados.

Analogia com o homem:

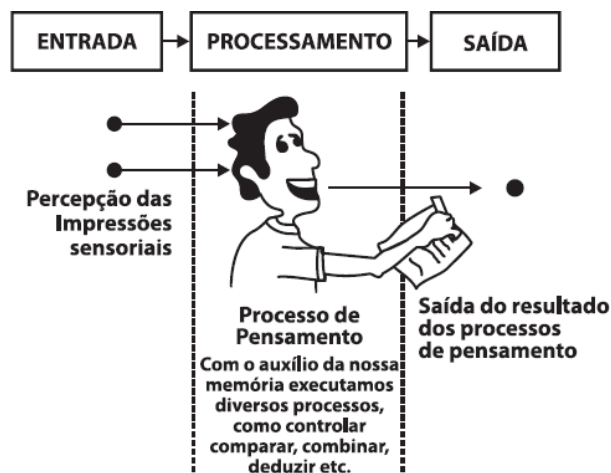


Figura 3.3 – Representação humana de execução de um algoritmo

Exemplo de algoritmo

Imagine o seguinte problema: calcular a media final dos alunos da 3a serie. Os alunos realizaram quatro provas: P1, P2, P3 e P4.

Onde:

$$\text{Media Final} = (P1 + P2 + P3 + P4) / 4$$

Para montar o algoritmo proposto, faremos três perguntas:

a) Quais são os dados de entrada?

R: Os dados de entrada são P1, P2, P3 e P4.

b) Qual será o processamento a ser utilizado?

R: O procedimento será somar todos os dados de entrada e dividi-los por 4 (quatro) $(P1 + P2 + P3 + P4) / 4$.

c) Quais serão os dados de saída?

R: O dado de saída será a média final.

Algoritmo Cálculo da Média

Início

Receba a nota da prova1

Receba a nota de prova2

Receba a nota de prova3

Receba a nota da prova4

Some todas as notas e divida o resultado por 4

Mostre o resultado da divisão

Fim

Para certificar que o algoritmo esta certo, nada melhor do que testá-lo, linha por linha, por meio de um exemplo. Para calcular a média final do algoritmo anterior, façamos:

Dados de Entrada:

Nota da prova1 = 8
Nota de prova2 = 10
Nota de prova3 = 10
Nota da prova 4 = 8

Processamento:

Some todas as notas e divida o resultado por 4:
 $(8 + 10 + 10 + 8) / 4$

Resultado de Saída:

Mostre o resultado da divisão: 9

a maioria das vezes, no entanto, algoritmos se tornam grandes demais e já não podemos testá-lo de uma só vez.

A regra básica é simplicidade e objetividade nas soluções encontradas.

Nesta unidade você aprendeu que algoritmo é uma seqüência de instruções ordenadas de forma lógica para a resolução de uma determinada tarefa ou problema. Este é um conceito central da programação e da Ciência da Computação, isto é, programar é basicamente construir algoritmos. É uma receita para um processo computacional e consiste de uma série de operações primitivas, interconectadas devidamente, sobre um conjunto de objetos.

Desta forma, a resolução de qualquer problema por processos informáticos, pode ser estruturada segundo três fases essenciais:

análise do problema; desenho do algoritmo e codificação.

Quanto melhor a etapa de desenho do algoritmo, melhor será a fase de codificação do programa.

Por fim, você viu duas formas de representação de um algoritmo.

A primeira é o pseudocódigo, que é um modelo de representação que utiliza comandos da língua portuguesa. A segunda forma de representação é o fluxograma, um modelo que utiliza figuras para representar o fluxo de dados e os comandos do algoritmo.

EXERCÍCIOS

1. Identifique, no algoritmo a seguir, quais são os dados de entrada, processamento e saída.

Receba código da peça

Receba valor da peça

Receba quantidade de peças

Calcule o valor total da peça (Quantidade * Valor da peça)

Mostre o código da peça e seu valor total

2. Faça um algoritmo para “calcular o estoque médio de uma peça”, sendo que $\text{ESTOQUE MÉDIO} = (\text{QUANTIDADE MÍNIMA} + \text{QUANTIDADE MÁXIMA}) / 2$.

Lembre-se que um algoritmo é formalmente uma seqüência finita de passos que levam a execução de uma tarefa e que, para montá-lo, você deve fazer as três perguntas: quais são os dados de entrada?; Qual será o processamento a ser utilizado?; Quais serão os dados de saída?

3. Teste o algoritmo anterior com dados definidos por você.

4 - CONSTANTES, VARIÁVEIS E TIPOS DE DADOS

Introdução

Em todos os momentos de nossa vida, precisamos identificar objetos de forma a poder distinguir, agir, combinar, separar, atribuir funções etc. Vamos pegar um exemplo do nosso cotidiano: precisamos identificar o nome de uma pessoa para chamá-la. O nome da pessoa é, portanto, um identificador.

Variáveis e constantes são os elementos básicos (identificadores) que um programa manipula. Uma variável é um espaço reservado na memória do computador para armazenar um tipo de dado determinado.

Como identificadores variáveis devem receber nomes para poderem ser chamadas e modificadas quando necessário.

Quando criamos uma variável, criamos um espaço na memória (endereço) que serve para armazenar valores.

Para não precisarmos manipular diretamente aquele endereço, basta identificarmos a variável pelo nome, e o resto o computador faz, ou seja, associa o nome da variável ao endereço de memória respectivo daquela variável.

Fazendo analogias, é muito mais fácil chamar pelo nome de uma pessoa do que pelo seu CPF.

Constantes e variáveis

Na maioria das vezes, precisamos armazenar dados para manipulá-los. Por exemplo, em um escritório de contabilidade, armazenamos os dados de clientes em fichários.

Sempre que desejar acessar os dados de um determinado cliente, vamos até as gavetas dos armários onde os fichários estão colocados. Podemos ler os dados (nomes, data de nascimento, endereço, telefone etc.), alterá-los ou até mesmo remove-los.



Assim é um programa de computador. Precisamos também armazenar valores para poder manipulá-los. Dessa forma, criamos variáveis para armazenar esses valores.

Um exemplo de onde precisamos armazenar dados em variáveis é o de um site de uma livraria. Nele podemos nos cadastrar para poder comprar livros pela internet, receber mensagens de e-mail da livraria, descontos etc. Precisamos preencher um cadastro com nosso nome, endereço postal e de e-mail, telefone de contato, entre outros dados, para nos tornarmos cliente da loja, certo?

Antes de serem armazenados na base de dados ou banco de dados da livraria on-line, os dados são armazenados em variáveis do programa de computador que é responsável pela página da livraria na internet. Após você clicar no botão enviar, os dados são transferidos das variáveis para o banco de dados da livraria.

Apenas se preocupe em saber que, após digitados em um site, os dados precisam ser guardados temporariamente antes de serem registrados definitivamente. A esses lugares temporários damos o nome de variáveis.

Vamos às definições iniciais?

Constantes

Constante é um determinado valor fixo que não se modifica ao longo do tempo, durante a execução de um programa.

Por exemplo, o número 2. Ele sempre vai corresponder a duas unidades durante todo o programa, desde o início até o fim do algoritmo.

As constantes, na verdade, são valores inseridos na memória do computador, ou seja, nas variáveis definidas pelo usuário.

Conforme o seu tipo, a constante é classificada como sendo numérica, lógica e literal.

Exemplos de constantes.

Numérica:

10 27 - 126 2.34 - 568.78

Literal: conjunto de caracteres (letras, dígitos ou símbolos especiais: &, \$ etc.) que devem ser colocados entre duplas aspas (" ").

"WEB" "Paulo" "23/10/2004" "X25tW2"

Lógica:

Falso Verdadeiro

Variáveis

Variável é a representação simbólica dos elementos de memória de um computador.

Cada variável corresponde a uma posição de memória, cujo conteúdo pode se alterado ao longo do tempo durante a execução de um programa. Embora uma variável possa assumir diferentes valores, ela só pode armazenar um valor a cada instante.

Importante

Sempre que criarmos uma variável, estamos criando um espaço na memória do computador para armazenar dados. É um endereço no qual o computador se referencia para manipular os dados.

Um endereço de memória são números hexadecimais e complicados de serem manipulados diretamente (veja na unidade 1). Por isso, criamos identificadores para as variáveis e passamos a chamá-las apenas pelo nome. Isso facilita bastante porque, conforme a introdução desta unidade, é mais fácil chamar uma pessoa pelo nome do que pelo seu CPF, você concorda?

As variáveis só podem armazenar constantes (valores) de um mesmo tipo, ou seja, para armazenar constantes do tipo numérico precisamos ter variáveis do tipo numérico; para armazenar constantes literais, precisamos de variáveis do tipo literal, e para armazenar constantes lógicas precisamos de variáveis do tipo lógica.

Quando criamos um novo fichário para um escritório de contabilidade, precisamos respeitar os campos nele contidos, ou seja, no campo nome, colocamos o nome do cliente (é um literal), no campo data de nascimento colocamos a data de nascimento (é um literal), no campo salário, colocamos o valor do salário atualizado do cliente (é um número), e assim sucessivamente. Por analogia, podemos dizer que os textos que mostram as informações de preenchimento no fichário são os nomes das variáveis e os locais em branco para preenchimentos são as constantes que estaremos colocando (nome, data de nascimento, salário etc.), tudo devidamente em seus respectivos lugares.

Vale-se para o nosso fichário, também vale para os programas de computador. E como se diz cada macaco no seu galho.

Tipos de variáveis

Numérico

Variável específica para armazenamento de números, que posteriormente poderão ser utilizados para cálculos.

Podem ser ainda classificadas como Inteiras ou Reais. As variáveis do tipo inteiro são para armazenamento de números inteiros e as Reais são para o armazenamento de números que possuam casas decimais.

A idade de uma pessoa é um número que deve ser armazenado em uma variável numérica; o peso em quilos de um animal de estimação deve ser armazenado também em uma variável numérica.

Não podemos armazenar qualquer outro valor senão número, seja ele inteiro ou real.

Alfanumérico ou literais

Específicas para dados que contenham letras, letras e números ou apenas números. Pode em determinados momentos conter somente dados numéricos ou somente literais.

Se usado somente para armazenamento de números, não poderá ser utilizada para operações matemáticas.

O nome de um cliente é um literal que deve ser armazenado em uma variável literal; a data de nascimento, o endereço residencial também são constantes literais que devem ser armazenadas em variáveis literais.

Lógico

Variáveis que armazenam somente dados lógicos que podem ser Verdadeiro ou Falso.

Queremos saber se a pessoa cadastrada tem (Verdadeiro) ou não tem (Falso) carteira de habilitação. Devemos armazenar o valor em uma variável lógica.

Porém, como diferenciar uma variável literal de uma numérica ou de uma lógica?

Para solucionar esse problema, devemos declarar as variáveis.

Declarar uma variável significa definir o nome da variável, e que tipo de constante será armazenado nessa variável.

Dessa forma, o primeiro passo é criar identificadores (nomes) para as variáveis.

Por exemplo, quero criar uma variável para armazenar o nome de um cliente e outra para armazenar a idade.

Em pseudocódigo, simplesmente coloque os nomes das variáveis, seguido de dois pontos (:) e o tipo da mesma, nessa ordem.

```
Início  
Nome: literal  
Idade: numérico  
Fim
```

Foi escolhido o tipo literal para a variável nome porque o conteúdo da variável é composto por letras, e escolhido o tipo numérico para a variável idade porque o conteúdo da variável deve ser um valor numérico inteiro.

Portanto, para declarar uma variável em pseudocódigo, a estrutura do comando é a seguinte:

<identificador>: <tipo de variável>

O símbolo < > significa que são informações preenchidas pelo próprio programador.

<identificador> mostra que você deve especificar um nome qualquer para identificar a variável; <tipo de variável> mostra que você deve especificar o tipo da variável, ou seja, que constante será armazenada na variável criada.

Veja mais um exemplo.

Para criar uma variável idade do tipo numérica, podemos fazer a seguinte declaração.

```
Início
idade: numérico
Fim
```

No exemplo anterior, perceba como a variável idade foi criada:

Idade: numérica. Aqui, <identificador> é idade e <tipo de variável> é numérico. Se houver a necessidade de criarmos várias variáveis do mesmo tipo, podemos simplesmente separar os nomes por vírgula. Veja o exemplo:

```
Início
IDADE, PESO: numérico
NOME, ENDEREÇO, FILIAÇÃO: literal
Fim
```

No exemplo anterior, tanto IDADE quanto PESO são variáveis numéricas; NOME, ENDEREÇO, FILIAÇÃO são variáveis literais.

Para o caso da representação em fluxogramas, não existem declarações de variáveis.

Como o identificador da variável e de responsabilidade do programador, ou seja, quem define o nome da variável e o programador, há algumas regras fundamentais que poderão lhe auxiliar na definição dos nomes. Essas regras são aplicadas tanto em lógica quanto em linguagens de programação.

1. Nomes de variáveis não podem conter espaços em branco.

Por exemplo, nome de aluno e numero de identidade são nomes de variáveis não permitidos.

Podemos declarar as mesmas variáveis como mostrado a seguir: nome<-de<-aluno, ou nomeDeAluno, por exemplo.

2. O nome de uma variável não pode ser inicializada por um numero.

Por exemplo, 1Nome, 45Idade são nomes de variáveis não permitidos.

Ou seja, o nome de uma variável sempre deve começar com uma letra. Depois se pode seguir uma outra letra ou numero qualquer.

Exemplos: Nome1, X25, teste<-54 etc.

3. Não podem ser utilizados caracteres especiais (c, ~, %, \$, -, \ etc.).

Por exemplo: coração, opção, nome-de-aluno.

4. Não pode ser uma palavra reservada.

Até o momento, identificamos as seguintes palavras reservadas: início, fim, literal, lógico, numérico (inteiro e real).

São palavras que servem para indicar o início e o fim do algoritmo, além de especificar os tipos das variáveis.

Mas você pode se perguntar: se são apenas pseudocódigos, por que as definições de variáveis não são livres, ou seja, por que, por exemplo, preciso criar uma variável chamada nomeDeAluno em vez de criar uma variável chamada nome-de-aluno ou nome de aluno apenas?

A resposta para esta questão é que as linguagens de programação não aceitam esses caracteres especiais para a criação das variáveis, mas apenas letras, letras seguidas de números e o underscore.

Mas o que o pseudocódigo tem a ver com isso?

Bem, quando criamos um pseudocódigo, criamos com objetivo de facilitar todo o desenvolvimento de um programa de computador.

Toda a solução lógica deve estar no pseudocódigo. Se bem feito, a linguagem de programação é apenas uma questão de digitação e não de raciocínio.

Seria de grande interesse que as variáveis criadas em Pseudocódigo se repetissem na linguagem de programação.

Por exemplo, uma variável `dataDeNascimento` em pseudocódigo teria o mesmo nome em uma linguagem de programação qualquer.

Caso contrário precisaria parar para pensar em um nome de variável caso a mesma estivesse definida `data de nascimento` em pseudocódigo.

Todo um padrão de nomes de variáveis podem ser criado em pseudocódigo.

Importante

Para algumas linguagens de programação, os nomes de variáveis são case sensitive, ou seja, uma variável identificado por `idade` é diferente de uma outra identificada por `Idade`, `iDade`, `IDADE`, por exemplo.

Dicas:

1. Mantenha os nomes das variáveis em pseudocódigo sempre com letras MAIÚSCULAS.

Isso é útil apenas para diferenciar dos comandos do pseudocódigo, principalmente para iniciantes. Se você desejar criar as variáveis com letras minúsculas, sintá-se à vontade. Apenas respeite o case sensitive.

Por exemplo: `Idade` é diferente de `idade`, que são diferentes de `IDADE` e assim por diante. Nos programas de computadores utilizando uma linguagem de programação qualquer, o padrão é criar nomes de variáveis em letras minúsculas.

2. Dê nome às variáveis que façam sentido para você.

Por exemplo, ao criar uma variável numérica `X`, você consegue identificar o que `X` vai armazenar? E se você criar uma variável chamada `IDADE`? Parece que deve armazenar a idade de um objeto ou pessoa, certo?

Então dê nome à variável de forma que seja fácil identificar a sua função.

Case sensitive: uma linguagem de programação é Case sensitive quando uma letra em caixa alta (maiúscula) tem significado diferente da mesma letra em caixa baixa (minúscula). Por exemplo, as variáveis `X` e `x` são duas variáveis diferentes.

EXERCÍCIOS

- 1) Construa um sistema de cadastro de alunos em um site de internet de uma escola, cujos dados são: nome, sexo, endereço, cidade, estado, CEP, telefone, data de nascimento, RG e grau de escolaridade. Utilize pseudocódigo para declarar as variáveis.

a) Primeiramente, identifique as constantes que serão utilizadas: nome, sexo, endereço, cidade, estado, CEP, telefone, data de nascimento, RG, grau de escolaridade.

b) Especifique os tipos de cada uma das constantes.

Variáveis literais (alfanuméricas): nome, sexo, endereço, cidade, estado, CEP, telefone, data de nascimento, RG.

Variáveis numéricas: grau de escolaridade. Exemplo: grau 1, 2 ou 3, representando nível fundamental, médio e superior respectivamente.

Variáveis lógicas: nenhuma.

c) Crie as variáveis para armazenar as constantes do problema.

Nome do aluno = NOME
Sexo = SEXO
Endereço = ENDEREÇO
Cidade = CIDADE
Estado = ESTADO
CEP = CEP
Telefone = TELEFONE
Data de Nascimento = DATA<-NASC
RG = RG
Grau de Escolaridade = GRAUESC

d) Para finalizar, coloque as definições no formato do pseudocódigo:

Início
NOME, SEXO, ENDEREÇO, CIDADE, ESTADO, CEP,
TELEFONE, DATA<-NASC, RG: literal.
GRAUESC: numérico
Fim

2) Verifique se os identificadores são válidos. Se não forem, explique por que.

nome-do-aluno
\$valor
idade/
data/nascimento
inicio
juros%
saldo

3) Seja um sistema de cadastro de um cliente em uma loja. Declare as variáveis desse cadastro.

Atribuindo valores às variáveis

Uma variável nunca é eternamente igual a um mesmo valor. Seu conteúdo pode ser alterado a qualquer momento.

A sintaxe desse comando (comando que atribui valor) é uma seta entre o identificador e o valor a ser armazenado.

1. Pseudocódigo

<identificador> ← <valor a ser armazenado>

2. Fluxograma

<identificador> ← <valor a ser armazenado>

Exemplos:

```
NOME ← "João Paulo" {João Paulo e uma constante literal}
{Por isso deve estar entre duplas aspas}
IDADE ← 25
```

Importante

O valor armazenado em uma variável deve ser do mesmo tipo da variável.

Nunca utilize o comando de atribuição para armazenar um valor em uma variável que não sejam compatíveis.

Por exemplo, IDADE <- "Pedro" (o nome Pedro não pode ser atribuído a uma variável numérica) ou NOME <- 45 (o número 45 não poder ser atribuído a uma variável literal).

Sintaxe: toda a linguagem de programação, incluindo aqui o pseudocódigo possui um conjunto de representações e/ou regras que são utilizadas para criar um programa.

Por meio dessas representações é que podemos estabelecer uma comunicação com o computador, fazendo com que ele execute o que nós queremos que ele faça. A esse conjunto de representações e/ou regras damos o nome de sintaxe da linguagem.

Exercícios: atribuindo valores às variáveis

- 1) Utilizando o exemplo de um sistema de informação para cadastro de um cliente:

```
Início
NOME, SEXO, ENDEREÇO, CIDADE, ESTADO, CEP, TELEFONE,
DATA<-NASC, RG: literal
GRAUESC: numérica
Fim
```

Podemos atribuir os valores às variáveis:

```
Início
{primeiro declaramos as variáveis, pois precisamos identificá-las e definir se são
literais ou numéricas}
NOME, SEXO, ENDEREÇO, CIDADE, ESTADO, CEP, TELEFONE,
DATA<-NASC, RG: literal
GRAUESC: numérica
{Agora, iremos atribuir valores às variáveis já criadas}
NOME ← "Paulo Martins"
SEXO ← "Masculino"
ENDEREÇO ← "Avenida Santa Catarina, número 20"
CIDADE ← "Florianópolis"
ESTADO ← "Santa Catarina"
CEP ← "88.035.002"
TELEFONE ← "(048)-235-66-00"
DATA<-NASC ← "12-10-1970"
RG ← "2.456.788" {número da carteira identidade}
GRAUESC ← 3 {representa o terceiro grau e não precisa colocar
entre aspas, pois é numérico}
Fim
```

EXERCÍCIOS:

- 2) Você é responsável pela elaboração de um sistema web de cadastro de livros de uma biblioteca. Declare as variáveis necessárias para um completo cadastramento e atribua valores as mesmas.

- 3) Crie um sistema de cadastro de alunos de uma escola. Declare as variáveis e atribua valores.

Expressões

Na solução da grande maioria dos problemas, é necessário que as variáveis tenham seus valores consultados ou alterados e, para isso, devemos definir um conjunto de operadores (básicos), sendo eles:

Operadores aritméticos

Operador aritmético Operação Descrição

Operador aritmético	Operação	Descrição
+	Adição	Faz a soma de dois números
-	Subtração	Faz a subtração de um número de outro
*	Multiplicação	Faz a multiplicação de um número pelo outro
/	Divisão	Faz a divisão de um número pelo outro
^	Potenciação	Faz a elevação de um número a uma potência

Exemplos de expressões aritmética com variáveis numéricas A, B, C.

A B C Operação Procedimento

A	B	C	Operação	Procedimento
-	-	-	$A \leftarrow 2$	Armazena o valor 2 na variável A
2	-	-	$B \leftarrow A + 3$	Soma o valor da variável A (que agora vale 2) com 3 e armazena o resultado em B (que passa então a valer 5)
2	5	-	$C \leftarrow B - A$	Subtrai do valor de B (que agora vale 5) do valor de A (que vale 2), armazenando o resultado (valor 3) em C
2	5	3	$B \leftarrow A * C$	Multiplica o valor de A (que vale 2) pelo valor de C (que vale 3) e armazena o resultado (valor 6) em B. Perceba que B valia 5 e agora vale 6
2	6	3	$C \leftarrow B^A$	Eleva o valor de B (que agora vale 6) ao valor de A (que vale 2) e armazena o resultado em C (que agora vale 36, que é o quadrado de 6)

A B C Operação Procedimento

A	B	C	Operação	Procedimento
2	6	36	$B \leftarrow C/A$	Divide o valor de C (que agora vale 36) pelo valor de A (que vale 2) e armazena o resultado em B. Portanto, B vale agora 18.
2	18	36	$A \leftarrow A + B$	Soma o valor de A (que vale 2) com o valor de B (que agora vale 18) e armazena na própria variável A (que muda de valor e agora vale 20).
20	18	36	-	Depois de todas as operações temos os valores de A, B e C conforme a tabela ao lado.

Simple, você não acha? Nada de novo até aqui. Porém, nosso mundo matemático não se resume apenas a somar, diminuir, multiplicar e dividir.

Temos algumas outras operações muito úteis que podem ser utilizadas para solucionar uma série de problemas matemáticos. São as chamadas funções aritméticas. Por exemplo, uma operação matemática para calcular a raiz quadrada de 4.

Algumas funções aritméticas básicas estão colocadas a seguir:

Função aritmética Sintaxe Descrição

Função aritmética	Sintaxe	Descrição
Divisão inteira	$DIV(x, y)$	Faz a divisão inteira da variável x pela variável y, ou seja, retorna apenas a parte inteira da divisão.
Resto da divisão	$RESTO(x, y)$	Retorna o resto da divisão da variável x pela variável y.
Valor absoluto	$ABS(x)$	Retorna o valor absoluto (valor positivo) de x.
Raiz quadrada	$RAIZ(x)$	Calcula a raiz quadrada do número x.

Exemplos de funções aritméticas com variáveis numéricas A, B, C.

A B C Operação Procedimento

A	B	C	Operação	Procedimento
-	-	-	$A \leftarrow 2$	Armazena o valor 2 na variável A
2	-		$B \leftarrow \text{DIV}(19, A)$	Realiza uma divisão inteira de 19 por A. Neste caso, realiza a operação $19/2 = 9$ (resultado inteiro). Desta forma, B passa a valer 9.
2	9	-	$C \leftarrow \text{RESTO}(B, A)$	Realiza a divisão de B (que vale 9) por A (que vale 2). O resto da divisão (valor 1) é armazenado em C.
2	9	1	$A \leftarrow \text{ABS}(-8)$	Armazena em A o valor absoluto de -8. Nesse caso, o valor numérico 8. Perceba que A valia 2 e agora vale 8.
8	9	1	$C \leftarrow \text{RAIZ}(B)$	Calcula a raiz quadrada de B (que vale 9) e armazena o resultado (valor 3) em C. Perceba que C valia 1 e agora vale 3.
8	9	3	-	Depois de todas as operações temos os valores de A, B e C, conforme a tabela ao lado.

Bastante úteis, também, são as operações que nos permitem fazer relações entre dois números, por exemplo, se um número é maior, menor ou igual a um outro número. São as chamadas operações relacionais.

Operadores relacionais

São utilizados para relacionar variáveis ou proposições, resultando num valor lógico (Verdadeiro ou Falso), sendo eles:

Operador relacional Operação Descrição

Operador relacional	Operação	Descrição
=	Igualdade	Verifica se dois valores são iguais. Se forem iguais, o teste de igualdade é verdadeiro, senão, é falso
< >	Diferença	Verifica se dois valores são diferentes. Se forem diferentes, o teste é verdadeiro, senão, é falso.

Operador relacional	Operação	Descrição
<	Menor que	Verifica se um número é menor do que o outro.
>	Maior que	Verifica se um número é maior do que o outro.
<=	Menor ou igual a	Verifica se um número é menor ou igual ao outro.
>=	Maior ou igual	Verifica se um número é maior ou igual ao outro.

Veja a seguir exemplos de expressões relacionais com as variáveis numéricas A e B e uma variável lógica X (variável lógica: Verdadeira ou Falsa).

A B X Operação Procedimento

A	B	X	Operação	Procedimento
-	-	-	$A \leftarrow 3$	Armazena na variável A o valor numérico 3.
3	-	-	$B \leftarrow 5$	Armazena na variável B o valor numérico 5.
3	5	-	$X \leftarrow (A = B)$	Armazena na variável X o valor lógico da expressão relacional que compara o valor de A com o valor de B. Colocamos entre parênteses apenas para facilitar a visualização da expressão. Como A é diferente de B, X armazenará falso. Observe a sequência das operações. Inicialmente é realizado o teste para verificar se A é igual a B. Como A vale 3 e B vale 5, sabemos que A é diferente de B. Portanto, o resultado da comparação é falso. Sendo falso, X recebe falso.
3	5	<u>falso</u>	$X \leftarrow (A < > B)$	Para esse caso, como o valor armazenado na variável A (que vale 3) é diferente do valor armazenado na variável B (que vale 5), temos que o valor lógico da expressão é verdadeiro. Sendo assim, a variável X recebe verdadeiro.

A B X Operação Procedimento

A	B	X	Operação	Procedimento
3	5	<u>verdadeiro</u>	$X \leftarrow (A < B)$	Armazena em X o valor lógico da expressão $A < B$. Como o valor de A é 3 e o valor de B é 5, a expressão é verdadeira. Sendo assim, a variável X assume o valor verdadeiro.
3	5	<u>verdadeiro</u>	-	Depois de todas as operações temos os valores de A, B e X, conforme a tabela ao lado.

Por fim, vamos aos operadores lógicos. São expressões que nos permitem verificar o resultado de uma operação lógica, ou seja, de operações entre duas ou mais variáveis lógicas.

O resultado é sempre um resultado lógico, ou seja, verdadeiro ou falso. Aqui é fundamental que você revise as definições e utilizações de tabela-verdade

Operadores lógicos

Operador lógico	Operação	Descrição
<u>E</u>	Intersecção	Operação de conjunção
<u>Ou</u>	União	Operação de disjunção
<u>Não</u>	Negação	Operação de negação

Veja, a seguir, exemplos de expressões lógicas com operadores relacionais utilizando as variáveis numéricas A e B (que armazenam números) e as variáveis lógicas X e Y (que armazenam verdadeiro ou falso).

A	B	X	Y	Operação	Procedimento
-	-	-	-	$A \leftarrow 15$	Armazena o valor numérico 15 na variável A.
15	-	-	-	$B \leftarrow 17$	Armazena o valor numérico 17 na variável B.

A	B	X	Y	Operação	Procedimento
15	17	<u>falso</u>	<u>falso</u>	$X \leftarrow \text{não } Y$	<p>É uma operação de negação (não). Verifique a tabela-verdade na unidade 2.</p> <p>A variável Y armazena <u>falso</u>. Como estamos negando o valor de Y, ou seja, estamos negando o valor <u>falso</u>, o resultado é <u>verdadeiro</u>. Portanto, o valor <u>verdadeiro</u> é armazenado na variável X.</p>
15	17	<u>verdadeiro</u>	<u>falso</u>	$X \leftarrow (\text{não } Y)$ $E (A \leq B)$	<p>Observe a operação de conjunção (operação <u>e</u>). Se observarmos a tabela-verdade de uma conjunção (unidade 2) vemos que o resultado de uma operação é verdadeira quando as duas variáveis são verdadeiras. Caso contrário, a operação é falsa. Vamos analisar as duas proposições antes e após o <u>E</u>. A variável Y tem valor <u>falso</u>. Negando Y através da operação <u>não</u> temos como resultado <u>verdadeiro</u>. A operação $A \leq B$ também é verdadeira, tendo em vista que A armazena 15 e B armazena 17. Sendo ambas as proposições verdadeiras, o resultado da operação <u>E</u> também é <u>verdadeiro</u>. Sendo assim, X armazena valor <u>verdadeiro</u>.</p>

A	B	X	Y	Operação	Procedimento
15	17	-	-	$X \leftarrow (A = B)$	A expressão $(A = B)$ é avaliada. Como A é diferente de B, temos que o resultado do teste é <u>falso</u> . Portanto, X armazena falso. Observação: colocamos entre parênteses apenas para facilitar a visualização da expressão.
15	17	<u>falso</u>	-	$Y \leftarrow (A < > B)$	A expressão $(A < > B)$ é avaliada. Como A é diferente de B, temos que o resultado do teste é verdadeiro. Portanto Y armazena <u>verdadeiro</u> .
15	17	<u>falso</u>	<u>verdadeiro</u>	$Y \leftarrow X \text{ ou } (A > B)$	Observe a operação de disjunção (operação <u>ou</u>). Se observarmos a tabela-verdade de uma disjunção (unidade 2) vemos que o resultado de uma operação é verdadeira quando uma ou duas variáveis são verdadeiras. Caso contrário, a operação é falsa. Observe inicialmente que X tem valor <u>falso</u> . Observe também que a variável A tem valor 15 e a variável B tem valor 17. Portanto, a expressão $A > B$ também é falsa. Sendo ambos falsos, o resultado da operação <u>ou</u> também é <u>falso</u> . Sendo assim, a variável Y armazena <u>falso</u> .

A	B	X	Y	Operação	Procedimento
15	17	<u>verdadeiro</u>	<u>falso</u>	-	Depois de todas as operações, temos os valores de A, B, X e Y, conforme a tabela ao lado.

Prioridade de operadores

Durante a execução de uma expressão que envolve vários operadores, e necessários a existência de prioridades, caso contrário poderemos obter valores que não representam o resultado esperado.

A maioria das linguagens de programação utiliza as seguintes prioridades de operadores:

- 1) Efetuar operações embutidas em parênteses "mais internos".
- 2) Efetuar funções.
- 3) Efetuar multiplicação e/ou divisão.
- 4) Efetuar adição e/ou subtração.
- 5) Operadores relacionais.
- 6) Operadores lógicos. Nos operadores lógicos, a prioridade mais alta é a negação, depois a operação e, e por fim a operação ou.

Veja agora exemplos de algoritmos mostrando todas as operações até agora estudadas e as seqüências de prioridades:

Sendo as variáveis do tipo numérico, qual o resultado de cada variável no final dos algoritmos A e B?

Algoritmo A

```

Início
A, B, C, D: numérico
D ← 9.5
B ← (RAIZ (ABS (D)))
C ← B ^ (RESTO ((D + B - 0.5), 3))
A ← (B * C) / 2 ^ 2
Fim
{mostrando os resultados da execução do algoritmo} D <- 9.5 {variável D assume valor 9.5}
B ← (RAIZ(ABS(9.5))) {módulo de 9.5 é 9.5}
B ← RAIZ(9.5) {raiz quadrada de 9.5 é 3.082207}
B ← 3.082207 {variável B assume valor 3.082207}
C ← 3.082207 ^ (RESTO ((9.5 + 3.082207 - 0.5), 3))
C ← 3.082207 ^ (RESTO (12.082207, 3)) {resto da divisão inteira à 12 dividido por 3 = resto 0}
C ← 3.082207 ^ 0
C ← 1 {variável C assume valor 1}
A ← (B * C) / 2 ^ 2
A ← (3.082207 * 1) / 2 ^ 2
A ← 3.082207 / 4
A ← 0.770552 {variável A assume valor <- 0.770552}

```

Sendo A, B, C variáveis numéricas e D, E variáveis lógicas, qual o resultado de cada variável no final do algoritmo?

```
Início
A, B, C: numérico
D, E: lógico
A ← 20
B ← (40 + A) / 3
C ← RAIZ (A + 80)
D ← (A >= B)
E ← (C = B)
Fim

{mostrando os resultados da execução do algoritmo}
A ← 20
B ← (40 + A) / 3
B ← (40 + 20) / 3
B ← 60 / 3
B ← 20
C ← RAIZ (A + 80)
C ← RAIZ (20 + 80)
C ← RAIZ (100)
C ← 10
D ← (A >= B)
D ← (20 >= 20)
D ← verdadeiro
E ← (C = B)
E ← (10 = 20)
E ← falso
```

Sendo A e B variáveis numéricas, e D e E variáveis lógicas, qual o resultado da variável E no final do algoritmo?

```
Início
A, B: numérico
D, E: lógico
A ← 20
B ← 10
D ← falso
E ← ((A >= B) E ( não D)) OU (A = B)
Fim

{mostrando os resultados da execução do algoritmo}
A ← 20
B ← 10
D ← falso
```

Resolvendo a expressão $E \leftarrow ((A \geq B) \text{ E } (\text{não } D)) \text{ OU } (A = B)$ por partes. Vamos aplicar as regras de precedência. Parênteses sempre maior prioridade.

Vamos resolver inicialmente tudo o que está dentro de parênteses, então. Observe que temos uma expressão antes do OU envolto por parênteses e uma após.

Antes do OU: $((A \geq B) \text{ E } (\text{não } D))$

Após o OU: $(A = B)$.

Resolvendo:

Antes do OU: $((A \geq B) \text{ E } (\text{não } D))$

Observe que temos uma instrução E. Portanto vamos separá-la também, uma antes e outra após o E.

Antes do E $(A \geq B)$. Sabendo que A tem valor 20 e B tem valor 10, o resultado dessa comparação é verdadeiro.

Após o E $(\text{não } D)$. Sabendo que D tem valor falso, negando D temos valor verdadeiro.

Avaliando então antes do OU temos como resultado:

(verdadeiro E verdadeiro). Observando a tabela verdade para a operação E, temos que o resultado é verdadeiro quando ambas as instruções são verdadeiras. Portanto, a expressão antes do OU tem como resultado lógico verdadeiro.

Após o OU: $(A = B)$

A instrução é de comparação. Podemos Verificar que o resultado é falso, tendo em vista que A é diferente de B.

Avaliando a expressão após o OU temos como resultado falso.

Sendo assim, temos antes do OU um valor verdadeiro e após o OU um valor falso. Pela tabela-verdade de uma operação de disjunção, temos que o resultado é verdadeiro. Acompanhe a execução passo a passo:

Realizando a operação ou final:

```
E ← ((A ≥ B) E (não D)) OU (A = B)
E ← ((verdade) E (verdade)) ou (falso)
E ← (verdade E verdade) ou (falso)
E ← (verdade) ou (falso)
E ← verdade ou falso
E ← verdade
```

Sendo A e B variáveis numéricas e D uma variável lógica, qual o resultado da variável D no final de cada algoritmo?

Algoritmo A

```
Início
A, B: numérico
D: lógico
A ← 20
B ← 10
D ← A + B <= 30
Fim
```

Solução: Se observarmos a relação de prioridade, a adição tem prioridade maior do que a operação relacional. Sendo assim, primeiro somamos A com B e depois comparando com o valor 30.

```
D ← 20 + 10 <= 30
D ← 30 <= 30
Como 30 é igual a 30, o resultado lógico da operação é verdadeiro
D ← verdadeiro
```

Algoritmo B

```
Início
A, B: numérico
D: lógico
A ← 20
B ← 10
D ← A*2 + B <= 30
Fim
```

Solução: Bastante simples. Basta seguirmos as regras de prioridade e realizar a multiplicação inicialmente, depois a soma e, por último, a operação relacional.

Sendo assim:

```
D ← 20*2 + 10 <= 30
D ← 40 + 10 <= 30
D ← 50 <= 30
Como 50 é maior do que 30, o resultado da operação é falso.
D ← falso
```

Qual o valor lógico de F no final do algoritmo:

```
Início
A, B: numérico
C, D, F: lógico
A ← 20
B ← 10
C ← verdadeiro
D ← verdadeiro
F ← A > B E C OU D
Fim
```

Solução: A expressão $F \leftarrow A > B \text{ E } C \text{ OU } D$ está bastante confusa. Talvez um parênteses aqui seja primordial para dar legibilidade à operação. Mesmo assim, vamos resolvê-la. Pelas regras de prioridades sabemos que as operações relacionais vêm antes das operações lógicas. Com isso, precisamos avaliar a expressão $A > B$ inicialmente. Como A tem valor 20 e B tem valor 10, o resultado de $A > B$ é verdadeiro.

Após, temos duas operações lógicas, um E e um OU.

O a operação E representa uma multiplicação entre as variáveis, lembra? (ver unidade 2). Sendo assim, resolvemo-la inicialmente. Como já sabemos que o resultado de $A > B$ é verdadeiro, vamos resolver a expressão verdadeiro E C. Como a variável C tem valor lógico verdadeiro, a expressão fica verdadeiro

E verdadeiro. Da tabela-verdade, o resultado é verdadeiro. Após, executamos a operação lógica OU. Como resultado temos, verdadeiro OU D.

Independente do valor da variável D, o resultado da expressão é verdadeiro, tendo em vista que, para uma operação de disjunção, basta uma expressão ou variável ter valor lógico verdadeiro para o resultado ser verdadeiro.

Veja agora como seria mais legível, com parênteses, a mesma expressão:

```
F ← ((A > B) E C) OU D
```

E se quiséssemos executar primeiramente o OU e depois o E? Como poderíamos fazer? Simples, basta colocar os parênteses isolando a operação OU.

Veja abaixo:

$F \leftarrow (A > B) \text{ E } (C \text{ OU } D)$ {aqui, tanto a operação relacional $A > B$ quanto à operação lógica OU são executadas antes da operação lógica E.

$F \leftarrow (\text{verdadeiro}) \text{ E } (\text{verdadeiro OU verdadeiro})$

$F \leftarrow (\text{verdadeiro}) \text{ E } (\text{verdadeiro})$

$F \leftarrow (\text{verdadeiro})$

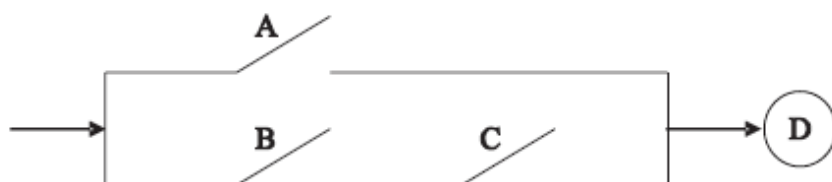
Lembre-se sempre: parênteses têm prioridade, ou seja, as proposições dentro dos parênteses têm que ser solucionadas primeiro.

Observe as seguintes proposições lógicas:

$D \leftarrow A \text{ OU } B \text{ E } C$

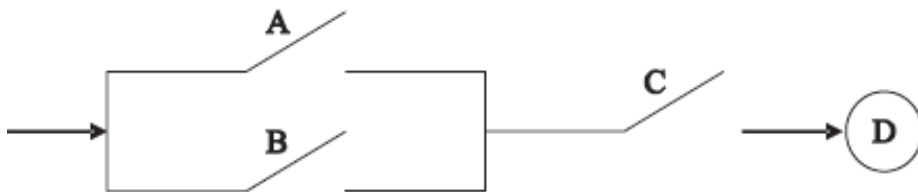
$F \leftarrow (A \text{ OU } B) \text{ E } C$

Vamos montar um circuito para ambas as proposições:



Para fazer com que D seja verdadeiro, devemos ou fechar o contato A ou fechar os contatos B e C. Caso contrario D será falso.

Veja agora o segundo circuito:



Podemos perceber que para que a proposição D seja verdadeira, A e C ou B e C devem ser verdadeiras, ou seja, devemos fechar os contatos A e C ou B e C.

Percebeu a diferença quando colocamos parênteses? Muito cuidado com isso, então. Em proposições mais complicadas o erro pode ser fatal no seu algoritmo. Por isso, ao elaborar proposições, na dúvida, monte um circuito, mostre a tabela-verdade e teste as condições. Certifique-se do que você está implementando e o que você realmente quer como resultado de saída.

O programador tem plena liberdade para incluir novas variáveis, operadores ou funções para adaptar o algoritmo as suas necessidades, lembrando sempre, de que, estes devem ser compatíveis com a linguagem de programação a ser utilizada.

Síntese

Nesta unidade vimos os conceitos iniciais de lógica de programação.

São os conceitos de constantes e variáveis. Definimos uma constante como sendo um determinado valor fixo que não se modifica ao longo do tempo, durante a execução de um

programa, e variável uma representação simbólica dos elementos de memória de um computador. Cada variável corresponde a uma posição de memória, cujo conteúdo pode ser alterado ao longo do tempo, durante a execução de um programa. As variáveis são utilizadas para armazenar as constantes de um programa e podem ser classificadas como sendo do tipo numérico, literal ou lógico. Cada variável declarada deverá ter seu tipo pré-definido e poderá armazenar constantes somente daquele tipo. Por exemplo, uma variável numérica somente poderá armazenar números.

Sempre que precisarmos utilizar uma variável precisamos declará-la. A sintaxe em Pseudocódigo é a seguinte:

<identificador>: <tipo de variável>

Identificamos o nome da variável e o seu tipo correspondente.

É importante observar as regras estabelecidas para os nomes das variáveis: não é possível nome de variáveis com acentuação, com espaços em branco, caracteres especiais como c, ~, -, etc. Os nomes devem começar necessariamente com uma letra. Após a primeira letra podemos ter outras letras e/ou números: por exemplo, TESTE, TESTE27, X25 etc. Para cada variável criada, devemos observar o case sensitive. Exemplo disso são as três variáveis TESTE, teste e Teste. São três variáveis diferentes e localizadas em posições de memória diferentes.

As variáveis são criadas para atribuímos valores a elas, ou seja, armazenarmos valores ou constantes. Para atribuir valores as variáveis,

vimos à sintaxe em pseudocódigo:

<identificador> ← <valor a ser armazenado>

O símbolo ← indica que estamos armazenando um valor (numérico, literal ou lógico) na variável identificada pelo identificador.

Por fim, vimos às expressões utilizadas para operações aritméticas (adição, subtração, multiplicação, divisão e potenciação), e para funções aritméticas (divisão inteira, cálculo do resto da divisão, valor absoluto, raiz quadrada); expressões relacionais (igualdade, diferença, maior que, menor que, maior ou igual a, menor ou igual a), e as expressões lógicas (e, ou, não).

Para operações que envolvam várias expressões (aritméticas, relacionais e lógicas) devemos observar a questão de prioridades, ou seja, quem será executado primeiro.

A tabela abaixo mostra essas prioridades:

Prioridade	Operadores/Operações
1	Aritméticas e literais
2	Relacionais
3	<u>não</u>
4	<u>e</u>
5	<u>ou</u>

É importante lembrar que o parêntese tem prioridade máxima, ou seja, expressões dentro de parênteses são solucionadas primeiras.

EXERCÍCIOS::

1. Observe a lista de variáveis abaixo com seus respectivos valores.

A partir desta relação, analise as expressões lógicas, retornando o resultado verdadeiro ou falso.

Lista de variáveis:

```
FRUTA ← 'laranja'
SALARIO ← 1000
HOBBY ← 'leitura'
IDADE ← 23
COR ← 'amarelo'
EHAPOSENTADO ← Falso
VALOR1 ← 10
VALOR2 ← 5
VALOR3 ← 0.5
```

- a) (FRUTA = "laranja") E (VALOR1 / VALOR2 > VALOR3) =
- b) (FRUTA="laranja") E (VALOR1 / VALOR2 < VALOR3) =
- c) (IDADE > (VALOR1 * VALOR2)) OU (NAO FRUTA= "laranja") =
- d) EHAPOSENTADO E (IDADE > 20) =
- e) (NAO EHAPOSENTADO) E (COR = "amarelo") E (RESTO(VALOR1, VALOR2) < 2) =
- f) não (IDADE > (110 DIV 15)) E (SALARIO = (10 ^ 3)) =

5 – PROCESSAMENTO SEQUENCIAL E CONDICIONAL

Introdução

Na elaboração de algoritmos, na maioria das vezes, estamos sempre incluindo comandos de entrada e/ou saída. Ou seja, alimentamos o nosso programa com entrada de dados, executamos o programa e colhemos (analisamos) os resultados de saída.

Fazendo analogias, como se o nosso algoritmo fosse uma receita de um delicioso bolo de chocolate, então os ingredientes como o chocolate, a manteiga, o açúcar etc., são as entradas. A atividade de fazer o bolo seguindo a receita e a execução do próprio algoritmo.

O resultado final é um bolo de chocolate decorado com frutas (saída).

A idéia desta unidade é introduzir comandos de entrada e saída aos algoritmos estudados até agora, completando com estudos de programação seqüencial e condicional.

Para isso, certifique-se de que você está sem dúvidas em relação aos conteúdos discutidos anteriormente, pois o que você estudou nessas unidades será fundamental neste momento.

No final desta unidade, você terá condições de elaborar algoritmos de programação seqüencial e condicional com comandos de entrada e saída.

Processamento seqüencial: comandos de entrada e saída

Foi anteriormente discutido que um computador pode receber informações através do teclado, toque na tela etc., e pode enviar essas e/ou outras informações para o monitor, imprimir em uma impressora etc. Dizemos que o teclado, toque de tela etc., são dispositivos de entrada de dados, e o monitor, a impressora etc., configuram como dispositivo de saída.

Por exemplo, você insere um texto digitando pelo teclado e o que você digitou é mostrado no monitor do computador.

Ao preparar um bolo que você faz uma vez por ano, por exemplo, um bolo de natal, dependendo de sua memória, você irá pegar a receita em seu livro de receitas. Na receita deve estar especificado os ingredientes e a quantidade de cada um deles. Esses são os dados de entrada da receita. O modo de preparar é o algoritmo ou processamento do algoritmo. Perceba que você faz toda a execução passo a passo, sem infringir – esperam os convidados – a receita. Essa execução passo a passo (linha a linha do seu caderno de receitas) é o que chamamos de execução seqüencial.

Bem, como resultado, se seguir a receita corretamente, você terá um bolo de natal. Esse resultado de saída da nossa receita, ou seja, do algoritmo.

Podemos assim dizer que, temos um processamento seqüencial quando temos um instruções (linhas) em que cada comando ou (em cada linha) é executado um após o outro, nenhum tipo de desvio no fluxo da lógica descrita.

As linguagens de programação tem se baseado muito nesses processamentos seqüenciais. Contudo, muitas vezes, precisamos de execuções que não são necessariamente seqüenciais.



é o
nosso

conjunto de
instrução
sem haver

Imagine que você é proprietário de uma confeitaria. Agora você tem que produzir não somente um, mas vários tipos de bolos a pedido dos clientes.

Você tem uma seqüência não definida, muito embora a receita de cada um deles possa ser seqüencial.

Como administrar essa situação? Quais são as seqüências de bolos que você vai fazer em um determinado dia?

Técnicas modernas podem representar esse fato com precisão. São as Linguagens de Orientação a Objetos.

Os comandos seqüenciais são três:

Leia

Escreva

Comando de Atribuição (←)

Comando Leia

O comando Leia (comando de entrada de dados) é utilizado para a leitura de qualquer dispositivo de entrada, como: teclado, disquete, scanner etc.

Imaginemos o seguinte: você deseja elaborar um sistema de cadastro dos alunos de uma escola. Vamos supor que, para nosso exemplo, basta saber o nome do aluno e seu código de matrícula. Construindo nosso algoritmo e cadastrando o aluno "Paulo" com o código de matrícula = 9000,

Temos:

```
Início
{declaração de variáveis}
NOME: literal
MATRICULA: numérico
{comandos de atribuição, onde um valor é
atribuído à variável}
NOME = "Paulo" {lembre-se: o literal usa aspas}
MATRICULA = 9000 {lembre-se que valor
numérico não usa aspas}
Fim
```

Você pode estar se perguntando: e se eu quiser cadastrar um outro nome que não seja Paulo, como posso fazer?

Você, na verdade, quer é especificar o nome do aluno (dado de entrada) e não um que esteja já predeterminado.

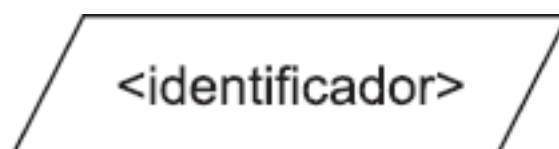
Para isso, utilize o comando Leia para a entrada de dados.

Em pseudocódigo, o comando Leia tem a seguinte sintaxe:

Leia <identificador> {onde "identificador" é o nome da variável}

Em fluxograma o comando para Ler se da com o uso da figura a seguir, com o nome que identifica à variável dentro da figura, assim:

Nosso algoritmo, em pseudocódigo, ficaria assim:



```
Início
{declaração de variáveis}
NOME: literal
MATRICULA: numérico
{comandos de entrada de dados}
Leia NOME
Leia MATRICULA
```

Fim

Dessa forma, como esse algoritmo é executado sequencialmente, a primeira linha executada é a declaração da variável NOME.

Na seqüência, tem-se a declaração da variável MATRICULA. No terceiro comando, o algoritmo pede que você especifique um nome de entrada. Agora você pode especificar qualquer nome do seu interesse.

Esse nome será armazenado na variável NOME.

A mesma explicação vale para a variável MATRICULA.

Observe que não atribuímos à variável NOME e nem a variável MATRICULA nenhum valor de antemão, ou seja, o nome do aluno e a matrícula são adicionados pelo usuário quando estiver executando o programa.

Assim, o programa fica bem mais flexível e interessante.

Para aplicar o comando Leia, você deve inicialmente declarar a variável que o comando Leia vai ler.

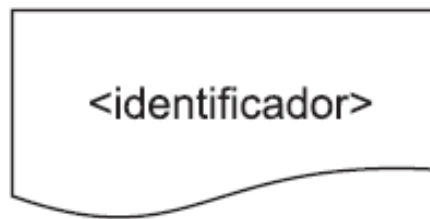
Comando Escreva

O comando Escreva é utilizado para mostrar o resultado na saída, seja no monitor, na impressora, no disquete etc.

Também podemos utilizar as palavras Imprima, Mostra, Exiba. Em pseudocódigo, o comando Escreva tem a seguinte sintaxe:

Escreva <identificadores>

Em fluxograma:



Vamos imaginar o seguinte, no algoritmo anterior, você quer mostrar na tela do seu computador o nome e a matrícula do aluno digitados.

```
Inicio
{declaração de variáveis}
NOME: literal
MATRICULA: numérico
{comandos de entrada de dados}
Leia NOME
Leia MATRICULA
{Mostra os valores das variáveis}
Escreva NOME
Escreva MATRICULA
Fim
```

Quando o algoritmo encontra o comando Escreva, o programa escreve o conteúdo da variável após o comando.

Assim, no algoritmo, o comando Escreva NOME, escreve o valor literal armazenado nessa variável, nesse caso, um nome de um aluno inserido por você, pelo teclado do seu computador.

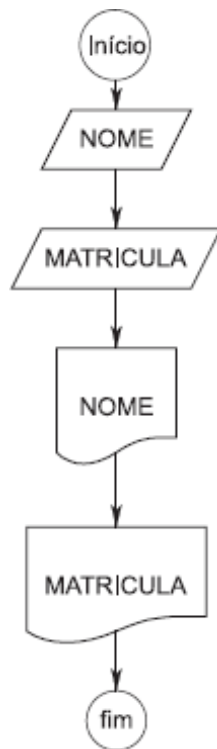
O algoritmo em fluxograma é mostrado a seguir.

Perceba que não há declaração de variáveis em fluxogramas. É subentendido que ao utilizarmos uma variável para a leitura de um dado, ela já tenha sido criada.

Compare Também o mesmo algoritmo escrito em pseudocódigo e em fluxograma.

Embora o fluxograma seja gráfico e que as simbologias nos dizem o que cada comando está realizando de forma bastante clara, para algoritmos um pouco maiores poderemos ter algumas dificuldades, principalmente em relação ao tamanho do mesmo.

É por isso que, apresentaremos os algoritmos em fluxogramas apenas para exemplificar a aplicação desses casos. Os demais exercícios serão feitos em pseudocódigos.



Para incrementar ainda mais o nosso algoritmo podemos colocar expressões textuais que gostaríamos que fossem mostradas no monitor.

Por exemplo, em vez de mostrar apenas o nome do aluno, seria interessante mostrar o texto: "Nome do Aluno cadastrado". Isso é muito fácil. Veja nosso algoritmo a seguir.

Início

```
{declaração de variáveis Nome e Matricula}
NOME: literal
MATRICULA: numérico
{comandos de entrada de dados para ler Nome e Matricula}
Leia NOME
Leia MATRICULA
{Mostra os valores das variáveis junto ao texto explicativo}
Escreva "Nome do Aluno Cadastrado:", NOME
Escreva "Nome da matricula do aluno:", MATRICULA
Fim
```

Ao ser encontrado o comando de saída Escreva "Nome do Aluno Cadastrado:", NOME, o literal Escreva "Nome do Aluno Cadastrado:" será escrito na saída do seu computador, mais o conteúdo da variável NOME.

Observe que o texto explicativo "Nome do Aluno Cadastrado:" e a variável NOME estão separados por vírgula.

A vírgula serve para indicar que estamos querendo imprimir vários dados em uma mesma linha na tela do computador.

Por exemplo, vamos supor que queremos que apareça na tela do computador a seguinte saída:

Nome do Aluno Cadastrado: Paulo. Matricula = 222779910.

Obs.: Podemos também considerar a variável MATRÍCULA como sendo literal. Neste caso, o valor 2227799-10 estaria correto.

Observe que estamos querendo mostrar tudo em uma mesma linha de saída.

Para isso utilizamos o comando `Escreva` e separamos os dados por vírgula.

O que é texto explicativo colocamos entre duplas aspas (" "). O que é variável colocamos simplesmente o nome da variável, seja ela numérica, literal ou lógica.

Para o nosso exemplo, teríamos o seguinte comando de saída:

`Escreva "Nome do Aluno Cadastrado: ", NOME, "Matricula = ", MATRICULA`

Perceba que o literal está entre duplas aspas e que a variável `NOME`, não.

Se você colocar a variável `NOME` entre duplas aspas também, a palavra `NOME` será impressa e não o conteúdo da variável. Por exemplo:

`Escreva "Nome do Aluno Cadastrado: NOME"`. Nesse caso, `NOME` não é mais variável, fazendo parte do literal apenas. Para acessar o conteúdo de uma variável, basta especificar o nome dela e utilizar o comando `Escreva` para imprimir o valor que ela armazena. Por exemplo: `Escreva "Nome do Aluno Cadastrado: ", NOME`.

Vale ressaltar que quando utilizamos o comando `LEIA`, o usuário não sabe o que tem que ser digitado, por exemplo:

`Leia NOME`

Para isso é necessário utilizarmos o comando `ESCREVA` juntamente com o comando `LEIA` para deixar claro ao usuário o que ele deve fazer:

`Escreva "Digite seu nome"`

`Leia NOME`

Ou seja, aparecerá na tela a mensagem "Digite seu nome". Então fica claro ao usuário que ele deve digitar o nome, o qual será armazenado na variável `NOME`.

Alguns autores Também sugerem utilizar o próprio comando `LEIA` para enviar a mensagem ao usuário. Assim:

`Leia "Digite seu nome:", NOME`

Ambos os casos estão corretos e serão aceitos.

Um exemplo completo

Um cliente vai ao restaurante e faz um pedido de acordo com o menu de entrada.

Itens	Preço
Cachorro-quente	R\$ 2,00
X-EGG	R\$ 4,50
Refrigerante – lata	R\$ 1,10
Batatas fritas	R\$ 1,00

Problema a resolver? Imprimir o pedido do cliente e mostra o valor total pago.

Pseudocódigo:

```
Inicio
{Declaração de variáveis}
CLIENTE : literal
QTDHOT, QTDXEGG, QTDREFRI, QTDBATATA,
TOTAL : numérico {QTDHOT = Quantidade de Cachorro-Quente; QTDXEGG = Quantidade de
XEGG;
QTDREFRI = Quantidade de Refrigerante; QTDBATATA
= Quantidade de Batatas Fritas; TOTAL = Valor total
comprado}
{Processamento do Algoritmo}
leia "Digite o nome do cliente:", CLIENTE {aparece na
tela - Digite o nome do cliente - e você deve escrever o
nome, que então é lido e guardado pelo programa}
leia "Digite a quantidade de Cachorro-Quente:",
QTDHOT
leia "Digite a quantidade de X-EGG:", QTDXEGG
leia "Digite a quantidade de refrigerantes:", QTDREFRI
leia "Digite a quantidade de batatas fritas:", QTDBATATA
TOTAL <- (QTDHOT * 2,0) + (QTDXEGG * 4,5) +
(QTDREFRI * 1,10) + (QTDBATATA * 1,00)
{Processamento de saída do algoritmo, imprimindo o nome do cliente e o valor total
da compra}
escreva "Cliente:", CLIENTE
escreva "Total do pedido:", TOTAL
Fim
```

Exercícios:

1. Modifique o algoritmo anterior (do exemplo que você acabou de ver) para permitir com que se entre com o preço dos itens no menu de entrada.

-
5. Crie um algoritmo em pseudocódigo que permita cadastrar um cliente de um banco: nome do correntista, o nome do banco, o número da conta, o limite de credito e saldo atual. Realize operações de crédito e débito da conta específica e, no final, imprima o saldo final do cliente.

Processamento condicional

Você estudou até agora como executar um algoritmo sequencialmente.

Esse processo é bastante simples, visto que começa com a palavra início, depois são executados comandos sequencialmente, e por fim, a palavra fim, determinando o término do algoritmo.

Para falar em processamento condicional, vamos recordar nossa receita do bolo de natal, onde executamos o processo de fazer o bolo passo a passo.

Imagine agora que no bolo temos a oportunidade de definir que tipo de chocolate utilizar (chocolate em pó ou granulado). A escolha vai depender unicamente do que temos em estoque em casa.

Dessa forma, condicionamos o nosso processo de fazer o bolo em função do tipo de chocolate que temos em casa, e o resultado será diferente.

Certamente, vamos executar passos diferentes em nossa receita em função dessa escolha.

*SE HOJE ESTIVER CHOVENDO...
VOU VER UM FILME.
SENÃO,
VOU CAMINHAR.*

Condicionamos o ato de ver o filme em função do tempo. Se estiver chovendo vou ver um filme, senão, vou caminhar e aproveitar o sol.

Você já deve ter percebido que o processo condicional também é um processo de decisão ou escolha de alternativas: se estiver chovendo, tomo a decisão de ir ao cinema. Perceba, Também, que o ato de ir ao cinema está vinculado ao fato de estar chovendo hoje.

Então pergunto: Vou ao cinema hoje? A resposta é sim se a proposição "hoje está chovendo" for verdadeira. A resposta é não e então vou caminhar se a proposição "hoje está chovendo" for falsa.

Tomamos a decisão em função de a proposição ser verdadeira ou falsa. Aqui, a tabela-verdade que você estudou na unidade 2 pode ajudar e muito.

Outro exemplo.

"Leia um número; se o número for maior que 20, divida o número por 2."

Para resolver o problema, devemos inicialmente fazer a seguinte pergunta: o número é maior do que 20?

Se a resposta for sim ou verdadeira, então dividimos o número por 2. Se for menor ou igual a 20, não fazemos nada.

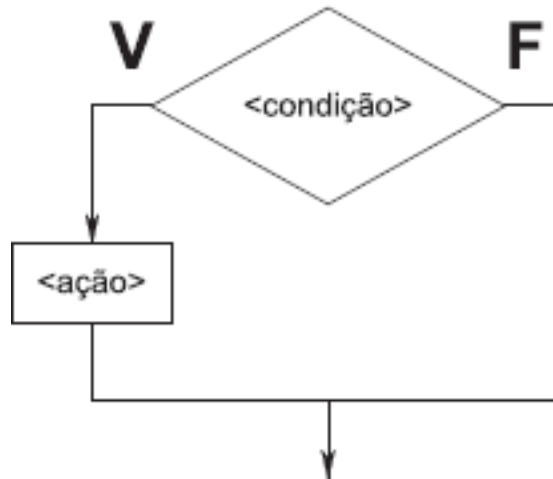
Poderíamos representar nossa solução pelo algoritmo a seguir:

```
se número > 20 então  
Número <- número/2  
fim-se {chamamos de fim-se para mostrar que a condição se encerra neste ponto}
```

A estrutura se/então/fim-se chama-se estrutura condicional simples. Os comandos que estão entre então e fim-se serão executados se e somente se a condição for verdadeira.

```
se <condição> então  
<comandos>  
fim-se
```

Em fluxograma, a representação é a seguinte:



Você já deve ter percebido que, no teste condicional, estamos trabalhando com proposições. E como proposições assumem apenas dois tipos de valores, falso ou verdadeiro, os testes condicionais serão baseados em variáveis lógicas.

Exemplos com Processo Condicional

1. Faça um algoritmo que leia um número e verifique se ele é maior do que 10. Se for, o algoritmo deve somar mais 10 a esse número. No final, o algoritmo deve imprimir o resultado final.

Solução:

- a) O problema indica a necessidade de termos uma variável numérica, onde será armazenado o valor lido. Então vamos declarar essa variável chamando-a de "NÚMERO", como está a seguir:

Início

NÚMERO: numérico

- b) 2. De acordo com o enunciado, o número deve ser lido, então precisamos do comando "leia":

Início

NÚMERO: numérico

Leia NÚMERO

- c) O algoritmo deve verificar se o número lido é maior que 10.

Início

NÚMERO: numérico

leia NÚMERO

se NÚMERO > 10 então

- d) Se o resultado da relação $NÚMERO > 10$ for verdadeiro, então devemos somar 10 ao número. Vamos então introduzir um comando onde NÚMERO passe a valer o seu antigo valor mais 10.

Início

NÚMERO: numérico

leia NÚMERO

```
se NÚMERO > 10 então
NÚMERO <- NÚMERO + 10
fim-se
```

e) Para finalizar, imprimimos o resultado do algoritmo

```
Inicio
NÚMERO: numérico
leia NÚMERO
se NÚMERO > 10 então
NÚMERO <- NÚMERO + 10
fim-se
escreva "O número é", NÚMERO
Fim
```

2. Vamos resolver o seguinte problema em pseudocódigo:

```
SE HOJE ESTIVER CHOVENDO...
VOU VER UM FILME.
SENÃO,
VOU CAMINHAR.
```

A proposição a ser testada é "está chovendo".

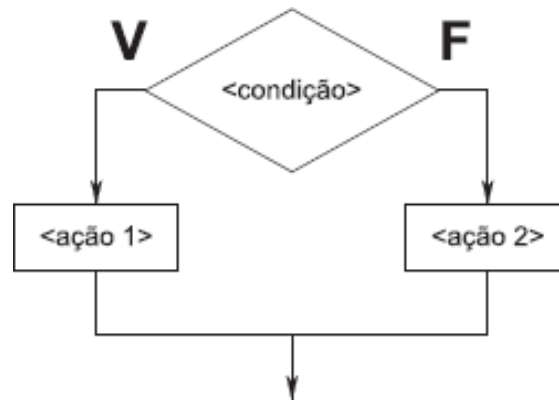
Agora estamos realmente preocupados em saber o que fazer ou que ações tomar, caso a proposição "está chovendo" não se confirme, ou seja, o que acontece quando a proposição é falsa. Para resolver essa situação, existe a estrutura de controle composta,

formada pela mesma estrutura condicional simples, acrescida da cláusula senão.

Vejamos como é a representação em pseudocódigo:

```
se <condição> então
<comandos>
senão
<comandos>
fim-se
```

Em fluxograma, a representação é a seguinte:



Baseado na estrutura de controle composto resolverá nosso algoritmo conforme a solução a seguir.

Solução:

1. O problema indica a necessidade de termos uma variável lógica, onde será armazenado um valor (verdadeiro ou falso) para especificar se está chovendo ou não.

```
Inicio
TEMPOCHUVA: lógico
```

2. De acordo com o enunciado, devemos verificar se está chovendo ou não. Para isso devemos atribuir verdadeiro ou falso a variável

```
TEMPOCHUVA.
Inicio
TEMPOCHUVA: lógico
Leia TEMPOCHUVA
```

3. O algoritmo deve verificar se a variável TEMPOCHUVA é verdadeira ou falsa.

```
Inicio
TEMPOCHUVA: lógico
Leia TEMPOCHUVA
se TEMPOCHUVA = verdadeiro então
```

4. Se o valor de TEMPOCHUVA for verdadeiro, então vou ao cinema. Imprimo o comando de saída "vou ao cinema"

```
Inicio
TEMPOCHUVA: lógico
Leia TEMPOCHUVA
se TEMPOCHUVA = verdadeiro então
Escreva "vou ao cinema"
fim-se
```

5. Se o valor de TEMPOCHUVA for falso, então vou caminhar. Imprimo o comando de saída "vou caminhar".

```
Inicio
TEMPOCHUVA: lógico
Leia TEMPOCHUVA
se TEMPOCHUVA = verdadeiro então
```

Escreva "vou ao cinema"
senão
Escreva "vou caminhar"
fim-se
Fim

3. Queremos resolver a seguinte situação: Se a média do aluno for superior a 7.0, devemos imprimir "a média é superior a 7.0".

Se a média do aluno for igual a 7.0, devemos imprimir "a média é igual a 7.0". Por fim, se a média for inferior a 7, devemos imprimir "a média é inferior a 7.0". Isso é muito útil para saber se o aluno passou, ou não!

Podemos perceber que temos três situações de saída, e todas as três estão baseadas na média final do aluno.

Vamos supor que a média do aluno é calculada por três avaliações (prova 1, prova 2 e prova 3). Resolveremos o problema por duas situações:

(a) Solução em pseudocódigo sem a cláusula *senão*.

1. O problema indica três notas e uma média final. Ou seja, precisamos criar 4 variáveis numéricas (3 para armazenar os valores das notas das provas e 1 para armazenar a média final).
{declaração das variáveis}

NOTA1, NOTA2, NOTA3, MEDIA: numérico

2. O algoritmo deve ler as três notas correspondendo as três avaliações.

NOTA1, NOTA2, NOTA3, MEDIA: numérico

Leia NOTA1

Leia NOTA2

Leia NOTA3

Como observação, vale colocar que poderíamos especificar a leitura das três variáveis utilizando o mesmo comando *leia*.

Exemplo:

Leia NOTA1, NOTA2, NOTA3

3. O algoritmo devera realizar o calculo da média final

NOTA1, NOTA2, NOTA3, MEDIA: numérico

Leia NOTA1, NOTA2, NOTA3

$MÉDIA \leftarrow (NOTA1 + NOTA2 + NOTA3)/3$

4. Algoritmo verifica se a média é maior que 7.0. Se for, escreve na saída o resultado.

NOTA1, NOTA2, NOTA3, MEDIA: numérico

Leia NOTA1, NOTA2, NOTA3

$MÉDIA \leftarrow (NOTA1 + NOTA2 + NOTA3)/3$

se $MÉDIA > 7.0$ então

escreva "A média é maior que 7.0"

5. Algoritmo verifica se a média é igual a 7.0. Se for, escreve na saída o resultado.

NOTA1, NOTA2, NOTA3, MEDIA: numérico

Leia NOTA1, NOTA2, NOTA3

$MÉDIA \leftarrow (NOTA1 + NOTA2 + NOTA3)/3$

se $MÉDIA > 7.0$ então

escreva "A média é maior que 7.0"

fim-se

se $MÉDIA = 7.0$ então

escreva "A média é igual a 7.0"

fim-se

6. Por fim, o algoritmo verifica se a média é menor que 7.0

NOTA1, NOTA2, NOTA3, MEDIA: numérico

Leia NOTA1, NOTA2, NOTA3

MÉDIA <- (NOTA1 + NOTA2 + NOTA3)/3

se MÉDIA > 7.0 então

escreva "A média é maior que 7.0"

fim-se

se MÉDIA = 7.0 então

escreva "A média é igual a 7.0"

fim-se

se MÉDIA < 7.0 então

escreva "A média é menor que 7.0"

fim-se

Fim

Você já poderia está se perguntando: onde está a cláusula senão?

Perceba que os testes condicionais, no nosso exemplo, são feitos por exclusão. Ou seja, se a resposta do primeiro teste (MÉDIA > 7.0) for verdadeira, então tanto o segundo teste (MÉDIA = 7.0)

quanto o terceiro (MÉDIA < 7.0) são falsos. Isso parece lógico porque não podemos ter uma média maior do que 7.0 e, ao mesmo tempo, 7.0 e inferior a 7.0. Porém esse algoritmo pode ser resolvido usando o senão.

(b) Solução em pseudocódigo com a cláusula senão.

Nosso algoritmo ficaria melhor assim:

Início

NOTA1, NOTA2, NOTA3, MEDIA: numérico

Leia NOTA1, NOTA2, NOTA3

MÉDIA <- (NOTA1 + NOTA2 + NOTA 3)/3

se MÉDIA > 7.0 então

escreva "A média é maior que 7.0"

senão

se MÉDIA = 7.0 então

escreva "A média é igual a 7.0"

Senão {NÃO PRECISA COLOCAR SE

NÃO SE

Senão se MÉDIA < 7.0 então}

escreva "A média é menor a 7.0"

fim-se

fim-se

Fim

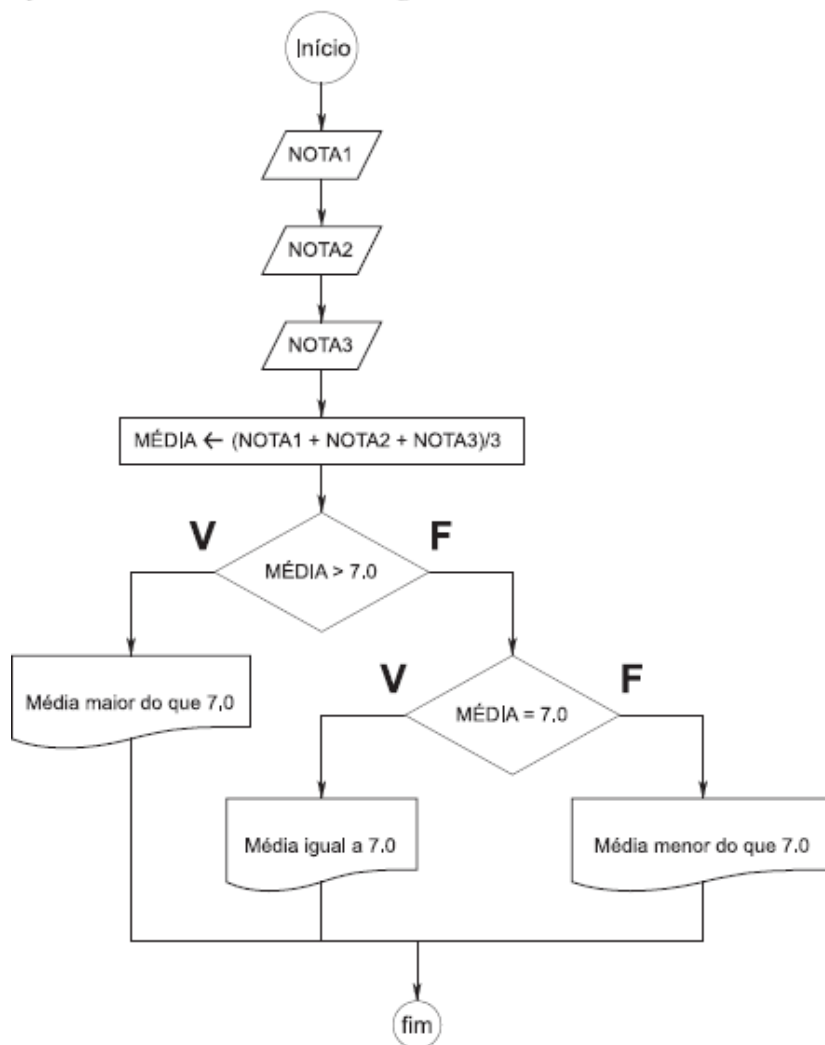
Qual a real vantagem desse último algoritmo em relação ao anterior onde não utilizamos na cláusula

senão?

A real vantagem está na velocidade dos testes. Perceba que, com a cláusula senão, quando uma alternativa se torna verdadeira, ela exclui as demais. Isso significa que o algoritmo não vai testar mais as alternativas, economizando tempo de processamento.

Se a média é maior que 7.0, para que testar se ela é igual a 7.0 ou menor que 7.0? são condições onde se uma alternativa se torna verdadeira, as demais são falsas.

Veja a mesma solução em fluxograma:



Para exemplificar mais, voltamos ao caso do cofre da unidade 2, lembra? Quero abrir um cofre de um banco com a senha 62. As duas proposições são:

a = "o primeiro dígito e o valor 6"

b = "o segundo dígito e o valor 2"

Em vez de resolver passo a passo, vamos apenas comentar os comandos do pseudocódigo:



comentar os

Solução em pseudocódigo

```

Início
{declaração de variáveis}
A, B: numérico
{leitura das variáveis, representando os dígitos do cofre}
leia A
leia B
{texto para verificar se os dígitos estão conforme a senha solicitada}
se A = 6 e B = 2 então
{cofre aberto com sucesso}
escreva "cofre está aberto"
senão
{cofre não está aberto}
escreva "senha digitada incorretamente"
  
```

fim-se

Fim

Quando o cofre será aberto?

Podemos perceber pelo algoritmo que isso acontece apenas quando a condição $A = 6$ e $B = 2$ for verdadeira. Em qualquer outro caso, o cofre não abrirá. Lembra da conjunção e utilizando a tabela-verdade? Veja como fica aqui no nosso exemplo:

<u>a</u>	<u>b</u>	<u>a e b</u>
V	V	V
V	F	F
F	V	F
F	F	F

De acordo com a tabela-verdade, apenas quando a e b forem verdadeiras e que o resultado será verdadeiro. Em nosso exemplo, a proposição a é verdadeira se o primeiro dígito for 6. Representamos isso pela variável numérica A . A proposição b é verdadeira se o segundo dígito for 2. Representamos isso pela variável numérica B . A conjunção A e B representa que ambas devem ser verdadeiras para que possamos executar o comando escreva "o cofre está aberto".

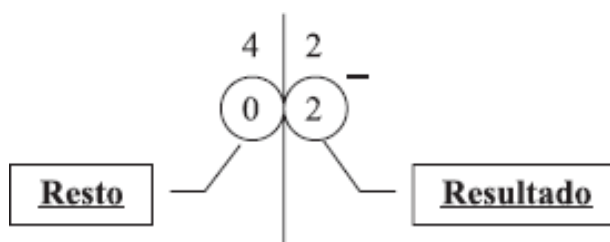
Exercite "seqüencial e condicional"

O objetivo desta parte é discutir exemplos de algoritmos com programação seqüencial e condicional. Portanto, aproveite, e na medida do possível, refaça-os sozinho quantas vezes achar necessário.

Isso vai dar a você mais confiança para continuar na seqüência.

1. Elabore um algoritmo que verifica se o número digitado é par ou ímpar.

A primeira pergunta que podemos fazer é: Como verifico na vida real se um número é par ou ímpar? Por exemplo, 2, 4, 5, 10, 17. Certamente você sabe que dentre os números apresentados, 2, 4 e 10 são pares e que 5 e 17 são ímpares. Por quê? Basta sabermos que um número é par se ele for divisível por 2, ou seja, o resto da divisão por 2 é 0 (zero). Caso contrário, o número é ímpar.



Qual o tipo de operação que devemos realizar para se obter o resto da divisão?

O resto da divisão é obtido por $\text{RESTO}(x, y)$ onde x é o dividendo e y o divisor. Assim, $\text{RESTO}(4, 2) = 0$ e $\text{RESTO}(17, 2) = 1$.

Dada a explicação de como verificar se um número é par ou ímpar, eis o nosso pseudocódigo:

Início

{declaração da variável numérica}

NÚMERO: numérico

```
leia: NÚMERO
{tomada de decisão para verificar se o número é par}
{perceba que ou o número é par ou ímpar}
se RESTO(NÚMERO, 2) = 0 então
escreva "Número é par"
senão
escreva "Número é ímpar"
fim-se
Fim
```

2. Crie um algoritmo que permita ao aluno responder qual é a capital do Brasil.

O problema indica a necessidade de termos uma variável literal, onde será armazenado o nome da cidade que digitarmos.

```
Início
{declaração de variáveis}
RESP: literal
{escrevendo um texto de entrada para mostrar ao aluno a pergunta}
escreva "Entre com o nome da capital do Brasil: "
{lendo a resposta digitada pelo aluno}
leia RESP
{Verifica se a variável RESP tem o literal Brasília}
se RESP = "Brasília" então
escreva "Parabéns"
senão
escreva "Errado: Estude mais geografia"
fim-se
Fim
```

3. No algoritmo do exemplo anterior, percebe-se que apenas se o aluno digitar Brasília a resposta será correta. Mas, se o aluno digitar brasília com "b" minúsculo? Resolva o exemplo anterior para aceitar tal situação.

```
Início
{declaração de variáveis}
RESP: literal
{escrevendo um texto de entrada para mostrar ao aluno a pergunta}
escreva "Entre com o nome da capital do Brasil: "
{lendo a resposta digitada pelo aluno}
leia RESP
{Verifica se a variável RESP tem o literal Brasília}
{perceba a operação de disjunção Ou}
se RESP = "Brasília" Ou RESP = "brasília" então
escreva "Parabéns"
senão
escreva "Errado: Estude mais geografia"
fim-se
Fim
```

4. Elabore um algoritmo que permita a entrada de dois números diferentes e verifique qual deles é o maior.

O problema indica a necessidade de termos duas variáveis numéricas para armazenar os dois números lidos:

```
Início
NUM1, NUM2: numérico
escreva "Entre com o número 1: "
leia NUM1
escreva "Entre com o número 2: "
leia NUM2
se NUM1 > NUM 2 então
```

```
escreva "O número maior é ", NUM1
senão
escreva "O número maior é ", NUM2
fim-se
Fim
```

5. Elabore um algoritmo que permita a entrada de três números e imprima o maior deles.

Novamente, precisamos criar três variáveis para armazenar os valores lidos.

```
Início
{declaração de variáveis}
NUM1, NUM2, NUM3: numérico
{comandos de leitura dos números}
escreva "Entre com o número 1: "
leia NUM1
escreva "Entre com o número 2: "
leia NUM2
escreva "Entre com o número 3: "
leia NUM3
{verifica se NUM1 é maior que NUM2 e NUM3}
se NUM1 > NUM2 e NUM1 > NUM3 então
escreva "Maior número é ", NUM1
senão
{o Senão indica que, ou NUM2 é maior que NUM1 ou NUM3 é maior que NUM1. Dessa
forma, basta comparar NUM2 com NUM3 para verificar quem é maior}
se NUM2 > NUM3 então
escreva "Maior número é ", NUM2
senão
escreva "Maior número é ", NUM3
fim-se
fim-se
Fim
```

Síntese

Nesta unidade você viu que, na elaboração de algoritmos, na maioria das vezes, estamos sempre incluindo comandos de entrada e/ou saída. Ou seja, alimentamos o nosso programa com entrada de dados, executamos o programa e colhemos (analisamos) os resultados de saída. Para realizar as operações de entrada e saída, apresentamos os comandos *leia* e *escreva* na sintaxe de pseudocódigo. O comando *leia* é utilizado para leitura de dados através de entradas do computador (teclado, por exemplo) e os valores lidos são armazenados em variáveis definidas anteriormente pelo usuário. O comando *escreva* é utilizado para escrever literais ou variáveis na saída do computador (por exemplo, no monitor).

Você Também conheceu os comandos sequenciais e os comandos condicionais, tanto em estrutura simples com apenas a cláusula *se/fim-se*, como Também a estrutura composta, formada pelo *se/senão/fim-se*.

Na próxima unidade você vai estudar o comando de repetição.

Até lá!

Exercícios

1. Elabore um algoritmo para os casos a seguir:
 - 1.1. Entrar com três números e imprimi-los em ordem decrescente (suponha números diferentes).

6.5 Um comerciante comprou um produto e quer vendê-lo com um lucro de 50% se o valor da compra for menor que R\$ 20,00. Caso contrário, o lucro será de 35%. Entrar com o valor do produto e imprimir o valor de venda.

6 – MÉTODOS DE REPETIÇÃO

Introdução

Muitas vezes, no nosso dia a dia, temos que repetir a mesma tarefa para chegar a um resultado específica. Por exemplo, na nossa receita do bolo de chocolate, suponha que vamos fazer três bolos para uma festa maior. Temos que repetir a mesma receita três vezes, sob pena de ter um bolo, diferente do outro. Seguindo a receita de apenas um bolo Também podemos ter um processo de repetição: mexer a massa até ficar consistente.

Perceba que ao fazermos três bolos, temos um número a ser atingido, nesse caso, três bolos. Poderiam ser 4, 5 etc., dependendo do tamanho da festa. No entanto, para a massa ficar consistente depende de uma serie de fatores e não apenas do número de vezes que iremos mexer. Se imaginarmos a expressão "a massa está consistente" como sendo uma proposição, ela pode assumir dois valores, verdadeiro ou falso. Então, o processo de repetição se basearia até que a proposição "a massa está consistente" fosse verdadeira. Caso contrário, o bolo não ficará tão bom assim.

Se as repetições aparecem de maneira não rara em algoritmos não computacionais, você já deve estar imaginando o que acontece com os algoritmos computacionais. Por isso, esta unidade tem o objetivo de mostrar a utilidade de estruturas de repetição, sua sintaxe e muitos exemplos.

Estruturas de controle de repetição

Utilizamos os comandos de repetição quando desejamos que um determinado conjunto de instruções ou comandos seja executado um número definido ou indefinido de vezes.

Por exemplo: podemos pedir para uma pessoa contar números até 1 até 10, de várias maneiras:

- Conte de 1 até 10.
- Enquanto não atingir o número 10, conte.
- Conte enquanto não atingir o número 10.

Vejamos a primeira alternativa "conte de 1 até 10". Nesse caso, ainda precisamos dizer para a pessoa contar de 1 em 1, certo?

Assim, ela começaria em 1, depois 2, e assim sucessivamente até alcançar o número 10.

Vamos supor que a contagem seja feita através de um grande painel. Quando começa a contagem, a pessoa apenas dita o número visto.

Poderemos dar a seguinte tarefa para essa pessoa:

Para os valores mostrados no painel de 1 até 10 faça:

"diga o número mostrado no painel"

Perceba que a instrução não deve deixar qualquer dúvida. A pessoa deve dizer os números mostrados no painel. Esses números começarão em 1 e terminarão em 10.

Nos algoritmos computacionais Também temos a necessidade de realizar esse tipo de controle.

Vamos imaginar um exemplo bastante simples, já visto na unidade 5. A

qui, para critério de exemplo, iremos apenas cadastrar o nome dos alunos.

O pseudocódigo a seguir mostra que devemos ler o nome do aluno e armazená-lo em uma variável chamada NOME.

Por fim, escrevemos esse nome no monitor do computador.

Início

NOME: literal

```
Leia NOME
Escreva NOME
Fim
```

Vamos supor agora que queremos cadastrar dois alunos. O algoritmo a seguir mostra este caso.

```
Início
NOME1: literal
NOME2: literal
Escreva "Digite 2 nomes"
Leia NOME1
Leia NOME2
Escreva NOME1
Escreva NOME2
Fim
```

Você será responsável pelo desenvolvimento de um sistema de cadastramento de alunos de uma escola com 100 alunos. Imagine o tamanho do problema se você quiser seguir o algoritmo anterior criando 100 variáveis.

Seu programa ficaria imenso. E se fossem 500 alunos? Maior ainda seria o seu programa. Para essas situações e que devemos utilizar controle de repetições, conforme a nossa contagem de números através do painel. Segundo essa mesma lógica, podemos dizer: Para número de alunos de 1 até 100 faça:

"processa o cadastramento dos alunos"

De uma maneira bem simples, nosso algoritmo ficaria conforme a seguir. Vamos colocar os números das linhas para facilitar o entendimento das repetições:

```
Linha 1: Início
Linha 2: NOME: literal
Linha 3: Para número de alunos de 1 até 100 faça
{processo de cadastro de alunos}
Linha 4: Escreva "Digite seu nome"
Linha 5: Leia NOME
Linha 6: Escreva NOME
Linha 7: Fim do cadastro
Linha 8: Fim
```

A seqüência de execução do algoritmo anterior é a seguinte:

Inicialmente o número de alunos cadastrados é 1 (linha 3).

São executados os comandos de cadastramento do aluno até encontrar Fim do cadastro (linha 6). No nosso exemplo, apenas os comandos leia e escreva são executados para cadastrar os alunos (linhas 4 e 5).

A seguir, o programa encontra a palavra Fim do cadastro na linha 6 e volta para a linha 3 do algoritmo.

O número de alunos cadastrados passa a ser 2, executando os comandos de cadastramento novamente, e assim sucessivamente.

Você já deve ter percebido que primeiro é incrementado o número de alunos cadastrados e depois é lido e escrito o nome do aluno. Esse processo de repetição continua até que o número de alunos cadastrados seja 100.

Porém, o algoritmo anterior não está seguindo uma sintaxe formal em pseudocódigo. E apenas um exemplo de como proceder a repetição.

Você pode estar se perguntando: mas, onde estão as informalidades?

Bem, para responder essa questão, faço a você outra pergunta:

O que significa o termo número de alunos dentro do algoritmo anterior?

Lembre-se de que um algoritmo é um conjunto de instruções sem ambigüidades, por isso, não deve apresentar dúvidas em sua execução.

Então, antes de estudar a sintaxe em pseudocódigo, vamos responder essa questão bastante importante.

Inicialmente, o termo “número de alunos” significa que estamos interessados em contar o número de alunos cadastrados.

Para cada aluno cadastrado, temos que ler e imprimir o nome do aluno no monitor apenas.

Sabendo que precisamos controlar o número de alunos já cadastrados, e necessário armazenar esse valor em algum lugar, e esse lugar, em termos de computação, é uma variável.

Lembra do conceito de variável e para que ela serve?

Podemos representá-la por um nome apropriado, por exemplo, NUMALUNOS.

Dessa forma, nosso algoritmo poderia ficar assim:

```
Linha 1: Inicio
Linha 2: NOME: literal
{variável NUMALUNOS criada para
armazenar o número de alunos cadastrados}
Linha 3: NUMALUNOS: numérico
Linha 4: Para NUMALUNOS de 1 até 100 faça
{processo de cadastro de alunos}
Linha 5: Escreva "Digite seu nome"
Linha 6: Leia NOME
Linha 7: Escreva NOME
Linha 8: Fim do cadastro
Linha 9: Fim
```

Pois bem, a variável NUMALUNOS vai sendo incrementada a medida que cadastramos mais um aluno, até que o número de alunos cadastrados seja 100, ou seja, até que a variável NUMALUNOS atinja o valor 100, como está colocado no nosso exemplo.

Perceba que o processo de repetição está entre as linhas 4 e 8 do nosso algoritmo, e tudo que está dentro desse laço de repetição é executado, enquanto NUMALUNOS for menor ou igual a 100.

Chamamos esses comandos de bloco de comandos. Para uma melhor visualização, colocamos esse bloco endentado, possibilitando visualizar com bastante clareza onde começa e termina o laço de repetição e quais os comandos que serão executados.

Sempre endente os comandos dentro de laços de repetição. Isso facilita bastante o acompanhamento do algoritmo e correções de possíveis erros.

Por fim, precisamos saber como a variável NUMALUNOS é incrementada. Para nosso exemplo, estamos cadastrando aluno por aluno, sendo assim, a variável NUMALUNOS é incrementada uma por uma, semelhante a um painel mostrando o número de alunos já cadastrados pela secretaria da escola. Essa forma de incremento não está sendo mostrada no algoritmo anterior, o que sugere que, em nossa sintaxe de pseudocódigo, temos que mostrar como será realizado os incrementos.

Importante

Cada vez que a variável NUMALUNOS é incrementada, é verificado se o valor armazenado na variável é menor ou igual a 100.

Isso é estabelecido pela palavra até em nosso algoritmo.

Perceba que estamos querendo contar até 100. Caso seja menor ou igual a 100, o sistema continua cadastrando alunos (lendo e escrevendo o nome do aluno).

Quando atingir o valor maior que 100, o sistema vai para os comandos após o fim do cadastramento, não executando mais o laço de repetição ou o cadastramento de alunos.

Dessa forma, nosso algoritmo fica completo. Sabemos agora o que contar e como será contado. Este é o procedimento básico para a repetição.

Como estrutura de repetição, este comando Também tem uma sintaxe específica, e pode ser representado em pseudocódigo e fluxograma.

Pseudocódigo:

```
para <variável> de <valor inicial> até <valor final> passo
<valor a ser acrescentado ou diminuído> faça
<comandos a serem repetidos>
fim-para
```

Esta estrutura é conhecida como estrutura para/faça/fim-para. A explicação de cada parâmetro colocado entre < ... > está a seguir:

<variável> Variável que armazenará o valor do laço de repetição.

É a variável de contagem. Seria semelhante ao nosso painel, mostrando a contagem dos números de 1 até 10.

<valor inicial> Valor inicial onde começa a contagem. Pode ser 1 como no nosso exemplo de contagem de 1 até 10, como pode ser qualquer outro número positivo, zero ou negativo.

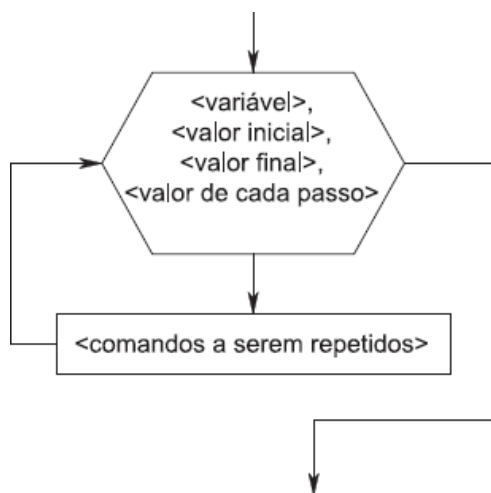
<valor final> Valor final onde termina a contagem. Pode ser 10 como no nosso exemplo de contagem de 1 até 10, como pode ser qualquer outro número positivo, zero ou negativo.

<valor a ser incrementado ou decrementado> Representa a maneira de como será executada a repetição.

Por exemplo, contagem de 1 em 1, 2 em 2 etc. Na contagem de números de 1 até 10, escolhemos por contar de 1 em 1. Poderíamos estar contando apenas números pares.

Nesse caso, estaríamos contando de 2 em 2.

Em fluxograma:



Exemplo 1:

```
para NUMALUNOS de 1 até 100 passo 1 faça  
  escreva "Digite seu nome"  
  leia NOME  
  escreva "Aluno cadastrado:" NOME  
fim-para
```

Para o algoritmo anterior, os parâmetros são os seguintes:

<variável> NUMALUNOS
<valor inicial> 1
<valor final> 100
<valor de incremento> 1

O que significa este algoritmo?

A leitura deste algoritmo nos mostra que a variável NUMALUNOS tem valor inicial 1 e vai até 100, incrementada passo a passo.

Como é executada a lógica deste algoritmo?

Inicialmente, a variável NUMALUNOS recebe o valor 1.

Há um teste para averiguar se o valor de NUMALUNOS é menor ou igual a 100.

Como NUMALUNOS possui valor 1, é executada a parte do algoritmo de leitura (leia) e escrita (escreva) do nome do aluno.

Na verdade, todos os comandos que são encontrados entre faça e fim-para são executados.

Ao encontrar a palavra fim-para (fim do laço de repetição), o algoritmo volta para o comando para e a variável NUMALUNOS é incrementada de um (valor numérico 1) (passo 1), ou seja, a variável NUMALUNOS assume valor 2.

Após o incremento, a variável é testada novamente para certificar se ela é menor ou igual a 100.

Como a variável armazena o valor 2, os comandos de leitura e escrita são executados novamente até encontrar a palavra fim-para. Após isso, a variável NUMALUNOS é incrementada novamente e recomeça o ciclo até NUMALUNOS atingir o valor 100.

Quando NUMALUNOS atingir o valor 100, os comandos de leitura e escrita ainda são executados, e após encontrar a palavra fim-para, o algoritmo volta para o comando para e a variável NUMALUNOS é incrementada novamente, passando a ter valor numérico de 101.

Neste ponto, verifica-se que NUMALUNOS possui valor maior que 100.

Assim, os comandos de leitura e escrita não são executados e o algoritmo salta para os comandos após a palavra fim-para.

Exemplo 2:

```
para CONTA de 5 até 0 passo -1 faça escreva "o valor da variável conta é: ", conta fim-para.
```

Neste algoritmo anterior, a variável inicial é CONTA.

A variável tem seu valor inicial igual a 5 e o valor final igual a 0.

Perceba que o valor inicial do comando para é maior que o final.

Porém, em vez da variável CONTA ser incrementada de 1, ela é incrementada de -1, ou seja, a variável CONTA é decrementada de 1.

Como são executados os testes de repetição neste caso?

Inicialmente a variável CONTA recebe valor 5.

É verificado que 5 é maior que 0, assim o comando escreva é executado e o resultado de saída é "o valor da variável conta é: 5".

Após executar o comando e encontrar a palavra fim-para, o algoritmo volta para o comando para e decrementa a variável de 1, passando o valor da variável CONTA para 4.

Novo teste de comparação entre o valor da variável CONTA e 0 é realizado.

É verificado que 4 é maior que 0, assim o comando escreva é executado e o resultado de saída é "o valor da variável conta é: 4", e assim sucessivamente.

Alguns algoritmos resolvidos

1. Criar um algoritmo que entre com cinco números e imprimir o dobro de cada número.

```
Inicio
{declaração das variáveis:
NUM: valor numérico a ser entrado pelo teclado
CONTA: variável que serve para realizar o laço de repetição}
NUM, CONTA: numérico
Para CONTA de 1 até 5 passo 1 faça
Escreva "Digite um número"
Leia NUM {leitura da variável NUM}
Escreva "Dobro de", NUM*2 {multiplica NUM por 2}
Fim-para {fim do laço de repetição}
Escreva "Fim do algoritmo de repetição" {comando executado após a variável CONTA
assumir valor maior do que 5}
Fim {fim do algoritmo}
```

- Inicialmente são criadas as variáveis NUM e CONTA. A variável NUM serve para armazenar o valor de entrada pelo teclado. CONTA é uma variável para realizar o laço de repetição.
- No comando para, a variável CONTA é inicializada na primeira vez com o valor numérico. É verificado que CONTA possui valor numérico menor que 5. Por isso, os comandos leia NUM e escreva NUM*2 são executados, até encontrar fim-para.
- Ao encontrar o comando fim-para, o algoritmo volta para o comando para incrementando a variável CONTA de 1, ou seja, a variável CONTA passa a ter valor numérico de 2. Como 2 é menor que 5, os comandos leia e escreva são executados novamente, e assim sucessivamente até CONTA assumir valor maior que 5.
- Quando o valor da variável CONTA ultrapassar o valor 5, o algoritmo passa para o primeiro comando após fim-para, no nosso exemplo, é executado o comando Escreva "Fim do algoritmo de repetição" e após o comando fim do algoritmo.

2. Criar um algoritmo que imprima todos os números pares de 1 até 100.

```
Inicio
{declaração das variáveis:
CONTA: variável que serve para realizar o laço de repetição}
CONTA: numérico
Para CONTA de 0 até 100 passo 2 faça
Escreva CONTA {mostra número para na tela}
Fim-para {fim do laço de repetição}
Fim {fim do algoritmo}
```


- Perceba agora que a variável CONTA é inicializada com o valor numérico 0 e vai até 100, porém incrementada de dois em dois.
- Como o valor inicial da variável CONTA é 0, o comando escreva é executado, mostrando na tela o valor 0.
- Como temos apenas um comando dentro do laço de repetição, o algoritmo encontra a palavra fimpara e volta para o comando para, incrementando a variável CONTA de 2. Assim, o novo valor da variável CONTA é o valor numérico de 2. O comando escreva é executado e o valor 2 é mostrado na tela. Novamente a variável CONTA tem seu valor incrementado de 2, passando agora de 2 para 4. O comando escreva é executado novamente, escrevendo o valor numérico 4 na tela. Novamente a variável CONTA tem seu valor incrementado de mais 2, passando ao valor 100. A sequência se repete até que a variável CONTA ultrapassar o valor de 100.
- Quando o valor contido na variável CONTA for 100 o comando escreva é executado mostrando o valor 100 na tela. Após isso, a variável CONTA tem seu valor incrementado de 2 passando para 102. Nesse caso, como 102 é maior do que 100 o algoritmo salta para os comandos após fim-para. Como não há nada mais para executar, senão o comando fim, o algoritmo termina.

3. Criar um algoritmo que some todos os números entre 30 e 60.

Antes de mostrarmos a solução, vamos entender o problema.

Queremos somar todos os valores contidos entre 30 e 60, inclusive os números 30 e 60.

Veja: Queremos somar $30+31+32+33+ \dots +60$ e mostrar o valor dessa soma.

Então vamos a solução passo a passo.

Inicialmente devemos criar duas variáveis: uma para realizar o laço de repetição e a outra para armazenar a soma de todos os valores.

3.1 Inicio
SOMA, CONTA: numérico

Criadas as variáveis, temos que realizar o laço de repetição que vai de 30 a 60. Nesse caso, no comando para a variável CONTA terá valor inicial de 30, valor final de 60 e será incrementada de 1 em 1.

Seguindo nossa sintaxe:

3.2 Para CONTA de 30 até 60 passo 1 faça {comandos de repetição}
Fim-para

Quais os comandos que queremos executar dentro do laço de repetição? Queremos somar todos os números entre 30 e 60.

A variável que vai armazenar a soma dos números será a variável SOMA. Perceba que a variável SOMA devesse representar a soma dos números $30+31+32+33 +60$.

Então, vamos montar as seqüências.

- Para o primeiro passo da repetição, CONTA assume valor 30. Dessa forma, soma Também deve ser 30, pelo fato da nossa contagem começar com 30. Assim, poderíamos pensar em algo como atribuir o valor 30 a variável SOMA. De acordo com a sintaxe de atribuição já vista, teríamos $SOMA \leftarrow 30$. De acordo com o passo 3.2 anterior, poderíamos imaginar algo assim:

Para CONTA de 30 até 60 passo 1 faça SOMA \leftarrow 30
Fim-para

Se observarmos bem o algoritmo acima, vamos verificar que para cada passo de execução, o valor de SOMA não se altera, sempre permanecendo em 30. Como fazer com que a variável SOMA assumira os valores da variável CONTA? Bem simples basta fazer `SOMA <- CONTA`, certo?

Para cada laço de repetição, agora, SOMA recebera o valor numérico armazenado em CONTA, ou seja, 30, 31, 32..., de acordo com cada repetição do laço. Nossa variável SOMA agora terá os valores 30, 31, 32,...60.

Obviamente você já deve ter observado que não estamos somando nada. O valor final de SOMA será igual ao valor final da variável CONTA, ou seja, o valor numérico 60.

b) O que queremos fazer pode ser visto no quadro a seguir:

Variáveis 1ª repetição 2ª repetição 3ª repetição 4ª repetição

CONTA 30 31 32 33

SOMA 30 (30+31)=61 (30+31+32)= 93 (30+31+32+33)= 126

Observe inicialmente a 2ª repetição.

Nela podemos verificar que a variável CONTA assume valor numérico igual a 31 e a variável SOMA 61.

O valor numérico 61 é a soma de 30 (valor anterior da variável SOMA na 1ª repetição) com 31 que é o valor atual armazenado em CONTA.

Para a 3ª repetição, podemos perceber que o valor da variável CONTA pula para 32 e a variável SOMA assume valor 93. Este valor 93 é o resultado da soma de 30+31+32.

Perceba que 30+31=61 e justamente o valor da variável SOMA na repetição anterior. Esse valor anterior, 61, somado com o valor da variável CONTA, 32, dá o resultado 93.

A variável SOMA tem como resultado de cada repetição a soma de seu valor numérico anterior a repetição, com o valor atual da variável CONTA. No passo c, este procedimento é descrito em algoritmo.

c) Bem, se a variável SOMA é resultado de um cálculo matemático de adição do valor anterior da variável SOMA com o valor atual da variável CONTA, temos que

SOMA <- SOMA + CONTA.

Vamos verificar novamente como a instrução é executada passo a passo, a partir da 2ª repetição.

Na 2ª repetição, o valor anterior de SOMA (1ª repetição) é 30 e o valor de CONTA é 31. Pois bem, perceba que o valor da variável SOMA permanece 30 até que seja atribuído um novo valor para ela. Assim, o comando `SOMA <- SOMA + CONTA` realiza o seguinte cálculo:

SOMA <- 30 + 31.

Realizando a operação de adição, `SOMA <- 61`.

Na 3ª repetição, a variável SOMA armazena o valor numérico 61 é o valor de CONTA e 32. Pois bem, perceba que o valor da variável SOMA permanece 61, até que seja atribuído um novo valor para ela. Assim, o comando `SOMA <- SOMA + CONTA` realiza o seguinte cálculo:

SOMA <- 61 + 32.

Realizando a operação de adição, $SOMA \leftarrow 93$. E assim, sucessivamente.

d) Nosso algoritmo de repetição poderia ficar assim:

```
Para CONTA de 30 até 60 passo 1 faça  $SOMA \leftarrow SOMA + CONTA$   
Fim-para
```

e) A cada repetição, pegamos o valor da variável SOMA (valor anterior) e somamos com o valor atual da variável CONTA e armazenamos o resultado na própria variável SOMA, atualizando seu valor

Também para a próxima iteração.

Mas agora, vamos observar a 1ª repetição.

Sabemos que a variável SOMA é igual ao valor anterior da variável mais a variável CONTA, ou seja, $SOMA \leftarrow SOMA + CONTA$. Para a 1ª repetição, sabemos que CONTA possui valor 30, mas e a variável SOMA?

Qual o valor anterior de SOMA? não temos repetição e por isso não temos um valor definido para ela até agora. Se seguirmos a mesma lógica, precisamos saber o valor da variável SOMA antes de iniciar as repetições.

Porém, um fato contribui para encontrarmos o valor de inicial da variável SOMA: Se estamos somando todos os números de 30 a 60, então o valor da variável SOMA após a 1ª repetição deve ser 30.

Isso significa que executando o comando de atribuição $SOMA \leftarrow SOMA + CONTA$ na 1ª repetição, temos que $SOMA \leftarrow (\text{valor inicial de SOMA antes da 1ª iteração}) + CONTA$.

Como a variável SOMA deve começar a somar a partir de 30 (valor inicial da variável CONTA), então temos a seguinte conclusão por meio de simples expressão matemática:

```
 $SOMA = (\text{valor inicial de SOMA antes da 1ª iteração}) + CONTA$   
 $30 = (\text{valor inicial de SOMA antes da 1ª iteração}) + 30$  (valor inicial de SOMA  
antes da 1ª iteração) = 0;  
Iterar: repetir
```

Dessa forma, o valor da variável SOMA deve ser inicializada com o valor numérico 0 antes da 1ª repetição.

f) f) Nosso algoritmo completo fica assim agora:

```
Inicio  
SOMA, CONTA: numérico  
 $SOMA \leftarrow 0$  {variável SOMA inicializada}  
Para CONTA de 30 até 60 passo 1 faça  $SOMA \leftarrow SOMA + CONTA$   
Fim-para  
Escreva "Resultado da soma =" SOMA  
Fim
```

4. Criar um algoritmo, que leia os limites inferior e superior de um intervalo e imprimir todos os números pares no intervalo considerado e seu somatório. Suponha que os dados digitados são para um intervalo crescente.

Limite Inferior: 3
Limite Superior: 12

Saída do algoritmo: 4, 6, 8, 10, 12.
Soma: 40.

- a) Inicialmente temos que criar 4 variáveis: a primeira para armazenar o valor inferior do intervalo, a segunda para armazenar o valor superior do intervalo, a terceira para realizar os passos de repetição e a quarta para armazenar o valor da soma.

```
Inicio
{declaração das variáveis}
LINFERIOR, LSUPERIOR, SOMA, CONTA: numérico
```

- b) Declarada as variáveis, é momento de obtê-las através do teclado.

```
Escreva "Entre com o limite inferior:"
Leia LINFERIOR
Escreva "Entre com o limite superior:"
Leia LSUPERIOR
```

- c) Definido os comandos de entrada, é hora de realizar o processamento do algoritmo. Nosso processamento é realizar um laço de repetição que começa em LINFERIOR e termina em LSUPERIOR, mostrando os números pares e somando-os.

O passo para verificar se um número é par é bastante simples:

Basta dividir por dois e verificar se o resto da operação é 0 (zero).

Dessa forma, nosso laço de repetição seria assim:

```
Para CONTA de LINFERIOR até LSUPERIOR passo 1 faça
Se RESTO(CONTA, 2) = 0 então
Escreva CONTA
SOMA <- SOMA + CONTA
Fim-se
Fim-para
```

Perceba que para cada passo da repetição, é verificado se a operação RESTO(CONTA, 2) é zero.

Caso seja zero, significando que o valor atual da variável CONTA é divisível por dois, e portanto é um número par, processa-se o comando escreva e o comando de atribuição SOMA <- SOMA + CONTA.

De forma semelhante ao nosso algoritmo anterior, a variável SOMA deve ser inicializada com 0 (zero). Assim, nosso algoritmo completo é:

```
Inicio
{declaração das variáveis}
LINFERIOR, LSUPERIOR, SOMA, CONTA:
numérico
SOMA <- 0 {inicializa variável SOMA}
Escreva "Entre com o limite inferior:"
Leia LINFERIOR
Escreva "Entre com o limite superior:"
Leia LSUPERIOR
Para CONTA de LINFERIOR até LSUPERIOR
passo 1 faça
Se RESTO(CONTA, 2) = 0 então
Escreva CONTA
SOMA <- SOMA + CONTA
Fim-se
Fim-para
Escreva "A soma dos números pares é:", SOMA
```

Fim

5. Crie o algoritmo para o problema a seguir, representando-o em pseudocódigo. A avaliação de um aluno nas disciplinas de uma escola segue os critérios a seguir.
- Em toda disciplina serão aplicadas três provas.
 - A média final é obtida com a média aritmética das três notas.
 - Para que o aluno seja aprovado, a sua média deve ser igual ou superior a 7,9 e ter a frequência mínima de 80% nas 32 aulas ministradas.

Faça um algoritmo para:

1. Ler os números das matrículas de 70 alunos, as três notas de cada um e o número de aulas frequentadas por eles.
2. Calcular e imprimir o número da matrícula do aluno, a sua média final e o resultado (se aprovado ou não).
3. Imprimir a média da turma, a maior e a menor média da turma de alunos;
4. Imprimir o total de alunos aprovados;
5. Imprimir o total de alunos reprovados por falta e por nota.

As explicações constam nos comentários. A sua atividade agora é entender o problema e como ele está solucionado pelo algoritmo a seguir.

Dicas:

- Verifique com cuidado cada uma das variáveis criadas;
- Entenda toda a entrada de dados;
- Verifique o processamento do algoritmo;
- Por fim, verifique qual ou quais são as saídas do algoritmo.

Pseudocódigo:

```
Inicio
{declaração das variáveis}
{as variáveis NOTA1, NOTA2, NOTA3 servem para
armazenar as notas dos alunos}

NUMMATRICULA, NOTA1, NOTA2, NOTA3,
NUMAULAS,
{as variáveis MÉDIA e MEDIATURMA servem para
armazenar as médias de cada aluno e a média da turma respectivamente}

MEDIA, MEDIATURMA,
{as variáveis MAIORMÉDIA e MENORMÉDIA servem
para armazenar as menores e maiores médias dos alunos respectivamente}

MAIORMEDIA, MENORMEDIA, NUMAPROVADOS,
{NUMREPROVMÉDIA é uma variável para armazenar o
número de estudantes reprovados por média inferior a 7.0}

NUMREPROVMEDIA,
{NUMREPROVFALTA é uma variável para armazenar o
número de estudantes reprovados por falta}

NUMREPROVFALTA, CONTADOR : numérico
NUMAPROVADOS <- 0
NUMREPROVMÉDIA <-0
NUMREPROVFALTA <-0
```

```

{inicializa MAIORMÉDIA com -1 pois qualquer média será
maior ou igual a zero}

MAIORMÉDIA <--1
{inicializa MENORMÉDIA com 11 pois qualquer média será
menor ou igual a 10}

MENORMÉDIA <-11
MEDIATURMA <-0

{leitura de 70 alunos}
para CONTADOR de 1 a 70 faça
leia "Digite o número de matricula do aluno:", NUMMATRICULA
leia "Digite a primeira nota:", NOTA1
leia "Digite a segunda nota:", NOTA2
leia "Digite a terceira nota:", NOTA3
leia "Digite o número de aulas freqüentadas:",
NUMAULAS

{cálculo da média do aluno}
MÉDIA <- (NOTA1 + NOTA2 + NOTA3)/3

{cálculo da média da turma}
MEDIATURMA <- MEDIATURMA + MEDIA

{escreve dados do aluno cadastrado}
escreva "Número da matricula:", NUMMATRICULA
escreva "Média final:", MEDIA
escreva "Número de aulas freqüentadas:", NUMAULAS

{verifica se o aluno está aprovado por média e por freqüência}
se MÉDIA >= 7 e NUMAULAS >= (32 * 0,8) então
escreva "Resultado: Aprovado"
NUMAPROVADOS <- NUMAPROVADOS + 1
{senão, verifica se o aluno está reprovado por média}

{neste caso, ou o aluno está reprovado por média ou por falta
ou ambos}
senão
se MÉDIA < 7 então
escreva "Resultado: Reprovado por Media"

NUMREPROVMÉDIA <- NUMREPROVMEDIA + 1
fim-se {fim do se MÉDIA < 7}

{senão, verifica se o aluno está reprovado por faltas}
se NUMAULAS < (32 * 0.8) então
escreva "Resultado: Reprovado por falta"
NUMREPROVFALTA <- NUMREPROVFALTA+ 1

fim-se {fim do se NUMAULAS < (32 * 0.8)}
fim-se {fim do se MÉDIA >= 7 e NUMAULAS >= (32 * 0,8)}

se MÉDIA > MAIORMÉDIA então
MAIORMÉDIA <- MEDIA
fim-se {fim do se MÉDIA > MAIORMÉDIA }
se MÉDIA < MENORMÉDIA então
MENORMÉDIA <- MEDIA
fim-se {fim do se MÉDIA < MENORMÉDIA }
fim-para

{Calcula a média da turma}
MEDIATURMA <- MEDIATURMA / 70

```

```
{resultado de saída do algoritmo}
escreva "Total de alunos aprovados:", NUMAPROVADOS
escreva "Total de alunos reprovados por media:", NUMREPROVMEDIA
escreva "Total de alunos reprovados por falta:", NUMREPROVFALTA
escreva "Média da Turma:", MEDIATURMA
escreva "Maior média da turma:", MAIORMEDIA
escreva "Menor média da turma:", MENORMEDIA
Fim {fim do algoritmo}
```

Síntese

Nesta unidade você aprendeu que devemos utilizar os comandos de repetição quando desejamos que um determinado conjunto de instruções, ou comandos seja executado um número definido ou indefinido de vezes. Você estudou a estrutura para/faça/fim-para.

Esta estrutura especi fica quantas vezes vamos executar o laço de repetição. Para isso, é necessário definir qual a variável que está sendo controlada, seu valor inicial, seu valor final e como será incrementada ou decrementada.

A estrutura para/faça/fim-para é utilizada principalmente quando sabemos de antemão quantas vezes iremos executar o laço de repetição.

Exercícios:

Elabore um algoritmo em pseudocódigo para cada uma das atividades a seguir:

1. Ler 200 números e imprimir quantos são pares e quantos são ímpares.

2. Entrar com 20 números e imprimir a soma dos positivos, e o total de números negativos.

3. Entrar com 10 números (positivos ou negativos) e imprimir o maior e o menor.

7 - MANIPULAÇÃO DE VETORES

Introdução

Embora uma variável possa assumir diferentes valores, ela só pode armazenar um valor a cada instante.

Você também estudou que as variáveis podem ser de três tipos: numéricas, alfanuméricas e lógicas, e que para declarar uma variável precisamos definir o seu nome e que tipo de dados será armazenado nela.

Veja a seguir:

início

```
nome: literal {variável do tipo literal} idade: numérica {variável do tipo numérica}  
fim
```

Outro exemplo bem fácil, já utilizando o sinal de atribuição.

início

```
nome: literal {variável do tipo literal} idade: numérica {variável do tipo numérica} idade <- 22  
{variável idade assume valor 22}  
nome <- "Paulo Pereira" {variável nome assume "Paulo Pereira"}  
fim
```

O algoritmo acima, amplamente discutido em Lógica de Programação , serve de base para a seguinte questão: Suponha que precisamos cadastrar dois nomes de clientes com suas respectivas idades. Basta criar duas novas variáveis, conforme mostro a seguir:

início

```
nome1, nome2: literal {variáveis do tipo literal}  
idade1, idade2: numérica {variáveis do tipo numérica}  
idade1 <- 22 {variável idade1 assume valor 22}  
nome1 <- "Paulo Pereira" {variável nome1 assume "Paulo Pereira"}  
idade2 <- 38 {variável idade2 assume valor 38}  
nome2 <- "Ana Luiza" {variável nome2 assume "Ana Luiza"}  
fim
```

Mas, e se quisermos criar um cadastro de alunos de um colégio? E agora? Quantas variáveis precisam criar? 500, 2500, 10000?

Basta utilizar o conceito de vetores para lidar com situações como essas.

- Toda a lógica de programação estudada até agora vai se repetir. Não há nenhum outro comando. Tudo que você aprendeu em Lógica de Programação I será utilizado agora. Os únicos assuntos novos são o conceito, a criação e a utilização de vetores.

Conceito e declaração de vetores

Antes de definir vetor, imaginemos a seguinte situação. Precisamos criar um programa que armazene as seguintes notas de um aluno em Lógica de Programação II: 8.0, 10.0, 9.0, 10.0, 8.5, 10.0 e que calcule a média final. A solução é bem simples:

```
início
nota, conta, media, soma: numérico
soma <- 0 {inicializa variável soma com o valor 0}
para conta de 1 até 6 passo 1 faça {laço de repetição}
  escreva "Entre com a nota: "
  leia nota {leitura da nota}
  soma <- soma + nota {soma de todas as notas entradas}
fim-para
media ← soma/6 {calcula a média final}
escreva "A média final é ", media {mostra o resultado na tela}
fim
```

Mas agora consta a seguinte complexidade: necessitamos imprimir também todas as notas do aluno, além da média final.

Uma solução extremamente pobre seria criar seis variáveis para armazenar as seis notas digitadas e imprimi-las.

Mas, se tiver mais notas (80, por exemplo), seu algoritmo já não resolveria mais o problema. Para resolver essa situação, utilize o conceito de vetor.

Um vetor nada mais é do que uma variável que pode armazenar vários valores do mesmo tipo.

Bem, mas o que significa isso? Inicialmente, acompanhe o conceito de variáveis e sua

Quando definimos uma variável, alocamos um espaço na memória do computador para armazenar uma e somente uma constante por vez, seja ela literal, numérica ou lógica.

Quando atribuímos um valor à variável sobrescrevemos seu conteúdo. Por exemplo, ao criarmos uma variável numérica chamada nota, criamos um espaço na memória para armazenar apenas um valor numérico por vez, conforme a seguir:

```
nota: numérica
.....
nota ← 10 {variável armazena valor numérico 10}
escreva "O valor da variável nota é: ", nota {aqui, o valor impresso
será 10}
nota ← 8 {variável armazena valor numérico 8}
escreva "O valor da variável nota é: ", nota {aqui, o valor impresso
será 8, ou seja, sobrescrevemos o valor 10}
.....
```

Isso parece bem lógico, pois estamos escrevendo na mesma posição de memória do computador.

Lembre-se que, sempre que criarmos uma variável, estaremos criando um espaço na memória do computador para armazenar dados. É um endereço na qual o computador se referencia para manipular os dados em questão.

Como queremos armazenar vários valores numéricos, precisamos criar várias posições de memória sob o nome de uma mesma variável.

O que devemos especificar é quantos valores queremos armazenar, ou seja, quantas posições de memória queremos alocar para armazenar esses números.

Vejamos o exemplo para armazenar as notas de um aluno conforme o algoritmo anterior. Queremos armazenar seis valores diferentes em seis posições de memória diferentes:

8.0	10.0	9.0	10.0	8.5	10.0
-----	------	-----	------	-----	------

Devemos criar 6 posições de memórias para armazenar esses valores. Como solução, podemos criar 6 variáveis ou criar um vetor com 6 posições de memória.


Esse esquema representa um vetor do tipo numérico de 6 posições, ou seja, 6 endereços de memória consecutivos alocados no computador que podem armazenar 6 valores numéricos diferentes.

Cada quadradinho representa uma posição de memória, onde podem ser armazenados os valores numéricos.

O número de posições que queremos criar é especificado na declaração.

Em resumo: um vetor é prático quando precisamos manipular um conjunto de dados do mesmo tipo sem que seja necessário declarar muitas variáveis. Por exemplo: O registro de 26 livros e seus respectivos preços; o registro de notas de 13 avaliações de um aluno etc.

Mas como criar um vetor? É muito simples. Especificamos o nome do vetor e o número de posições da memória que queremos alocar. Cada posição de memória pode armazenar um valor diferente dos demais.

Sintaxe do vetor 	<code><nome do vetor>: <u>vetor</u> [tamanho do vetor] <tipo de constante que o vetor poderá conter></code>
---	---

Quando um vetor é declarado, ele se apresenta assim na memória:

Valor 1	Valor 2			Valor n
---------	---------	--	--	---------

notas: vetor[6] numérico {vetor numérico de 6 posições. Pode armazenar até 6 valores numéricos diferentes}

estados: vetor[27] literal {vetor de caracteres de 27 posições. Pode armazenar até 27 caracteres diferentes}

Importante

Alguns autores preferem utilizar uma sintaxe diferente para a criação de vetores, conforme a seguir:

`<nome do vetor>: vetor[<posição inicial do vetor>.. <posição final do vetor>] <tipo de constante que o vetor poderá conter>`

Exemplo

notas: vetor [1..50]: numérico {criamos um vetor que inicia com índice 1 e vai até 50}.

O mesmo vetor poderia ser criado da seguinte maneira:

notas: vetor [0..49]: numérico {criamos um vetor que inicia com índice 0 e vai até 49}.

Qual a melhor maneira? Você pode escolher. Porém, em linguagens de programação de alto nível como C/C++, JAVA etc., os vetores começam sempre com índice 0.

Desde que você faça a conversão correta entre pseudocódigo e uma linguagem de programação de alto nível, não há problemas de qual a maneira que você vai criar seus vetores. O importante é que seja claro e sem quaisquer ambigüidades.

Operação de vetores

Até agora nossa preocupação foi em saber como criar um vetor e saber quando ele é necessário. Sempre que trabalharmos com grandes quantidades de dados, estaremos criando um ou mais vetores.

A sintaxe da criação é bastante simples conforme seção anterior. Mas como iremos trabalhar com um vetor? Por exemplo, como atribuir valores a um vetor? Como recuperar um valor de um vetor? Como realizar operações básicas de adição, subtração etc. de vetores?

Bem, vamos por etapa. Inicialmente vamos inserir valores em cada parte do vetor. Como já foi dito anteriormente, quando um vetor é declarado, são reservados espaços na memória para armazenar as constantes (literal, numérica ou lógica). Porém, como acessar esses espaços? É muito simples, basta indicar a posição que você quer acessar.

Veja um exemplo de um vetor de notas:

notas: vetor[6] numérico

Quando o vetor notas é declarado, ele se apresenta assim na memória:



Cada quadrado, representando uma posição de memória, é uma posição do vetor declarado. Chamamos essa posição de índice do vetor. O primeiro quadrado (posição inicial), dizemos que tem índice 0 e o acesso a essa posição se dá através do nome do vetor seguido do abre colchete '[', do valor 0, seguido do fecha colchete ']'. Para as posições seguintes, temos os índices 1, 2, 3,

Para atribuírmos um valor à posição inicial do vetor notas, podemos então fazer: notas[0]<- 8.0

Para atribuírmos um valor à segunda posição do vetor notas, podemos fazer:

notas[1]<- 10.0

Para atribuírmos um valor à terceira posição do vetor notas, podemos fazer:

notas[2]<- 9.0. E assim sucessivamente, até preencher todo o vetor de notas.

Nosso vetor ficaria assim preenchido:

8.0	10.0	9.0			
-----	------	-----	--	--	--

Importante

Perceba que a primeira posição do vetor tem índice 0, a segunda posição tem índice 1, a terceira posição tem índice 2, e assim sucessivamente.

Isso significa que para um vetor de tamanho N, o último índice é N-1.

Por exemplo, um vetor de notas de tamanho 150, declarado da seguinte forma `notas: vetor [150]` numérico tem índices de vão de 0 (posição inicial) até 150-1, ou seja, índice 149. Não existe a posição 150. 150 é o tamanho do vetor que vai de 0 até 149.

Voltando ao nosso exemplo, onde queremos imprimir as notas do aluno, além da sua média, podemos armazenar os valores (notas digitadas) em um vetor de notas.

Cada nota digitada será armazenada em uma posição específica. Perceba que temos 6 notas como entrada, o que sugere um vetor de tamanho 6, com índices variando de 0 até 5.

início

conta, media, soma: numérico

notas: vetor[6] numérico {vetor para armazenar as notas digitadas}

soma \leftarrow 0 {inicializa variável soma com o valor 0}

para conta de 0 até 5 passo 1 faça {laço de repetição}

escreva "Entre com a nota: "

leia notas[conta] {leitura das notas. Perceba que o que aparece entre colchetes é a variável conta.

Dentro do laço de repetição, a variável conta vai sendo incrementada de 1, sendo seu valor inicial de 0 e o final de 5, conforme o comando para.

Para o primeiro laço de repetição, a variável conta tem valor 0. Sendo assim, o comando leia notas[conta] está lendo notas na posição 0, ou seja, leia notas[0]. Para o segundo laço de repetição, a variável conta tem valor 1. Sendo assim, o comando leia notas[conta] está lendo notas na posição 1, ou seja, leia notas[1], e assim sucessivamente até atingir o valor 5, última posição do vetor. }

soma \leftarrow soma + notas[conta] {soma das notas digitadas}

fim-para

{para imprimir as notas, basta ler o vetor notas da posição 0 até a posição 5. Fazemos com o laço de repetição novamente}

para conta de 0 até 5 passo 1 faça {laço de repetição}

escreva "Nota: ", notas[conta]

fim-para

media \leftarrow soma/6 {calcula a média final}

escreva "A média final é ", media {mostra o resultado na tela}

fim

Dica: para trabalhar com vetores, sempre precisamos especificar o seu tamanho inicial. Sendo assim, a forma mais usual e fácil para escrita/leitura de valores para/de vetores pode ser feita através do comando para/ m-para. Você consegue saber por quê?

Tudo, agora, não passa de operações simples, como se estivéssemos manipulando variáveis independentes.

Por exemplo, vamos criar um vetor de números pares e um vetor de números ímpares: a diferença entre eles será armazenada em um terceiro vetor. Vamos assumir os 50 primeiros valores numéricos, ou seja, de 0 até 49.

```
Início
{declaração das variáveis}
pares: vetor[25] numérico {armazena números pares com 25 posições,
pois se são os 50 primeiros números positivos, temos apenas 25 pares}
Impares: vetor[25] numérico {armazena números ímpares}
subtracao: vetor[25] numérico {armazena a diferença entre números
pares e ímpares}
j,i: numérico {variáveis utilizadas para realizar o laço de repetição}

{vamos armazenar os números pares no vetor pares}
j ← 0 {será o contador do vetor pares e vetor ímpares}
para i de 0 até 49 passo 2 faça
    {armazena 0,2,4,6.....48. Perceba que o passo do laço de
    repetição é de 2 em 2.
    Então o valor da variável i pula de 2 em 2. Mas no vetor pares
    precisamos guardar em posições sequenciais. Vamos criar
    uma outra variável para contar os vetores pares (j).}
    pares[j] ← i
    j ← j + 1
fim-para

{vamos armazenar os números ímpares no vetor ímpares}
j ← 0 {inicializar novamente, para iniciar o vetor ímpares de 0.}
para i de 1 até 49 passo 2 faça
    Impares[j] ← i {armazena 1,3,5,7.....49. Perceba que o passo
    do laço de repetição é de 2 em 2, mas a contagem começa
    com 1. O contador j irá guardar o valor de i na posição correta
    em sequência.}
    j ← j + 1
fim-para

{vamos realizar as subtrações de cada posição dos vetores pares e
ímpares e armazenar o resultado no vetor subtracao. A estrutura
repetição para-faça inicia em 0 e vai até 25, pois cada vetor tem apenas
25 elementos (25 pares e 25 ímpares) dentre os 50 primeiros números
0-49}
para i de 0 até 24 passo 1 faça
    subtracao[i] = pares[i] - Impares[i]
fim-para

{como resultado de saída, vamos imprimir os vetores}
para i de 0 até 24 passo 1 faça
    escreva "Números Pares: ", pares[i]
    escreva "Números Ímpares: ", Impares[i]
    escreva "Diferença Pares - Ímpares: ", subtracao[i]
fim-para

fim
```

Algoritmos com manipulação de Vetores

A melhor maneira de aprender a programar é programando. Continua valendo a idéia: A melhor maneira de aprender a utilizar vetores é construindo algoritmos com vetores.

Nesta seção, mostraremos algumas aplicações que manuseiam vetores.

Para esses exemplos, vamos definir as variáveis com nomes maiúsculos apenas por questões de estética. Por padrão, as variáveis são todas com letras minúsculas, mas isso para as linguagens de programação de alto nível como C/C++, JAVA, Visual etc. Para o pseudocódigo, podemos representar por letras maiúsculas, sem perda de padronização.

-
1. Ler um vetor de 50 números e montar outro vetor com os valores do primeiro multiplicados por 3.

Pseudocódigo:

```
início  
  VET1,VET2 : vetor [50] numérico  
  CONTADOR : numérico  
  para CONTADOR de 0 até 49 faça  
    leia "Digite um número: ",VET1[CONTADOR]  
    VET2[CONTADOR] ← VET1[CONTADOR] * 3  
  fim-para  
fim
```

2. Um armazém contém 400 produtos e para cada tipo de produto existe um código. Faça um algoritmo para ler o código do produto e a quantidade em estoque. Depois, monte dois vetores para armazenar respectivamente os códigos das mercadorias e a quantidade dos produtos.

Pseudocódigo:

```
início  
  {declaração das variáveis}  
  CODIGOPRODUTO : vetor [400] literal  
  QUANTIDADE : vetor [400] numérico  
  CONTADOR : numérico  
  para CONTADOR de 0 até 399 faça  
    {leitura de 400 códigos de produtos e a quantidade em estoque  
    de cada um}  
    escreva "Digite o código do produto: "  
    leia CODIGOPRODUTO[CONTADOR]  
    escreva "Digite a quantidade do produto em estoque: "  
    leia QUANTIDADE[CONTADOR]  
  fim-para  
fim
```

3. Ler um vetor contendo 100 números, que correspondem a matrículas de alunos. Ler cinco números e imprimir uma mensagem informando se eles estão ou não presentes no vetor.

Pseudocódigo:

início

{declaração de variáveis}

ALUNOS : vetor[100] numérico {lembre que o tamanho do vetor é 100, mas ele vai de 0 até 99}

POS, PESQUISADO, CONT :numérico

{laço de repetição para preencher o vetor ALUNOS}

para POS de 0 até 99 faça

escreva "Digite o código de matrícula de um aluno: "

leia ALUNOS[POS]

fim-para

{5 entradas de dados para verificar se alunos estão cadastrados ou não}

para CONT de 1 até 5 faça

escreva "Digite o número de matrícula a ser pesquisado: "

leia PESQUISADO

POS \leftarrow -1 {inicialmente é atribuído o valor -1 para a variável POS porque na sequência, no comando repita, a variável POS será incrementada de 1 (POS \leftarrow POS + 1). Como queremos começar a ler o vetor ALUNOS a partir da posição 0, no primeiro laço, a variável POS assume valor 0}

repita

POS \leftarrow POS+1

{repete o laço até que o número sendo pesquisado (valor armazenado na variável PESQUISADO) seja igual ao número armazenado no vetor ALUNOS em uma determinada posição POS, ou que a variável POS seja maior que o tamanho do vetor ALUNOS, nesse caso, seja maior que 100}

até que PESQUISADO = ALUNOS[POS] OU POS > 99

se POS > 99 então

{se a variável POS possui um valor superior a 99, ou seja, 100, isso significa que todo o vetor ALUNOS foi lido desde o índice 0 até 99 (100 alunos), mas que não há nenhum número armazenado nesse vetor é igual ao valor armazenado na variável PESQUISADO}

escreva "Número não cadastrado !"

senão

escreva "Número localizado na posição ", POS, " do vetor."

fim-se

fim-para

fim

4. Criar um algoritmo que leia o preço de compra e o preço de venda de 100 mercadorias. O algoritmo deverá imprimir quantas mercadorias proporcionam:
- Lucro menor do que 10%
 - Lucro maior ou igual 10% e menor ou igual a 20%
 - Lucro superior a 20%

Pseudocódigo:

início

{declaração de variáveis}

PRECOCOMPRA: vetor [100] numérico {armazena os preços de compra das mercadorias}

PRECOVENDA: vetor [100] numérico {armazena os preços de venda das mercadorias}

LUCRO, TOTLUCROMENOR10, TOTLUCROMENOR20, TOTLUCROMAIOR20: numérico
{variáveis utilizadas para armazenar o lucro, o lucro menor que 10%, menor do que 20% e maior do que 20%, respectivamente}

i : numérico {variável utilizada como contadora para laços de repetição}

{precisamos inicializar as variáveis contadoras de lucros}

TOTLUCROMENOR10 \leftarrow 0

TOTLUCROMENOR20 \leftarrow 0

TOTLUCROMAIOR20 \leftarrow 0

{entrada de dados}

para i de 0 até 99 passo 1 faça

{vamos especificar o preço de venda e de compra de todas as 100 mercadorias}

escreva "Entre com o preço de compra da mercadoria: "

leia PRECOCOMPRA[i]

escreva "Entre com o preço de venda da mercadoria: "

leia PRECOVENDA[i]

fim-para

{Processamento do algoritmo. Vamos verificar o lucro de cada mercadoria (preço de venda – preço de compra) e verificar as condições de lucro (10%, 20% ou maior do que 20%. Para isso, vamos ter que ler os vetores PRECOVENDA e PRECOCOMPRA novamente)}

para i de 0 até 99 passo 1 faça

LUCRO \leftarrow (PRECOVENDA[i] – PRECOCOMPRA[i]) * 100 / PRECOCOMPRA[i]

{se a diferença entre preço de venda e preço de compra, ou seja, o lucro for menor do que 10, então incrementa a variável contadora de mercadorias com lucros inferior a 10%}

se LUCRO < 10.0 então

TOTLUCROMENOR10 \leftarrow TOTLUCROMENOR10 + 1

senão

{se o lucro não é inferior a 10%, então ele só pode ser superior a 10%. Mas quanto? Precisamos saber se o lucro é menor a 20% ou superior a esse valor. Por isso colocamos outra condição dentro do comando senão. Essa condição verifica se o lucro é inferior ou superior a 20%.}

Síntese

Finalizamos mais uma etapa. Mais um passo da nossa caminhada. Nessa unidade, vimos que um vetor é uma variável que pode armazenar várias constantes do mesmo tipo (homogêneas). Isso permite manipular uma grande quantidade de dados sem a necessidade de declarar várias variáveis.

Para declarar um vetor em pseudocódigo, utilizamos o seguinte comando:

<nome do vetor>: vetor [tamanho do vetor] <tipo de constante que o vetor poderá conter>

Quando um vetor é declarado, ele se apresenta assim na memória:

Valor 1	Valor 2			Valor n
---------	---------	--	--	---------

Cada posição (representado por um quadrado no desenho acima) é uma posição de memória do computador.

Para inserir ou ler valores de um vetor basta especificar seu nome e o índice (posição) que queremos acessar dentro do vetor. Por exemplo: notas [10] 8.5. Sabendo que o índice dos vetores começa com o valor numérico 0.

Quando colocamos notas [10] estamos nos referindo ao índice 10, mas a posição no vetor (representado por um quadrado) é o 9. Poderíamos representar as posições de um vetor de tamanho 5 conforme apresentado abaixo:

Valor 1	Valor 2			Valor n
---------	---------	--	--	---------

[0]	[1]	[2]	[3]	[4]
-----	-----	-----	-----	-----

Podemos perceber que o índice [3] está no quarto quadrado. Tudo isso porque um vetor tem seu início no índice 0, conforme já dito.

Bem, um vetor é unidimensional, ou seja, apresenta apenas uma dimensão ou 1 linha. Poderíamos estar trabalhando com vetores bidimensionais ou vetores que possuem várias colunas e várias linhas. A esses vetores damos o nome de matrizes.

EXERCÍCIOS:

1. Crie um vetor para armazenar 25 valores de temperaturas.
2. Crie um vetor para armazenar 150 alunos de um colégio infantil.
3. Um site na web precisa registrar 2500 produtos cadastrados de um fornecedor. Crie um vetor para representar esses produtos.
4. Criar um algoritmo que realize as reservas de passagem aéreas de uma companhia.

Além da leitura do número de vôos e da quantidade de lugares disponíveis, leia vários pedidos de reserva, constituídos do número da carteira de identidade e do número do vôo desejado. Para cada cliente, verificar se há possibilidade no vôo desejado. Em caso afirmativo, imprimir o número de identidade do cliente e o número do vôo, atualizando o número de lugares disponíveis. Caso contrário, avisar ao cliente a inexistência de lugares.

8 - MANIPULAÇÃO DE MATRIZES

Introdução

Trabalhamos na unidade anterior apenas com vetores unidimensionais, ou seja, variáveis que podem conter diferentes valores de um mesmo tipo em diversas colunas.

Na verdade, o conceito de vetor pode ser representado por uma tabela com 1 linha e várias colunas, onde o tamanho do vetor especifica o número de colunas, como pode ver visto na figura abaixo:

Linha 0	Valor 1	Valor 2	Valor 3	...	Valor n
	Coluna 0	Coluna 1	Coluna 2		Coluna n-1

Perceba na figura acima que temos uma linha, denominada *linha 0*, e várias colunas, denominadas de *coluna 0*, *coluna 1* e assim sucessivamente.

Como vimos na unidade anterior, cada valor de um vetor é armazenado em uma posição de memória, ou seja, em um dos quadrados representados na figura anterior.

Já sabemos também que não precisamos saber o endereço de memória do computador para acessar os elementos do vetor, bastando especificar o índice que queremos acessar.

Por exemplo, para um vetor chamado de NOTAS de tamanho 5, podemos acessar os índices de 0 até 4 (lembre-se de que um vetor sempre começa com índice 0).

Para acessar qualquer posição do vetor, basta especificar o índice de interesse: NOTAS [2] □ 7.5.

Se observarmos o índice que estamos acessando, no exemplo em questão, o índice 2 do vetor NOTAS, podemos representá-lo conforme a figura anterior:

Linha 0			7.5		
	Coluna 0	Coluna 1	Coluna 2		Coluna n-1

Observando a figura, podemos constatar que o índice 2 está na linha 0 e coluna 2. Isso mesmo, você já deve ter percebido que um vetor nada mais é do que uma tabela de 1 (uma) linha, na qual chamamos de linha 0 e várias colunas, onde é armazenado

cada um dos valores, seja ele numérico, alfanumérico ou lógico, e que o índice que especificamos para acessar o vetor é exatamente o número da coluna desse vetor.

Outro exemplo, NOTAS [4] 10.0. Podemos dizer que estamos inserindo no vetor NOTAS o valor 10.0 no índice 4, ou seja, na linha 0 e coluna 4 do vetor.

Mas agora vem a questão: e, se quisermos trabalhar com um vetor que possua várias linhas e várias?

Para um vetor com várias linhas e colunas., damos o nome para essas estruturas de matrizes.

Conceito e declaração de matrizes

Uma matriz nada mais é do que um vetor de 2 dimensões (linhas e colunas) capaz de armazenar variáveis do mesmo tipo (numérica, literal ou lógica).

Pode ser representada por uma tabela, conforme você pode ver a seguir:

Cada quadrado representa uma posição de memória onde podem ser armazenadas as variáveis, de maneira idêntica aos vetores.

Porém, os vetores são unidimensionais (apenas colunas) e as matrizes são bidimensionais (2 dimensões: linhas e colunas). Agora teremos não somente a linha 0, mas também a linha 1, a linha 2 e assim sucessivamente.

O número de posições que queremos criar é especificado na declaração, de forma similar a dos vetores.

Uma matriz é prática quando precisamos manipular um conjunto de dados do mesmo tipo, sem que seja necessário declarar muitas variáveis e precisamos fazer relações de 2 variáveis.

Queremos registrar 4 temperaturas de três dias da semana (segunda-feira, terça-feira e quarta-feira).

Nesse caso, temos duas variáveis: temperatura e dias da semana. A melhor maneira de representar isso é através de uma tabela, mostrando nas linhas os dias da semana e, nas colunas, as temperaturas medidas.

Veja a tabela a seguir:

Dias da semana	Temperatura 1	Temperatura 2	Temperatura 3	Temperatura 4
Segunda	27	29	30	24
Terça	25	27	28	22
Quarta	21	23	25	20

Se olharmos para a tabela anterior, podemos saber, por exemplo, que na terça-feira, a 4ª temperatura medida foi 22°C e que na quarta-feira a 2ª temperatura medida foi de 23°C.

Estamos fazendo uma correspondência entre a variável Temperatura e o Dia.

Nesse caso, para determinar a temperatura em uma determinada hora de um dia.


Para representar essa correspondência é que utilizamos o conceito de matrizes.

Agora teremos uma estrutura de linhas e colunas e não apenas de colunas como eram os vetores.

Para criar uma matriz, deve-se especificar o nome da matriz seguido do número de linhas e colunas que a matriz conterá, além do tipo de variável que será armazenada.

À exemplo de vetores, as matrizes só podem armazenar dados do mesmo tipo.

Por exemplo: somente dados numéricos ou somente dados literais ou somente dados lógicos. Não podemos ter em uma matriz dados numéricos e literais ao mesmo tempo.

Sintaxe da matriz 	<nome da matriz>: <u>matriz</u> [número de linhas][número de colunas] <tipo de constante que o vetor poderá conter>
--	--

Exemplos:

notas_de_alunos: matriz[6][4] numérico

{matriz que possui 6 linhas – numeradas de 0 até 5 e 4 colunas, numeradas de 0 até 3. Dizemos que a matriz tem dimensão de 6x4. No total, são 6x4 = 24 posições para armazenar valores numéricos}

matriz_de_alunos: matriz[10][30] literal

{matriz que possui 10 linhas – numeradas de 0 até 9 e 30 colunas, numeradas de 0 até 29. Dizemos que a matriz tem dimensão de 10x30. No total, são 10x30 = 300 posições para armazenar valores literais}

Assim como na criação de vetores, para a criação de matrizes não precisamos especificar os endereços de memória do computador. Isso é feito automaticamente, sem que sequer saibamos onde os valores estão armazenados na memória. A única coisa que sabemos é que é alocado um espaço para armazenar esses valores e que são armazenados em endereços de memória sequenciais. Para acessar esses endereços ou os valores armazenados nesses endereços é que é um pouco diferente dos vetores. Veremos isso na seção 2.

Operação de matrizes

Até agora nossa preocupação foi em saber como criar uma matriz e saber quando ela é necessária. Sempre que trabalharmos com grandes quantidades de dados e precisamos relacionar duas ou mais variáveis, estaremos criando uma ou mais matrizes. A sintaxe da criação é bastante simples conforme seção anterior.

Mas como iremos trabalhar com uma matriz? Por exemplo, como atribuir valores a uma matriz? Como recuperar um valor de uma matriz?

Para acessar as posições de uma matriz, basta indicar a linha e a coluna desejadas.

Exemplo: NOTAS.

NOTAS: matriz [6][4] numérico

1. A Matriz NOTAS é uma matriz de dimensão 6x4, ou seja, 6 linhas por 4 colunas.
2. Para armazenar uma constante numérica na matriz NOTAS, é necessário identificar a linha e coluna que queremos acessar. Isso é feito da seguinte forma: nome da matriz [índice da linha][índice da coluna], lembrando que os índices começam sempre com o valor 0.
Por exemplo, a 3ª (terceira) linha da matriz tem índice 2, isso por que a 1ª (primeira) linha tem o índice 0, a 2ª (segunda) linha tem índice 1 e a 3ª (terceira) o índice 2.
3. Para armazenar a constante numérica 10 na 3ª (terceira) linha (linha 2) da 2ª (segunda) coluna (coluna 1), utilizamos os índices da matriz NOTAS, nesse caso, o índice 2 para linha e o índice 1 para coluna. Assim, NOTAS [2][1] 10. Veja com fica a matriz.

	10.0		

Todo o processo de manipulação agora se torna simples, bastando especificar a posição onde estaremos armazenando os valores, ou seja, basta especificar o nome da matriz e, a seguir, entre colchetes, o índice que representa a linha e, depois, também entre colchetes, o índice que representa a coluna.

Dica: para trabalhar com matrizes, sempre precisamos especificar a sua dimensão, ou seja, o número de linha pelo número de colunas. Sendo assim, a forma mais usual e fácil para escrita/leitura de valores para/de matrizes pode ser feita através do comando para/m-para. Identifique por quê!

1. Montar uma matriz nas dimensões 4x5 e imprimir a soma das linhas e colunas.

Pseudocódigo:

início

MAT : matriz [4][5] numérico {matriz de dimensão 4x5. 4 linhas e 5 colunas}

SOMALINHA, SOMACOLUNA, LINHA, COLUNA : numérico

{Aqui um fato importante. Para percorrer um vetor desde o seu início até o seu final, utilizamos o comando para/faça/fim-para. Agora temos que percorrer toda a matriz, ou seja, todas as linhas e colunas. O processo é bem simples. Para percorrer todas as colunas de uma linha de uma matriz continuamos utilizando o comando para/faça/fim-para. Assim que todas as colunas de uma linha são lidas ou acessadas, passa-se para a próxima linha. Novamente, para essa nova linha, todas as colunas são acessadas, e assim sucessivamente. Podemos perceber que para cada incremento da linha devemos ler ou acessar todas as colunas daquela linha. Isso sugere dois laços de repetição: 1 para pular as linhas e outro para pular as colunas. Dessa forma, elaboramos 2 laços de repetição, um para linha com a variável LINHA sendo incrementada de 0 até 3 (4 linhas no total), e um laço para coluna com a variável COLUNA sendo incrementada de 0 até 4 (5 colunas no total). Para cada repetição do laço dentro da linha executamos 5 repetições para as colunas, percorrendo assim toda a matriz}

para LINHA de 0 até 3 passo 1 faça

para COLUNA de 0 até 4 passo 1 faça

escreva "Digite um número: "

leia MAT[LINHA][COLUNA]

fim-para

fim-para

{Processamento do Algoritmo. Soma das linhas. A explicação desta parte está após o fim-para do primeiro laço de repetição}

```
para LINHA de 0 até 3 passo 1 faça
  SOMALINHA ← 0
  para COLUNA de 0 até 4 passo 1 faça
    SOMALINHA ← SOMALINHA + MAT[LINHA][COLUNA]
  fim-para
  escreva "Total da linha", LINHA,":", SOMALINHA
fim-para
```

{Explicando a lógica anterior: Nossa matriz possui 4 linhas e 5 colunas. Sendo assim, vamos percorrer as linhas de 0 a 3 e as colunas de 0 a 4 (São os índices da matriz. Para o primeiro laço de repetição para LINHA de 0 até 3 passo 1 faça, a variável LINHA é incrementada de 0 até 3. Inicialmente ela tem valor numérico 0. O primeiro comando dentro do laço é inicializar a variável SOMALINHA igual a 0. A variável SOMALINHA armazenará a soma de todas os valores numéricos de cada linha. Sabendo que estamos na linha 0, devemos percorrer cada coluna dessa linha. Isso é feito pelo laço de repetição seguinte para COLUNA de 0 até 4 passo 1 faça. Quando o programa entra nesse segundo laço, a variável COLUNA é incrementada com passo 1, de 0 até 4, executando os comandos que estão dentro da estrutura para/ faça/ fim-para, no nosso caso, apenas o comando $SOMALINHA \leftarrow SOMALINHA + MAT[LINHA][COLUNA]$. Quando o segundo laço de repetição é finalizado, o programa executa os comandos seguintes ao fim-para, ou seja, executa o comando de impressão na tela escreva "Total da linha", LINHA,":", SOMALINHA. Ao encontrar o fim-para do primeiro laço de repetição, o programa incrementa a variável LINHA de 1 no comando para LINHA de 0 até 3 passo 1 faça passando o valor da variável LINHA para o valor 1, e executa todos os comandos dentro da estrutura para/ faça/ fim-para novamente. Como SOMALINHA armazena o valor numérico da soma dos valores da linha anterior, a variável é reinicializada com valor 0. Um ótimo exercício é montar uma matriz e executar o algoritmo passo a passo. Essa é uma atividade para você fazer logo a seguir.}

```

{Processamento do Algoritmo. Soma das colunas}
para COLUNA de 0 até 4 passo 1 faça
    SOMACOLUNA ← 0
    para LINHA de 0 até 3 passo 1 faça
        SOMACOLUNA ← SOMACOLUNA + MAT[LINHA][COLUNA]
    fim-para
    escreva "Total da coluna", COLUNA, ":", SOMACOLUNA
fim-para
fim

```

Algoritmos com manipulação de matrizes

Para montar algoritmos utilizando estruturas de matrizes, são utilizados os mesmos comandos como leia, escreva, para/faça/fim-para, entre outros.

1. Monte uma matriz para quando o usuário informar um número correspondente a um mês, o algoritmo imprima o nome do mês indicado em português, a abreviação e o nome do mês em inglês. Por exemplo, o usuário digita o número 4 e o algoritmo é ativado para imprimir: 4Abril, Abr, April.

Pseudocódigo:

```

início
    MESES : matriz [12][3] literal
    LINHA, NUM : numérico
    para LINHA de 1 até 12 passo 1 faça
        escreva "Digite o nome do ", LINHA, "º mês:"
        leia MESES[LINHA][1]
        escreva "Digite a abreviação do mês de ", MESES[LINHA][1], ":"
        leia MESES[LINHA][2]
        escreva "Digite o nome em inglês do mês ", MESES[LINHA][1], ":"
        leia MESES[LINHA][3]
    fim-para
    leia "Digite o número do mês a ser consultado: ", NUM
    escreva NUM, " ", MESES[NUM][1], " ", MESES[NUM][2], " ", MESES[NUM][3]
fim

```

2. Uma Floricultura conhecedora de sua clientela gostaria de fazer um algoritmo que pudesse controlar via Web sempre um estoque mínimo de determinadas plantas, pois todo o dia, pela manhã, o dono faz novas aquisições.

Criar um algoritmo que deixe cadastrar 50 tipos de plantas e nunca deixa o estoque car abaixo do ideal. O algoritmo será utilizado para construir um programa na página da empresa.

Pseudocódigo:

início

{vamos montar uma matriz de 50 linhas e 3 colunas. As 50 linhas servem para cadastrar todos os produtos e as três colunas servem para especificar a quantidade de produtos em estoque, a quantidade desejada e o resultado da diferença entre a quantidade em estoque e a quantidade desejada respectivamente.}

PRODUTOS: matriz [50][3] : numérico

NOME: vetor [50] literal

i: numérico

para i de 0 até 49 passo 1 faça

escreva "Entre com o nome do produto: "

leia NOME [i]

escreva "Entre com a quantidade em estoque: "

leia PRODUTOS[i][0]

escreva "Entre com a quantidade desejada: "

leia PRODUTOS[i][1]

se PRODUTOS[i][0] < PRODUTOS[i][1] então

 [PRODUTOS [i][2] \leftarrow PRODUTOS[i][1]
 - PRODUTOS[i][0]

senão

 [PRODUTOS [i][2] \leftarrow 0

fim-se

fim-para

{dados de saída do algoritmo}

escreva "Total de Compras: "

para i de 0 até 49 passo 1 faça

escreva "Produto: ", NOME [i], " Qtde = ", PRODUTOS[i][2]

fim-para

Síntese

Nesta unidade, você viu que para relacionar duas ou mais variáveis precisamos manipular matrizes.

Diferentemente de vetores, que são unidimensionais, as matrizes são bidimensionais, possuindo linhas e colunas.

Podemos ter matrizes com mais dimensões, mas nesta unidade trabalhamos apenas com 2.

Similarmente aos vetores, as matrizes só podem armazenar dados do mesmo tipo, ou seja, quando definimos uma matriz, especificamos que tipo de variável a mesma vai armazenar (numérico, literal ou lógico).

A sintaxe em pseudocódigo para definir uma matriz é a seguinte:

<nome da matriz>: matriz [número de linhas][número de colunas] <tipo de constante que o vetor poderá conter>

Quando criamos uma matriz, o computador reserva um espaço na memória para armazenar Linhas x Colunas valores. É o que chamamos de dimensão da matriz. Para acessar qualquer elemento da matriz, basta especificar o índice da linha e coluna, lembrando sempre que os índices começam com o valor numérico 0. Por exemplo, uma matriz NOTAS de dimensão 5x 6 começa em [0][0] e termina em [4][5]. Podemos acessar qualquer posição da matriz desde que esteja dentro das dimensões especificadas: NOTAS[3][2] 8.5. Estamos acessando a quarta linha e a terceira coluna da matriz.

Por fim, viu que para percorrer uma matriz de ponta a ponta, precisamos de dois laços de repetição, um sendo utilizado para percorrer as colunas de cada linha e o outro para percorrer as linhas da matriz.

EXERCÍCIOS

1. A distância em quilômetros entre algumas capitais é mostrada no quadro a seguir.

Suponha que você tenha sido contratado por uma empresa, que vende mapas, para montar um programa (algoritmo) que leia as capitais e suas respectivas distâncias e também deverá imprimir a distância entre duas capitais solicitadas por um usuário. Esse será um programa que poderá ser acessado via Web.

Tabela mostrando as distâncias entre as capitais:

	1	2	2		27
1	0	23	45		
2	23	0	10		
27	110	50	66	72	0

9 - MANIPULAÇÃO DE REGISTROS

Introdução

Quantas vezes você já preencheu fichas de cadastros, seja em hotéis, em videolocadoras, para propostas de consórcios ou ainda fichas de matrícula escolar?

Nessas fichas, certamente você precisou entrar com dados como: seu nome, sua idade, seu telefone de contato, endereço etc.

Nesses casos, estamos trabalhando com dados de diferentes tipos.

Por exemplo: nome é uma variável literal, idade é uma variável numérica, informação se possui ou não veículo pode ser um valor lógico.

Percebeu que estamos entrando em um mundo onde as coisas não são tão homogêneas assim, ou seja, nem tudo que estamos trabalhando são apenas variáveis numéricas ou literais ou ainda lógicos? Há uma mistura de tipos de dados que devemos trabalhar.

Mas, o que tem haver isso com lógica de programação? Não poderíamos criar variáveis independentes, ou seja, cada informação armazenada em um local diferente como temos feito até agora? Poderíamos criar, por exemplo, as variáveis em pseudocódigo representando nossa ficha cadastral. Acompanhe a seguir.

início

{Declaração de variáveis}

NOME, ESTADOCIVIL, ENDERECO, BAIRRO: literal

IDADE, DATANASC, NUMERO, CEP, FONE: numérico

....

fim

Se fôssemos preencher apenas uma ficha cadastral, isso resolveria nossos problemas. Mas vamos supor que queremos cadastrar 50 hóspedes de um hotel. Certamente, tendo visto e estudado as unidades sobre vetores e matrizes, você responderia que a solução continua simples. Basta fazer, de todas as variáveis criadas anteriormente, vetores de dimensão 50.

NOME: vetor [50] literal; IDADE: vetor[50] numérico, e assim por diante. Isso também resolveria nossos problemas. Se você pensou assim, pensou certo. Parabéns, por que mostra que você conseguiu entender os conceitos das unidades anteriores.

Mas o que há de novo, então?

Quando criamos variáveis independentes, o computador irá colocá-las em qualquer endereço de memória reservado para essas ocasiões. Ele não vai se preocupar em colocar em uma certa ordem que muitas vezes são necessárias por quesitos de velocidade de execução do programa. Imagine sua ficha cadastral onde o nome está em uma folha, o endereço está duas folhas a seguir, depois volta uma folha para preencher a idade e assim sucessivamente.

Parece desorganizado você não acha? Além do tempo de preenchimento que será bem maior. Ou seja, se estamos criando variáveis para representar nossas fichas cadastrais, seriam interessantes que todas elas fossem declaradas próximas uma das outras. Isso faria com que o desempenho do programa fosse melhor.

Pode parecer estranho, mas para um programa com muitas informações e dados para manipular, isso pode fazer a diferença entre um programa bom e um ruim.

Pois bem, no mundo da lógica de programação, podemos criar estruturas de dados heterogêneas, ou seja, capaz de armazenar variáveis de tipos diferentes, de forma que as mesmas estejam declaradas próximas umas das outras no que diz respeito à posição de memória do computador. São os chamados registros.

Com esse tipo de estrutura, podemos declarar múltiplas variáveis de diferentes tipos, todas organizadas umas próximas das outras, de forma similar a uma ficha cadastral. Na seção seguinte, vamos definir essa estrutura e vamos aprender a como declará-las.

Quando trabalhamos com vetores e matrizes nas unidades anteriores, os dados armazenados nessas estruturas devem ser homogêneos, ou seja, tudo do mesmo tipo. Não podemos utilizar nem vetores e nem matrizes para armazenar dados de tipos diferentes.

Mas, se quisermos trabalhar com dados do tipo literal e numérico juntamente, isso é possível?

Posso dizer que sim. Nesta unidade você vai aprender como trabalhar com dados de tipos diferentes, ou seja, vai conhecer as estruturas ou registros de dados.

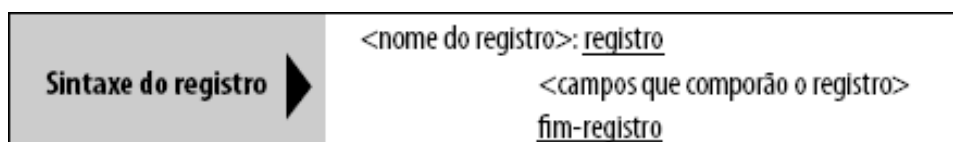
Conceito e declaração de registros

Mas, se quiser trabalhar com dados do tipo literal e numérico juntamente, isso é possível?

Em lógica de programação um registro é um recurso que permite a criação de diferentes tipos de variáveis em um mesmo bloco de memória do computador. Quando criamos um registro, criamos um espaço na memória do computador que permite armazenar dados heterogêneos, ou seja, constantes de vários tipos. É como se fosse uma ficha de dados, organizada de forma que os dados estão próximos um dos outros dentro da memória do computador. Uma das grandes vantagens disso, além da organização, é a velocidade de acesso às informações ali contidas.

Para lembrar!

Diferentemente de vetores e matrizes que só podem armazenar dados do mesmo tipo, os registros são estrutura ou recursos que permitem armazenar constantes de diferentes tipos.



onde <campos que comporão o registro> são todas as variáveis que irão compor uma ficha de dados, ou seja, NOME, IDADE, CPF, ENDERECO etc.

Vamos imaginar uma ficha cadastral que chamaremos de FICHA com as seguintes informações a serem preenchidas por um cliente de um hotel:

Nome, estado civil, endereço, bairro, cidade, estado, e-mail, idade, telefone, número.

Utilizando a sintaxe de criação de registro, temos:

```
FICHA: registro  
      NOME, ESTADOCIVIL, ENDERECO, BAIRRO, CIDADE, ESTADO,  
EMAIL: literal  
      IDADE, TELEFONE, NUMERO: numérico  
      fim-registro
```

Bem, você já deve estar perguntando. O que mudou além da palavra reservada registro e fim-registro?

Pois bem, conforme dito anteriormente, quando declaramos a FICHA anterior como sendo registro, uma parte da memória do computador é reservada e nela são inseridas as variáveis declaradas entre os comandos registro e fim-registro.

Portanto, elas ocupam o mesmo bloco de memória, tornando o desempenho do programa mais rápido. Agora, por exemplo, NOME e IDADE estão num mesmo bloco de memória. Seria análogo a ter nome e idade em uma mesma ficha cadastral. Outra diferença importante é a questão de como acessamos as variáveis declaradas dentro de uma estrutura.

Como podemos ler e escrever nessas variáveis?

A resposta é bastante simples. Basta especificarmos o nome do registro criado, seguido de um ponto (.) e o nome da variável.

Por exemplo, para acessar a variável NOME, precisamos especificar o nome do registro do qual ela pertence. Sendo assim, FICHA.NOME.

Observe que a variável NOME não é uma variável independente qualquer. Ela pertence ao registro FICHA. Por isso, precisamos preceder o nome da variável NOME com o nome do registro FICHA seguido de um ponto (.).

Assim como na criação de vetores e matrizes, para a criação de registros não precisamos especificar os endereços de memória do computador. Isso é feito automaticamente, sem que sequer saibamos onde os valores estão armazenados na memória. A única coisa que sabemos é que é alocado um espaço para armazenar esses valores e que são armazenados em blocos de memória.

Operação com registros

Novamente, você já deve ter percebido que não criamos qualquer outra lógica de controle além das já estudadas. É mais uma oportunidade de mostrar a você a importância dos comandos básicos de lógica de programação.

Nesta seção, vamos mostrar uma aplicação típica de registros: o conceito de fichas cadastrais. A figura a seguir mostra uma possível ficha cadastral:

Nome:	Idade:
Estado Civil:	
Endereço:	
Número:	
Bairro:	Cidade:
Estado:	Telefone:

Observando a figura anterior, podemos verificar que temos variáveis literais e numéricas.

Vamos classificá-las inicialmente: Nome, Estado Civil, Endereço, Bairro, Cidade e Estado são variáveis literais. Idade, Número e Telefone de Contato são variáveis numéricas. Observe que poderíamos colocar Telefone de Contato como literal também.

Se quiséssemos representar um número telefônico por 278-8080, isso é um literal e não um valor numérico. Mas para nosso exemplo, vamos considerar o Telefone de Contato como sendo numérico. Para o telefone especificado devemos colocar então como sendo 2788080.

Criaremos um registro para a ficha cadastral. Iremos chamar esse registro de FICHA. Sendo assim, em pseudocódigo, temos:

```
FICHA: registro  
      {variáveis da ficha}  
      fim-registro
```

Agora vamos definir as variáveis. Iremos especificar 6 variáveis literais e três variáveis numéricas. Colocaremos os nomes das variáveis de forma a não deixar qualquer tipo de dúvida em relação às constantes que serão armazenadas. Dessa forma, nosso registro será conforme a seguir:

```
FICHA: registro  
      {variáveis da ficha}  
      NOME, ESTADOCIVIL, ENDERECO, BAIRRO, CIDADE, ESTADO: literal  
      IDADE, NUMERO, TELEFONE: numérico  
      fim-registro
```

Isso é tudo. Nossa ficha já está montada. Basta utilizá-la agora acessando as variáveis do registro. O algoritmo a seguir, mostra um exemplo completo.

Pseudocódigo:

início

{declaração do registro FICHA}

FICHA: registro

{variáveis da ficha}

NOME, ESTADOCIVIL, ENDERECO, BAIRRO, CIDADE, ESTADO:

literal

IDADE, NUMERO, TELEFONE: numérico

fim-registro

{entrada de dados}

escreva "Nome: "

leia FICHA.NOME

escreva "Estado Civil: "

leia FICHA.ESTADOCIVIL

escreva "Idade: "

leia FICHA.IDADE

escreva "Endereço: "

leia FICHA.ENDERECO

escreva "Bairro: "

leia FICHA.BAIRRO

escreva "Número: "

leia FICHA.NUMERO

escreva "Cidade: "

leia FICHA.CIDADE

escreva "Estado: "

leia FICHA.ESTADO

escreva "Telefone de Contato: "

leia FICHA.TELEFONE

fim

No algoritmo anterior não mostramos nenhuma saída. O exemplo foi apenas para mostrar como trabalhar com criar um registro e como acessar suas variáveis.

Criando novos tipos de variáveis

Muitas vezes, os tipos básicos de variáveis que utilizamos até agora (numérico, literal e lógico) não são suficientes para resolver um algoritmo.

Seria interessante que pudéssemos criar tipos definidos pelo usuário. Por exemplo, um tipo de variável que armazenasse, ao mesmo tempo, um valor literal, um valor numérico e um valor lógico.

Isso é possível sim, graças à estrutura de registros que acabamos de estudar. Podemos fazer de um registro criado um tipo de variável.

Por exemplo, além de termos os tipos básicos como o numérico, o literal e o lógico, poderíamos ter também o tipo FICHA. FICHA na verdade, é um registro criado pelo programador. Chamamos isso de tipo definido pelo usuário.

Além de podermos criar variáveis, podemos criar agora, também tipos de variáveis.

Mas como posso fazer isso e para que serve?

Bem, para criar um tipo registro, em pseudocódigo, é necessário apenas colocar a palavra reservada tipo antes do nome do registro, ficando assim sua sintaxe:

```
tipo <nome do registro> = registro  
    <campos que comporão o registro>  
    fim-registro
```

O comando tipo serve para criar novos tipos de variáveis a partir dos tipos básicos.

Por exemplo, no exercício de para manipular uma ficha cadastral colocado anteriormente, poderíamos fazer do registro FICHA um tipo e declarar variáveis daquele tipo. A sintaxe caria conforme a seguir:

```
tipo FICHA = registro  
    {variáveis da ficha}  
    NOME, ESTADOCIVIL, ENDERECO, BAIRRO, CIDADE, ESTADO: literal  
    IDADE, NUMERO, TELEFONE: numérico  
    fim-registro
```

Para criar uma variável do tipo FICHA, criamos da mesma forma que criamos as variáveis dos tipos primitivos. Por exemplo, ficha1: FICHA, onde ficha1 é agora uma variável do tipo FICHA. Passamos a acessar as variáveis do registro FICHA a partir da variável ficha1.

Exemplo: cha1.NOME, cha1.ENDERECO e assim por diante.

Observe que FICHA é um tipo definido pelo programador e que FICHA1 é uma variável do tipo FICHA que ocupa um espaço na memória para armazenar 6 variáveis literais e 3 variáveis numéricas, conforme nosso exemplo.

Assim, veja parte do algoritmo anterior atualizado:

início

{declaração do registro FICHA}

tipo FICHA = registro

{variáveis da ficha}

NOME, ESTADOCIVIL, ENDERECO, BAIRRO, CIDADE, ESTADO: literal

IDADE, NUMERO, TELEFONE: numérico

fim-registro

ficha1 : FICHA {declaração da variável do tipo FICHA. Observe que o tipo FICHA é definido antes de ser utilizado. Isso é obrigatório, tendo em vista que o programa precisa saber o que é FICHA. Como definimos FICHA como sendo um registro, ao especificar ficha1 como sendo do tipo FICHA, nenhum problema será encontrado}

escreva "Nome: "

leia ficha1.NOME

escreva "Estado Civil: "

leia ficha1.ESTADOCIVIL

..... {restante do algoritmo}

fim

Mas qual é a utilidade de se criar tipos como no nosso exemplo?

Perceba que criamos apenas uma ficha cadastral de um hóspede.

Se quiséssemos preencher o cadastro para um novo hóspede, bastaria criarmos outra variável, ficha2, por exemplo. Poderíamos criar tantas fichas quanto quisermos.

Considerando o tipo FICHA anteriormente criado, podemos criar várias variáveis:

ficha1, ficha2, ficha3: FICHA.

Cada variável representa uma ficha para cadastrar um hóspede.

É como se tivéssemos 3 fichas cadastrais para serem preenchidas na mão.

Para acessar o nome de ficha1, apenas colocaríamos ficha1. NOME.

Para acessar o nome de ficha2, ficha2. NOME, e para ficha3, ficha3. NOME.

Como cada variável, nesse caso, ficha1, ficha2 e ficha3 são três blocos de memórias independentes, é como se tivéssemos três variáveis independentes em locais de memória diferentes, porém, cada uma delas contendo seis variáveis literais e três numéricas, conforme você já leu.

Criando um conjunto de registros

Precisamos incrementar nosso algoritmo anterior. Ele permite o cadastramento de 5 hóspedes apenas. Nosso hotel tem 100 quartos. Podemos admitir até 100 hóspedes, certo? Devemos criar 100 fichas cadastrais. E agora? Você terá problemas se tiver que criar 100 variáveis do tipo FICHA.

Lembra para que sirvam os vetores? Será que você deduziu que podemos utilizar vetores para esse caso?

Vamos por analogia: se estamos precisando armazenar 100 valores numéricos, criamos um vetor do tipo numérico; se precisamos armazenar 100 nomes de clientes, criamos um vetor do tipo literal. Raciocinando da mesma maneira, se precisarmos armazenar 100 fichas de clientes, criamos um vetor de FICHA (tipo definido por nós). Na sintaxe de pseudocódigo teríamos: `fichas: vetor [100] FICHA`. Agora, definimos um vetor de 100 posições chamado de `fichas`, onde cada posição (quadrado) tem uma variável do tipo FICHA. Veja a seguir:

Ficha 1	Ficha 2	Ficha 3	...	Ficha 99
---------	---------	---------	-----	----------

Ficha1 está na posição 0 do vetor `fichas`, Ficha2 está na posição 1 do vetor `fichas`, e assim sucessivamente.

Mas o que é uma FICHA mesmo? FICHA é um registro que tem os seguintes campos definidos anteriormente:

NOME, ESTADOCIVIL, ENDERECO, BAIRRO, CIDADE,
ESTADO: literal

IDADE, NUMERO, TELEFONE: numérico.

Como podemos acessar os dados de cada ficha dentro do vetor?

De forma similar ao acesso dos dados em vetores, precisamos saber qual a posição que queremos acessar do vetor.

Depois de sabermos qual o índice do vetor, devemos lembrar que dentro de cada posição do vetor temos um registro com aqueles campos ou variáveis definidas. Veja a figura a seguir.



Para acessar um desses campos basta colocar um ponto (.) seguido do nome da variável definida dentro do registro.

Por exemplo, para acessar o nome do primeiro hóspede (definido na posição 0 do vetor fichas), colocamos `fichas [0]. NOME`; para acessar a idade dessa mesma ficha, colocamos `fichas [0].IDADE` e assim por diante.

Supondo que estamos cadastrando o cliente 60 do nosso hotel. Como poderíamos proceder?

Assumindo que um vetor sempre inicia na posição 0 (zero), assim o cliente 60 está definido da posição 59 do nosso vetor.

Desta forma, podemos preencher os dados conforme a seguir:

```
fichas[59].NOME ← "Luiz Silva"
fichas[59].ESTADOCIVIL ← "solteiro"
fichas[59].ENDERECO ← "Avenida Paulista"
fichas[59].IDADE ← 30
{e assim para os demais dados deste hóspede}.
```

Algoritmos com manipulação de registros

Veja agora alguns exercícios resolvidos com manipulação de registros.

1. Cadastrar os dados gerais de 300 disciplinas que os professores lecionam: nome, conteúdo, frequência e nota mínimas para aprovação. Ler cinco nomes de disciplinas e mostrar frequência e média mínimas para aprovação em cada uma.

Pseudocódigo:

início

tipo FICHA = registro {criando um tipo FICHA}

NOME, CONTEUDO : literal

FREQUENCIA, MEDIA : numérico

fim-registro

DISCIPLINAS : vetor [300] FICHA {criando um vetor para cadastrar
300 disciplinas, onde cada disciplina é constituída por um nome
(NOME), conteúdo (CONTEUDO), frequência (FREQUENCIA) e média
(MEDIA)}

NOMECONSUL : literal {Nome da disciplina a ser consultada}

CONT, POS : numérico {Variáveis para controle de laço de repetição}

{Entrada de dados para 300 disciplinas}

para POS de 0 até 299 passo 1 faça

escreva "Digite o nome da disciplina:"

leia DISCIPLINAS[POS].NOME

escreva "Digite o conteúdo da disciplina:"

leia DISCIPLINAS[POS].CONTEUDO

escreva "Digite a frequência mínima necessária para aprovação:"

leia DISCIPLINAS[POS].FREQUENCIA

escreva "Digite a média mínima necessária para aprovação:"

leia DISCIPLINAS[POS].MEDIA

fim-para

{verificação de 5 disciplinas, conforme o enunciado}

para CONT de 1 até 5 passo 1 faça

escreva "Digite o nome da disciplina a ser consultada:"

leia NOMECONSUL {nome da disciplina a ser consultada}

POS \leftarrow 0 {inicializa a variável POS pois o vetor começa em 0}

enquanto NOMECONSUL < > DISCIPLINAS[POS].NOME ou POS < 300

{percorre todo o vetor DISCIPLINAS até que a variável NOMECONSUL seja igual a DISCIPLINAS [POS].NOME e POS < 300. Utilize aqui, como exercício, uma tabela-verdade para saber quando a condição resultante se torna falsa. É uma excelente revisão da operação lógica ou.}

POS \leftarrow POS + 1 {enquanto não encontrar a disciplinas digitadas, soma POS de 1, o que significa pular para a próxima posição do vetor}

fim-enquanto

se NOMECONSUL = DISCIPLINAS[POS].NOME então

escreva "Disciplina:", DISCIPLINAS[POS].NOME

escreva "Média Mínima:", DISCIPLINAS[POS].MEDIA

escreva "Freq. Mínima:", DISCIPLINAS[POS].FREQUENCIA

senão

escreva "Disciplina não consta no cadastro!"

fim-se

fim-para

fim

2. Elaborar um algoritmo para cadastrar 5000 CDs de uma loja. Os dados a serem cadastrados são: código, nome do CD, nome do cantor/grupo, tipo de música, produtora e ano de produção. Exibir os códigos e nomes dos CDs solicitados por um usuário por meio do nome de um cantor ou grupo musical.

Pseudocódigo:

início

{declaração do tipo FICHA}

tipo FICHA = registro

NOME, CANTOR, TIPO, PRODUTORA : literal

CODIGO, ANOPRODUCAO : numérico

fim-registro

{declaração do vetor CDS. Cada posição do vetor contém um registro do tipo FICHA}

CDS : vetor [5000] FICHA

CANTORCONSUL : literal {Cantor a ser consultado}

POS : numérico {variável para percorrer o vetor CDS}

{Dados de entrada para o programa de cadastramento de CDS}

para POS de 0 até 4999 passo 1 faça

escreva "Digite o código do CD:"

leia CDS[POS].CODIGO

escreva "Digite o nome do CD:"

leia CDS[POS].NOME

escreva "Digite o nome do cantor ou do grupo:"

leia CDS[POS].CANTOR

escreva "Digite o tipo de música:"

leia CDS[POS].TIPO

escreva "Digite o nome da produtora:"

leia CDS[POS].PRODUTORA

escreva "Digite o ano de produção (somente os números) do CD:"

leia CDS[POS].ANOPRODUCAO

fim-para

{processamento do algoritmo}

escreva "Digite o nome do cantor ou grupo a ser consultado (ou FIM para encerrar):"

leia CANTORCONSUL

enquanto CANTORCONSUL < > "FIM" faça

POS \leftarrow 0

enquanto CANTORCONSUL < > CDS[POS].CANTOR ou POS < 5000

{percorre cada posição do vetor para verificar a existência do cantor}

POS \leftarrow POS + 1

fim-enquanto

```

se CANTORCONSUL = CDS[POS].CANTOR então
    escreva "Código:", CDS[POS].CODIGO
    escreva "Nome do CD:", CDS[POS].NOME
senão
    escreva "Cantor ou grupo musical não possui nenhum CD cadastrado !"
fim-se
{Observe aqui que estamos solicitando que o usuário entre com o nome
do cantor ou a palavra FIM novamente. Perceba que a primeira vez ocor-
reu fora do laço de repetição. Dessa vez, o comando de leitura é realizado
para permitir com que o usuário finalize o programa digitando FIM. Caso
não colocássemos essa opção, teríamos um laço infinito. Lembre que todo
algoritmo tem que possuir um fim?}
    escreva "Digite o nome do cantor ou grupo a ser consultado (ou FIM para
encerrar):"
    leia CANTORCONSUL
fim-enquanto
fim

```

Síntese

Este módulo apresentou os conceitos de tipos de variáveis de diferentes tipos e inclusive de tipos definidos pelo programador.

Um registro, que nada mais é do que um recurso que permite a criação de diferentes tipos de variáveis em um mesmo bloco de memória do computador, facilitando, sobretudo, o desempenho do programa.

Quando criamos um registro, criamos um espaço na memória do computador que permite armazenar dados heterogêneos, ou seja, constantes de vários tipos. É como se fosse uma ficha de dados, organizada de forma que os dados estão próximos um dos outros dentro da memória do computador.

A sintaxe em pseudocódigo para criar um registro é a seguinte:

```

<nome do registro>: registro
    <campos que comporão o registro>
    fim-registro

```

Foram discutidos também a mistura de dados misturando vetores com registros.

Podemos construir vetores de registros, onde cada posição do vetor é um registro de dados. O acesso a cada elemento dentro do vetor é a seguinte:

```

<nome do vetor>[índice do vetor].<campo ou variável que compõe o registro>

```

EXERCÍCIOS

Primeira maneira:

```
início  
    NOME: vetor[50] literal  
    CPF: vetor[50] literal  
    FONE: vetor[50] numérico  
    .... {implementação do algoritmo}  
fim
```

Segunda maneira:

```
início  
    tipo FICHA = registro  
        NOME: literal  
        CPF: literal  
        FONE: numérico  
    fim-registro  
    fichas: vetor[50] FICHA {declaração de um vetor de FICHAS}  
    .... {implementação do algoritmo}  
fim
```

Explique com suas próprias palavras as diferenças entre as duas formas de implementações citadas e as vantagens em se optar pela segunda maneira.

10 - PROGRAMAÇÃO ESTRUTURADA

Introdução

Você está iniciando agora a última unidade da disciplina. Isso não significa que os estudos acabaram por aqui. Muito pelo contrário, acabaram de começar. No final desta unidade você estará pronto para começar a se tornar um excelente programador Web. O que estamos lhe oferecendo é apenas a base, independente da linguagem de programação que você irá utilizar ou já utiliza.

Você precisa entender agora um dos conceitos que foi alvo de muitos estudos em décadas passadas: a programação estruturada ou programação modular. Esse tipo de programação surgiu quando ocorreu a crise do software, por volta da década de 60. Os programas de computadores se tornaram grandes e caros, principalmente em relação à manutenção dos mesmos. Não havia metodologia para a criação de programas de computadores. A probabilidade de acontecerem erros na programação era enorme, mesmo depois de o programa estar funcionando. Eram necessárias manutenções diárias, aumentando substancialmente os custos de desenvolvimento.

A tecnologia mudou, o mundo mudou, os problemas também mudaram. Temos um grau bem maior de complexidade, e a pressão por redução de custos fez com que os programadores adotassem metodologias de desenvolvimento de software. Uma dessas metodologias é a programação estruturada. O objetivo principal da programação estruturada é decompor o problema em partes ou blocos, fazendo com que cada bloco execute uma função específica e, juntos, esses blocos formam o programa como um todo. A proposta é desenvolver os algoritmos em partes integradas, de forma a adquirirem maior legibilidade, facilidade de uso e manutenção.

Por isso, nossa meta nesta unidade é a de decompor os problemas em partes. Vamos dividir para poder conquistar, ou seja, vamos transformar nossos algoritmos em uma forma bem mais estruturada do que temos feito até agora.

O que você acha de iniciar, então?

Modularização: conceitos iniciais

Quando nos deparamos com problemas complexos, a melhor maneira de solucioná-los é decomporlo em partes, ou seja, resolver o problema em etapas para que possamos ter êxito no final. A cada uma dessas partes bem definidas, que contribuem para a solução do problema, chamamos de módulo.

Assim também são nossas soluções lógicas. Caso separemos um problema em partes, podemos construir algoritmos para cada uma das partes de forma bem definida e independentes uma das outras. A essa técnica de dividir os algoritmos em módulos chamados de modularização.

- Retome primeiro o conceito de algoritmo em Lógica de Programação I.

Para lembrar!

Um algoritmo é formalmente uma seqüência infinita de passos que levam a execução de uma tarefa. Podemos pensar em algoritmo como uma receita, uma seqüência de instruções, que tem a função de atingir uma meta específica. Estas tarefas não podem ser redundantes nem subjetivas na sua definição, devendo ser claras e precisas.

Sendo assim, cada parte ou módulo construído deve ter as mesmas características citadas, devendo cada módulo:

Implementar tarefas não-redundantes e nem subjetivas.

Ter uma função específica e clara.

Ser independente de todos os outros módulos de um programa.

Ser testado e corrigido antes de ser integrado à solução completa tendo em vista que é parte da solução lógica do problema.

Poder ser utilizado em outros algoritmos que requerem os mesmos procedimentos.

Veja um exemplo:

Imagine que você seja responsável por construir uma página na intranet de uma empresa prestadora de serviços. O programa deverá realizar a entrada de dados para cada funcionário da empresa, irá executar o cálculo da folha de pagamento e, por fim, irá emitir os contracheques e a relação bancária.

Parece um problema bem mais complexo dos que temos visto até agora, você concorda? Pois bem, se quisermos implementar as mesmas soluções lógicas até agora estudadas, vamos ter um programa grande e complexo, sujeito a falhas e erros. Podemos separar o problema em partes, construir pequenos módulos, testá-los e depois integrá-los para alcançar o objetivo final. Com esse procedimento, a complexidade dos algoritmos diminui, os testes se tornam rápidos e a manutenção também diminui. Pode parecer estranho, mas os resultados são de arregalar os olhos. Voltando ao problema, temos que o nosso alvo é a construção de um programa de mostrar a folha de pagamento. Vamos então dividi-lo conforme figura a seguir:

FolhaPagamento constitui o nosso programa principal. Esse programa é constituído de módulos chamados de EntradaDados, Movimento e Saídas. Por sua vez, o Módulo EntradaDados é constituído de dois outros módulos: Funcionários e Dependentes. Cada módulo resolve um pequeno problema e pode ser solucionado por uma pessoa diferente, diminuindo o tempo de desenvolvimento.

O módulo Saída mostra a formatação do texto e formulário no site da empresa. O módulo EntradaDados permite que os dados cadastrais de funcionários e/ou dependentes sejam entrados no programa. Cada um desses módulos deve funcionar de maneira independente e precisa, ou seja, sempre que precisarmos imprimir algo na tela, chamamos o módulo responsável por isso, o módulo Saída. Se precisarmos cadastrar um novo funcionário, chamamos o módulo EntradaDados, e assim sucessivamente. Tudo funciona por partes. Não preciso chamar o módulo Movimento, se meu interesse é apenas mostrar os resultados na saída ou na página da empresa.

Um fato importante: Os módulos são independentes, mas todos eles estão integrados ao módulo que chamamos de módulo principal do programa em nosso exemplo o módulo FolhaPagamento. A partir desse módulo é que os outros são chamados para resolver problemas como entrada de dados, impressão de resultados etc.

Lembra como era anteriormente? Entrávamos com os dados, processávamos o algoritmo e mostrávamos o resultado na saída? Agora, as coisas se tornaram independentes. Chamamos apenas o módulo que nos convier naquele momento, na sequência em que quisermos. É realmente uma forma estruturada de trabalhar, percebeu?

De forma resumida, nossos programas terão agora um módulo principal, chamado de programa principal e diversos outros módulos que irão compor todo o programa. Cada módulo é chamado a partir do programa principal. Quando um módulo é chamado a partir do programa principal, os comandos definidos dentro do módulo são executados. Quando finalizados, o controle volta para o programa principal.

Nas linguagens de programação, cada módulo é chamado de função.

Mas, o que é uma função?

Uma função é dita como sendo uma sub-rotina ou módulo do algoritmo principal. Cada função é independente uma da outra. Chamamos uma função a partir do algoritmo principal. Dentro da função são executados comandos que realizam alguma tarefa específica. Quando todos os comandos dentro da função forem executados, o controle do programa volta para o programa principal. A partir do programa principal, podemos chamar outras funções de interesse também.

Por que usar funções?

Para permitir o reaproveitamento de código já construído por você ou por outros programadores.
Para evitar que um trecho de código que seja repetido várias vezes dentro de um mesmo programa.

Para permitir a alteração de um trecho de código de uma forma mais rápida. Com o uso de uma função é preciso alterar apenas dentro da função que se deseja.

Para que os blocos do programa não fiquem grandes demais e, por consequência, mais difíceis de entender.

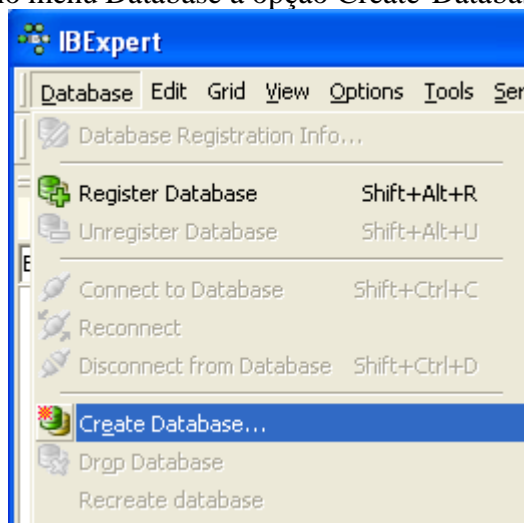
Para facilitar a leitura do programa de uma forma mais fácil.

Para separar o programa em partes (blocos) que possam ser logicamente compreendidas de forma isolada.

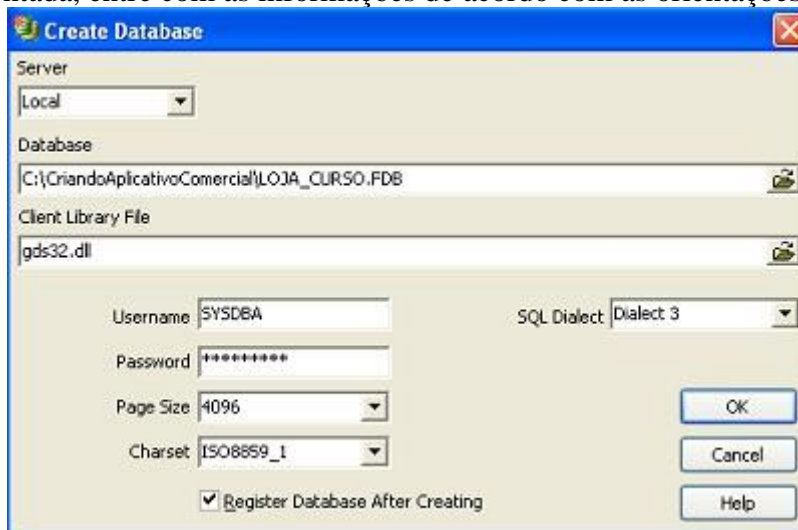
11 - CRIANDO O BANCO DE DADOS

Vamos definir o banco de dados da aplicação. Para isso, siga os passos abaixo:

- Abra o editor de banco de dados IBExpert (Barra de Tarefas, Iniciar, Programas, HK-Software, IBExpert);
- No IBExpert, selecione no menu Database a opção Create Database;

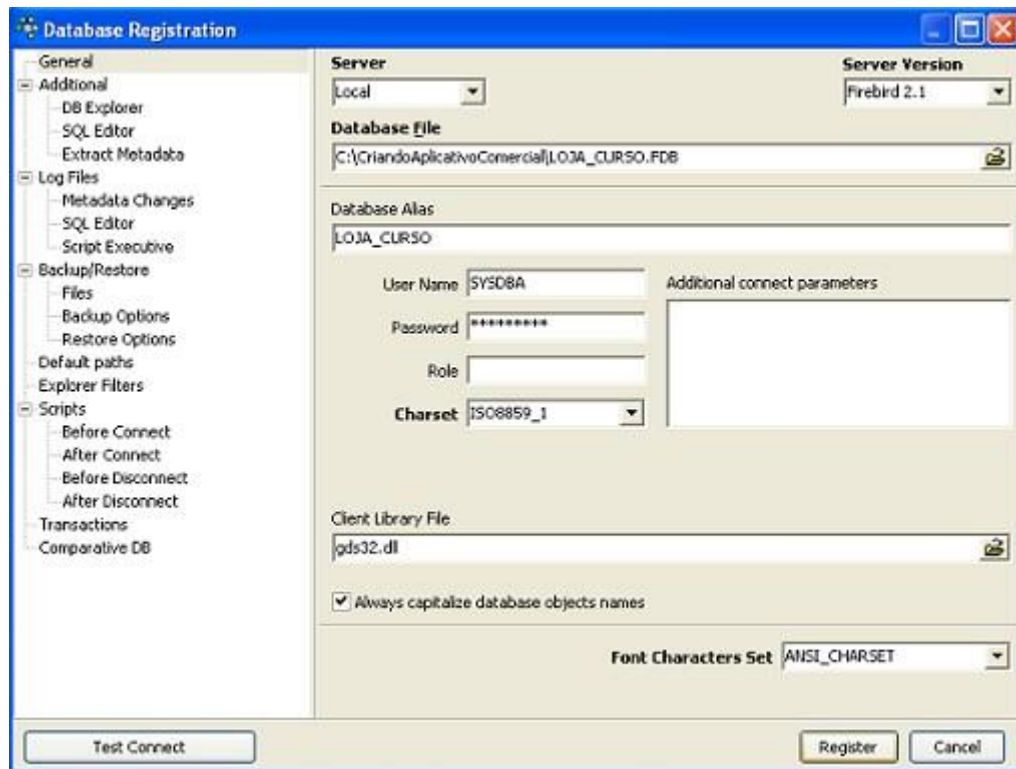


- Na tela apresentada, entre com as informações de acordo com as orientações abaixo:



Databasename: C:\CriandoAplicativoComercial\LOJA_CURSO.fdb
User Name: SYSDBA
Password: masterkey
Confirm Password: masterkey
Page Size: 4096
Char Set: DOS857
Database SQL Dialect: selecione Dialect 3
Register After Creating: Marque a caixa
Finalize pressionando o botão OK.

- Será apresentada a tela a seguir:



- Na tela apresentada, entre com as informações abaixo:

Na aba General:

Campo Database Alias: LOJA_CURSO (apelido do banco no IBExpert)
 Campo User Name: SYSDBA (nome do usuário do banco)
 Campo Password: masterkey (senha do usuário no banco)
 Campo Charset: Selecione a opção ISO8859-1 (conjunto de caracteres)

Na aba Additional:

Selecione a caixa Auto Commit Transactions

- Em seguida clique no botão “Test Connect” para efetuar o teste de conexão com o banco de dados (a mensagem “Connected” deverá ser exibida, caso contrário, verifique se as informações acima solicitadas estão corretas e efetue um novo teste de conexão).
- Após o teste de conexão (Connected), clique sobre o botão Register.
- Deverá ser apresentado o Alias LOJA_CURSO, na aba Databases do DBExplorer do IBExpert (tecla de atalho para o DBExplorer: F11);
- Em seguida, dê um duplo clique do mouse sobre o Alias LOJA_CURSO, no quadro Data Bases;
- No DBExplorer será aberta a seguinte estrutura:

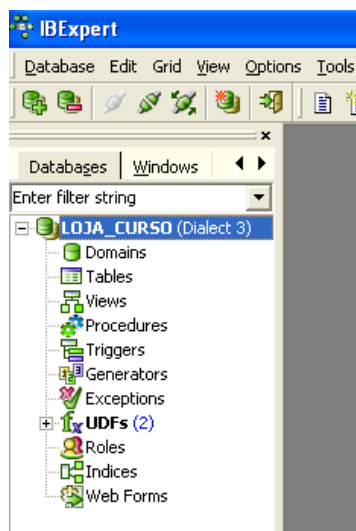


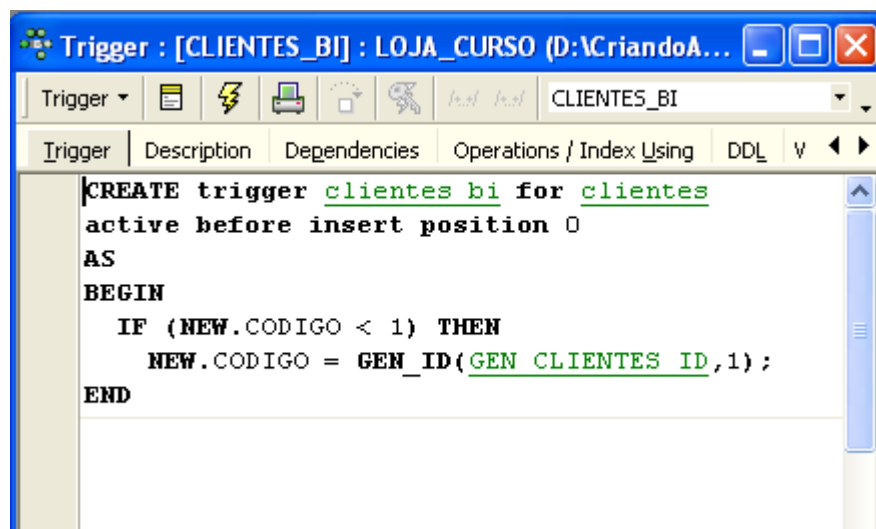
Tabela de Clientes

Inicialmente, vamos criar a tabela que armazenará as informações de todos os clientes do nosso Aplicativo Comercial.

- Ainda no IBExpert, selecione o item de menu Database, submenu New table;
- Especifique os seguintes campos para a tabela de clientes conforme a tela a seguir:

#	FK	PK	Field Name	Field Type	Domain	Size	Scale	Subtype	Array	Not Null
1		<input checked="" type="checkbox"/>	CODIGO	INTEGER						<input checked="" type="checkbox"/>
2			CNPJ	VARCHAR		19				<input type="checkbox"/>
3			NOME	VARCHAR		30				<input type="checkbox"/>
4			ENDERECO	VARCHAR		35				<input type="checkbox"/>
5			COMPLEMENTO	VARCHAR		50				<input type="checkbox"/>
6			CIDADE	VARCHAR		25				<input type="checkbox"/>
7			ESTADO	CHAR		2				<input type="checkbox"/>
8			CEP	VARCHAR		9				<input type="checkbox"/>
9			FONE_RESIDENCIAL	VARCHAR		20				<input type="checkbox"/>
10			FONE_COMERCIAL	VARCHAR		20				<input type="checkbox"/>
11			RENDIA_FAMILIAR	NUMERIC		18	2			<input type="checkbox"/>

- No campo CODIGO, clique 2 vezes sobre a coluna PK, para criar a chave primária da tabela, não se esqueça de especificar (ligar) a coluna Not Null;
- Na linha do campo CODIGO, procure a coluna Auto Inc, e dê um duplo clique sobre ela;
- Será apresentada uma janela, onde deve ser ligada a caixa Create New Generator;
- Clique na aba Trigger e ligue a caixa Create Trigger;
- Na janela apresentada, altere o texto IS NULL por < 1 e pressione o botão OK;



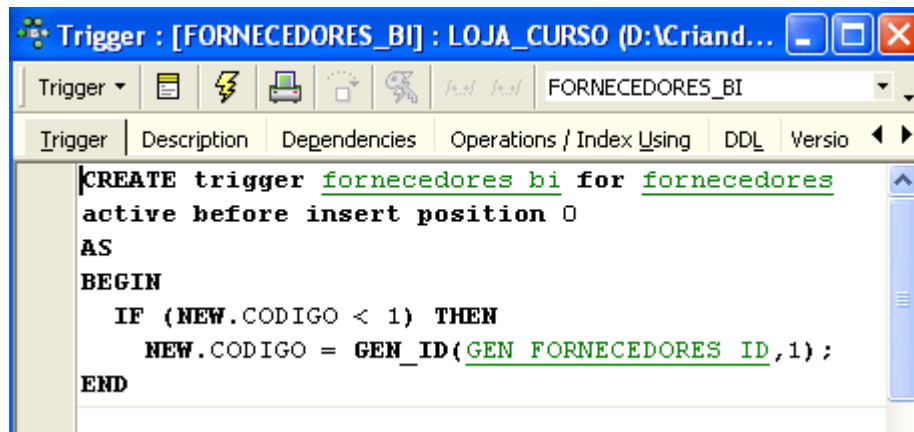
- Pressione o botão OK;
- Clique sobre o botão Compile (CTRL+F9, figura de um raio);
- Deve ser apresentada uma tela mostrando os comandos SQL gerados para a criação desta tabela.
- Clique sobre o botão Commit.

Tabela de Fornecedores

- Selecione o item de menu Database e New table;
- Especifique os seguintes campos para esta tabela;

#	FK	PK	Field Name	Field Type	Domain	Size	Scale	Subtype	Array	Not Null
1		<input checked="" type="checkbox"/>	CODIGO	INTEGER						<input checked="" type="checkbox"/>
2			CNPJ	VARCHAR		19				<input type="checkbox"/>
3			NOME	VARCHAR		30				<input type="checkbox"/>
4			ENDERECO	VARCHAR		35				<input type="checkbox"/>
5			COMPLEMENTO	VARCHAR		50				<input type="checkbox"/>
6			CIDADE	VARCHAR		25				<input type="checkbox"/>
7			ESTADO	CHAR		2				<input type="checkbox"/>
8			CEP	VARCHAR		9				<input type="checkbox"/>
9			FONE	VARCHAR		19				<input type="checkbox"/>
10			RESPONSAVEL	VARCHAR		20				<input type="checkbox"/>

- No campo CODIGO, clique 2 vezes sobre a coluna PK, para criar a chave primária da tabela, não se esqueça de especificar (ligar) a coluna Not Null;
- Na linha do campo CODIGO, procure a coluna Auto Inc, e dê um duplo clique sobre ela;
- Será apresentada uma janela, onde deve ser ligada a caixa Create New Generator;
- Clique na aba Trigger e ligue a caixa Create Trigger;
- Na janela apresentada, altere o texto IS NULL por < 1 e pressione o botão OK;



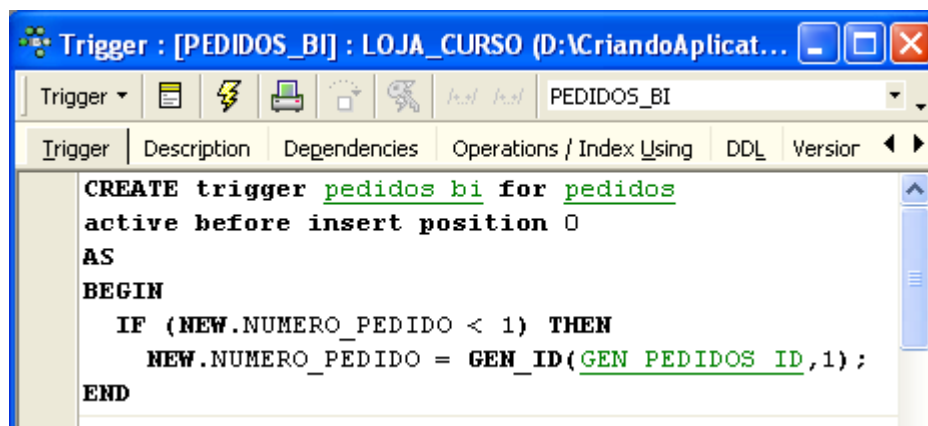
- Pressione o botão OK;
- Clique sobre o botão Compile (CTRL+F9, figura de um raio);
- Deve ser apresentada uma tela mostrando os comandos SQL gerados para a criação desta tabela.
- Clique sobre o botão Commit.

Tabela de Pedidos

- Selecione o item de menu Database e New table;
- Especifique os seguintes campos para esta tabela;

#	FK	PK	Field Name	Field Type	Domain	Size	Scale	Subtype	Array	Not Null
1		<input checked="" type="checkbox"/>	NUMERO_PEDIDO	INTEGER						<input checked="" type="checkbox"/>
2		<input type="checkbox"/>	DATA_PEDIDO	DATE						<input type="checkbox"/>
3		<input type="checkbox"/>	DATA_RECEBIMENTO	DATE						<input type="checkbox"/>
4		<input type="checkbox"/>	PRECO_TOTAL	NUMERIC		18	2			<input type="checkbox"/>
5		<input type="checkbox"/>	CODIGO_FORNECEDOR	INTEGER						<input type="checkbox"/>

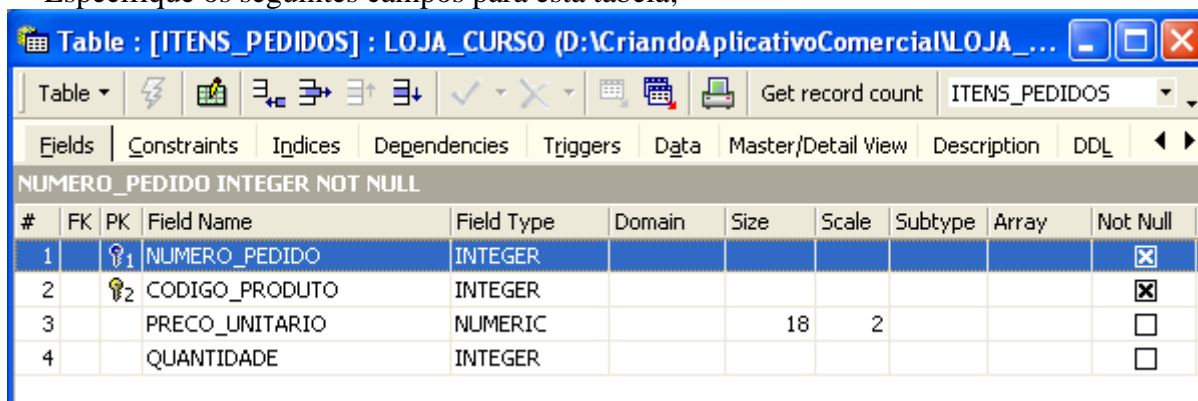
- No campo NUMERO_PEDIDO, clique 2 vezes sobre a coluna PK, para criar a chave primária da tabela, não se esqueça de especificar (ligar) a coluna Not Null;
- Na linha do campo NUMERO_PEDIDO, procure a coluna Auto Inc, e dê um duplo clique sobre ela;
- Será apresentada uma janela, onde deve ser ligada a caixa Create New Generator;
- Clique na aba Trigger e ligue a caixa Create Trigger;
- Na janela apresentada, altere o texto IS NULL por < 1;



- Pressione o botão OK;
- Clique sobre o botão Compile (CTRL+F9, figura de um raio);
- Deve ser apresentada uma tela mostrando os comandos SQL gerados para a criação desta tabela.
- Clique sobre o botão Commit.

Tabela de Itens do Pedido

- Selecione o item de menu Database e New table;
- Especifique os seguintes campos para esta tabela;



- No campo NUMERO_PEDIDO, clique 2 vezes sobre a coluna PK, para criar a chave primária da tabela, não se esqueça de especificar (ligar) a coluna Not Null;
- Clique sobre o botão Compile (CTRL+F9, figura de um raio);
- Deve ser apresentada uma tela mostrando os comandos SQL gerados para a criação desta tabela.
- Clique sobre o botão Commit.

Tabela de Produtos

- Selecione o item de menu Database e New table;
- Especifique os seguintes campos para esta tabela;

#	FK	PK	Field Name	Field Type	Domain	Size	Scale	Subtype	Array	Not Null
1			CODIGO	INTEGER						<input checked="" type="checkbox"/>
2			DESCRICAO	VARCHAR		35				<input type="checkbox"/>
3			ESTOQUE_ATUAL	INTEGER						<input type="checkbox"/>
4			ESTOQUE_MINIMO	INTEGER						<input type="checkbox"/>

- No campo CODIGO, clique 2 vezes sobre a coluna PK, para criar a chave primária da tabela, não se esqueça de especificar (ligar) a coluna Not Null;
- Na linha do campo CODIGO, procure a coluna Auto Inc, e dê um duplo clique sobre ela;
- Será apresentada uma janela, onde deve ser ligada a caixa Create New Generator;
- Clique na aba Trigger e ligue a caixa Create Trigger;
- Na janela apresentada, altere o texto IS NULL por < 1;

```

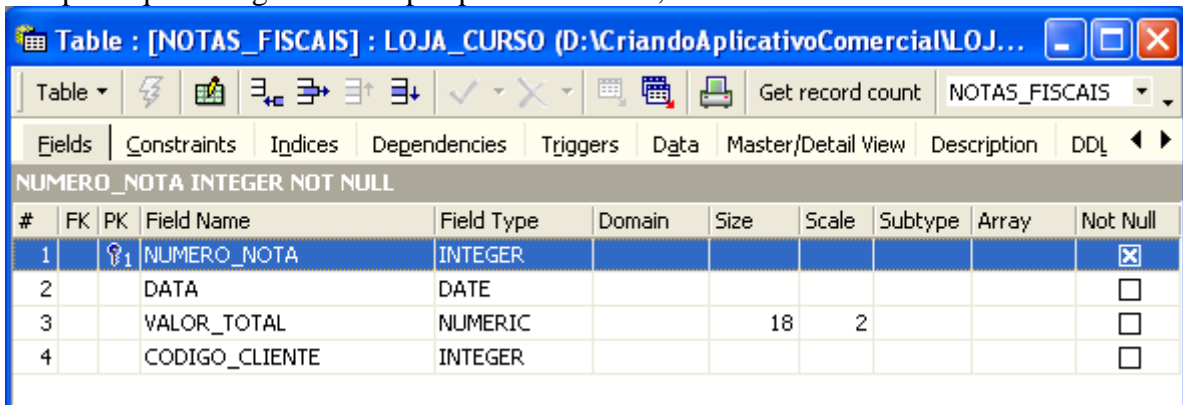
CREATE trigger produtos bi for produtos
active before insert position 0
AS
BEGIN
    IF (NEW.CODIGO < 1) THEN
        NEW.CODIGO = GEN_ID(GEN PRODUTOS ID,1);
    END

```

- Pressione o botão OK;
- Clique sobre o botão Compile (CTRL+F9, figura de um raio);
- Deve ser apresentada uma tela mostrando os comandos SQL gerados para a criação desta tabela.
- Clique sobre o botão Commit.

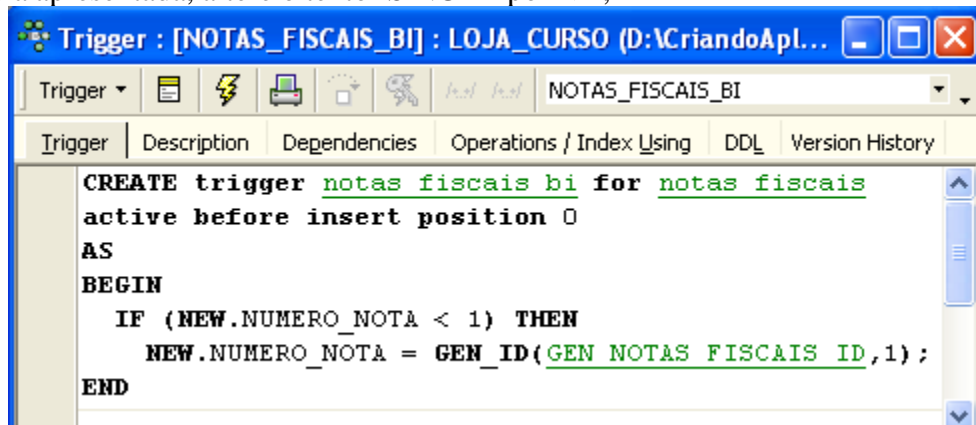
Tabela de Notas Fiscais (Vendas)

- Selecione o item de menu Database e New table;
- Especifique os seguintes campos para esta tabela;



#	FK	PK	Field Name	Field Type	Domain	Size	Scale	Subtype	Array	Not Null
1			NUMERO_NOTA	INTEGER						<input checked="" type="checkbox"/>
2			DATA	DATE						<input type="checkbox"/>
3			VALOR_TOTAL	NUMERIC		18	2			<input type="checkbox"/>
4			CODIGO_CLIENTE	INTEGER						<input type="checkbox"/>

- No campo NUMERO_NOTA, clique 2 vezes sobre a coluna PK, para criar a chave primária da tabela, não se esqueça de especificar (ligar) a coluna Not Null;
- Na linha do campo NUMERO_NOTA, procure a coluna Auto Inc, e dê um duplo clique sobre ela;
- Será apresentada uma janela, onde deve ser ligada a caixa Create New Generator;
- Clique na aba Trigger e ligue a caixa Create Trigger;
- Na janela apresentada, altere o texto IS NULL por < 1;

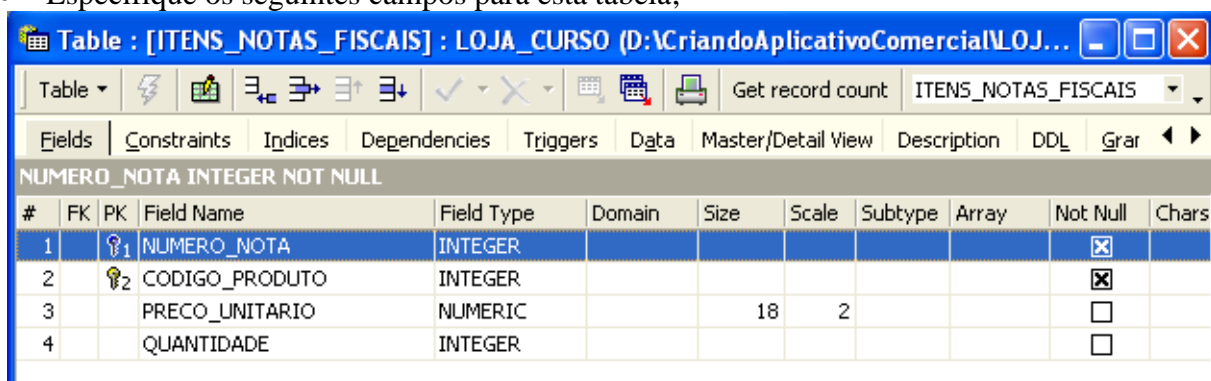


```
CREATE trigger notas fiscais bi for notas fiscais
active before insert position 0
AS
BEGIN
  IF (NEW.NUMERO_NOTA < 1) THEN
    NEW.NUMERO_NOTA = GEN_ID(GEN NOTAS FISCAIS ID,1);
  END
```

- Clique sobre o botão Compile (CTRL+F9, figura de um raio);
- Deve ser apresentada uma tela mostrando os comandos SQL gerados para a criação desta tabela.
- Clique sobre o botão Commit.

Tabela de Itens de Notas Fiscais

- Selecione o item de menu Database e New table;
- Especifique os seguintes campos para esta tabela;



#	FK	PK	Field Name	Field Type	Domain	Size	Scale	Subtype	Array	Not Null	Chars
1			NUMERO_NOTA	INTEGER						<input checked="" type="checkbox"/>	
2			CODIGO_PRODUTO	INTEGER						<input checked="" type="checkbox"/>	
3			PRECO_UNITARIO	NUMERIC		18	2			<input type="checkbox"/>	
4			QUANTIDADE	INTEGER						<input type="checkbox"/>	

- No campo NUMERO_NOTA, clique 2 vezes sobre a coluna PK, para criar a chave primária da tabela, não se esqueça de especificar (ligar) a coluna Not Null;
- Clique sobre o botão Compile (CTRL+F9, figura de um raio);
- Deve ser apresentada uma tela mostrando os comandos SQL gerados para a criação desta tabela.
- Clique sobre o botão Commit.

EXERCÍCIO

- Crie uma tabela para armazenar os dados de um cadastro de funcionários.

12 - INTRODUÇÃO À ORIENTAÇÃO A OBJETOS

Conceito

O termo orientação a objetos pressupõe uma organização de software em termos de coleção de objetos discretos incorporando estrutura e comportamento próprios.

Esta abordagem de organização é essencialmente diferente do desenvolvimento tradicional de software, onde estruturas de dados e rotinas são desenvolvidas de forma apenas fracamente acopladas. Implementa-se um conjunto de classes que definem os objetos presentes no sistema de software. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.

Na programação orientada a objetos o programador é responsável por moldar o mundo dos objetos, e explicar para estes objetos como eles devem interagir entre si. Os objetos "conversam" uns com os outros através do envio de mensagens, e o papel principal do programador é especificar quais serão as mensagens que cada objeto pode receber, e também qual a ação que aquele objeto deve realizar ao receber aquela mensagem em específico.

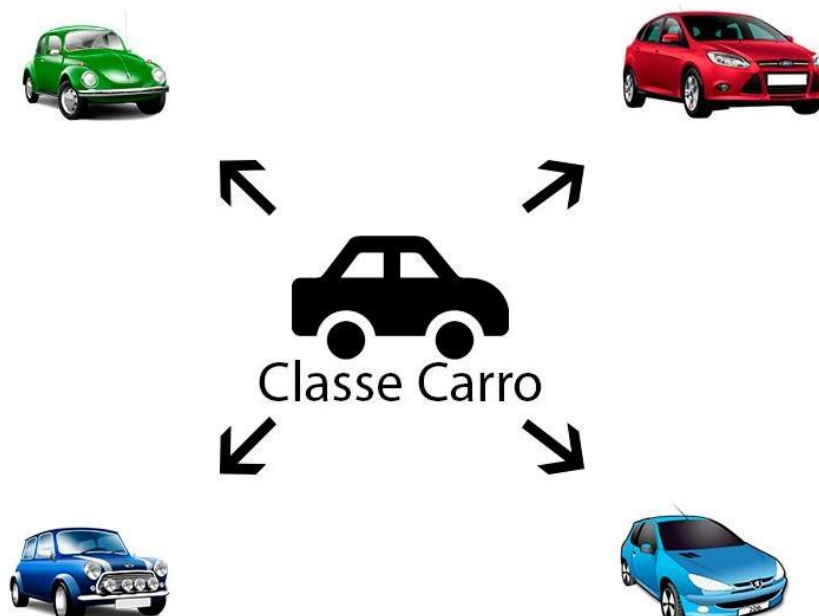
Classe

Na orientação a objetos precisamos realizar comparativos com a real, por esse motivo iremos abordar exemplos simples para compreender o conceito de classe.

Primeiramente devemos referenciar a classe como uma espécie de molde, onde o desenvolvedor deverá criar um objeto com determinadas características.

Vamos ao nosso primeiro exemplo, imagine que um colega seu diz que vai comprar um carro, a expressão carro é bem genérica já que temos à disposição uma quantidade expressiva de carros à disposição, veja que em nenhum momento foi dito qual é o carro que será comprado pelo colega.

O exemplo abaixo ilustrará de maneira simplificada o exemplo utilizado acima:



Objeto

O objeto nada mais é que o resultado obtido no molde da classe carro, cada objeto possui suas próprias características, além de poder realizar ações.

Atributo

Atributos são as características dos objetos, retomando o exemplo exemplificar a classe, podemos verificar que cada carro possui atributos diferenciados sendo eles: cor, fabricante, modelo, ano, entre outras informações.

Método

São as ações que os objetos podem executar, um carro por exemplo pode ter os seguintes métodos: buzinar, andar, frear, abrir porta, entre outros

Para compreender melhor esses quatro termos vamos fazer um exemplo utilizando uma classe carro, um objeto fusca e alguns atributos e métodos.



EXEMPLO PRÁTICO

Para compreender melhor os temas abordados iremos desenvolver uma classe carro, um objeto fusca e adicionar seus atributos e métodos utilizando a linguagem de programação JAVA, porém os conceitos vistos nesse módulo poderão ser aplicados nas demais linguagens que ofereçam suporte a orientação a objetos.

Para essa atividade iremos criar um molde para trabalharmos com carros, vale lembrar que é no molde em que o desenvolvedor deverá adicionar as características (atributos), e adicionar todas as funções (métodos) que a classe pode utilizar para realizar determinada ação.

1ª Etapa - Criação da classe Carro:

```
1 //Criando a classe Carro
2 class Carro {
3
4 }
```

2ª Etapa – Adicionando os atributos na classe Carro:

```
1 //Criando a classe Carro
2 class Carro {
3
4     //Atributos da classe Carro
5     String cor;
6     String modelo;
7     String fabricante;
8
9 }
```

3ª Etapa – Criando um método para exibir a cor que será informada ao criar o objeto:

```
1 //Criando a classe Carro
2 class Carro {
3
4     //Atributos da classe Carro
5     String cor;
6     String modelo;
7     String fabricante;
8
9     //Método exibirCor
10    void exibirCor(){
11        System.out.print("A cor do seu carro é: "+this.cor);
12    }
13 }
```

4ª Etapa – Criando a classe TestarCarro:

```
1 //Criando a classe TestarCarro
2 class TestarCarro {
3
4     //Inicializando
5     public static void main(String[] args) {
6
7         //Criando o objeto e adicionando os atributos
8         Carro meuCarro;
9         meuCarro = new Carro();
10        meuCarro.cor = "Verde";
11        meuCarro.modelo = "Fusca";
12        meuCarro.fabricante = "Volkswagem";
13
14        //Método - Exibir cor
15        meuCarro.exibirCor();
16    }
17
18 }
```

Após realizado esse exercício pratico, teremos desenvolvido nossa primeira aplicação utilizando a Orientação a Objetos. Vale ressaltar que o mesmo exemplo pode ser utilizado em outras linguagens

com suporte a Orientação a Objetos, porém cada linguagem terá sua particularidade, sendo assim basta apenas adaptar o código a linguagem.

A grande vantagem em utilizar a programação Orientada a Objetos é sua organização, mesmo o código sendo extenso é possível perceber a facilidade em compreender cada passo que a aplicação deverá realizar.

Construtores

Quando usamos a palavra reservada `new`, estamos construindo um objeto. Sempre quando o `new` é utilizado automaticamente é executado o construtor da classe, que tem como função realizar alguma ação pré-definida pelo programador.

Isso se deve ao fato de que um objeto deve ser construído cada vez que chamamos a classe. E a responsabilidade de fazer isso é do construtor. Isso parte do princípio que podemos ter dois objetos com a mesma característica, mas que não são os mesmos objetos.

Vamos realizar um exemplo simples, tendo como base o exemplo prático realizado anteriormente:

1ª Criar a classe Carro:

```
1 //Criando a classe Carro
2 class Carro {
3
4     //Atributos da classe Carro
5     String cor;
6     String modelo;
7     String fabricante;
8
9     //Construtor
10    public Carro(String cor, String modelo, String fabricante){
11        this.cor = cor;
12        this.modelo = modelo;
13        this.fabricante = fabricante;
14    }
15
16    //Método exibirCor
17    void exibirCor() {
18        System.out.println("Seu carro é da cor: "+this.cor);
19    }
20 }
```

2ª Criar a classe TestarCarro:

```
1 //Criando o objeto
2 class TestarCarro {
3
4     //Inicializando
5     public static void main(String[] args) {
6         //Criando o objeto e adicionando os atributos
7         Carro meuCarro;
8         meuCarro = new Carro("Verde", "Fusca", "Volkswagem");
9
10        //Método - Exibir cor
11        meuCarro.exibirCor();
12    }
13 }
```

Herança

Enquanto programamos em Java, há a necessidade de trabalharmos com várias classes. Muitas vezes, classes diferentes tem características comuns, então, ao invés de criarmos uma nova classe com todas essas características usamos as características de um objeto ou classe já existente.

Em outras palavras, o conceito de herança pode ser definido como: é uma classe derivada de outra classe.

Para compreender melhor esse conceito de reutilização, iremos criar as seguintes classes: Pessoa, Funcionário e Cliente. A classe pessoa será a classe principal, já a Funcionário e Cliente serão classes que herdarão determinado atributo e/ou método da classe Pessoa.

1º Criando a classe Pessoa, adicionando seus respectivos atributos e criando um método:

```
1 //Criando a classe Pessoa
2 public class Pessoa {
3     //Atributos
4     String nome,
5     int idade,
6     String endereco= "R: Java ,501";
7
8
9     //Método ImprimeNome
10    public void ImprimeNome(){
11        System.out.println("o nome é:");
12        System.out.println("Endereco: " + endereco);
13    }
14
15 }
```

2º Criando a classe Fornecedor e estendendo informações da classe Pessoa.

```
1 //Criando a classe Fornecedor e estendendo dados da classe Pessoa
2 public class Fornecedor extends Pessoa {
3
4     //Atributo
5     String cnpj;
6
7     //Método
8     public void ImprimeNome (){
9         System.out.println("O nome do fornecedor é : "
10             + nome + "\n Cnpj: " + cnpj);
11     }
12
13 }
```

3º Criando a classe Cliente e estendendo as informações da classe Pessoa

```
1 //Criando a classe Cliente e estendendo dados da classe Pessoa
2 public class Cliente extends Pessoa {
3
4     //Atributo
5     String cpf;
6
7     //Método
8     public void ImprimeNome (){
9         System.out.println("Nome do cliente é : " + nome
10                             + "\n Nº CPF: " + cpf
11                             + "\n Seu endereço : " + endereço);
12     }
13
14 }
```

4º Criando a classe Principal, que terá como função criar dois objetos e repassar os parâmetros necessários para sua execução, além de invocar os métodos desejados.

```
1 //Criando a classe Principal que irá executar nossa aplicação
2 public class Principal {
3
4     //Inicialização
5     public static void main(String[] args) {
6
7         //Objeto Cliente
8         Cliente c = new Cliente();
9
10        //Atributos do objeto Cliente
11        c.nome="Luiz";
12        c.cpf="073.777.796-21";
13
14        //Método que deverá utilizar como parâmetro o objeto Cliente
15        c.ImprimeNome();
16
17
18
19
20        //Objeto Fornecedor
21        Fornecedor f = new Fornecedor ();
22
23        //Atributos do objeto Fornecedor
24        f.nome="Proway";
25        f.cnpj="073.856.9856-52-10";
26
27        //Método que deverá utilizar como parâmetro o objeto Fornecedor
28        f.ImprimeNome();
29
30    }
```

Considerações Finais

Os tópicos abordados nessa apostila sobre Orientação a Objetos são destinados para compreender os conceitos básicos da linguagem que são:

1. Classes
2. Objetos
3. Atributos
4. Métodos
5. Construtores
6. Herança

Ainda há outros conceitos que devem ser implementados para poder utilizar o máximo que a Orientação a Objetos dispõe, para isso é interessante o desenvolver estar familiarizado com os conceitos básicos citados anteriormente, e em seguida implementá-los na linguagem de sua preferência.